LNCS 3350

Manuel Hermenegildo
Daniel Cabeza (Eds.)

Practical Aspects of Declarative Languages

7th International Symposium, PADL 2005
Long Beach, CA, USA, January 2005
Proceedings

Springer

Manuel Hermenegildo Daniel Cabeza (Eds.)

Practical Aspects of Declarative Languages

7th International Symposium, PADL 2005
Long Beach, CA, USA, January 10-11, 2005
Proceedings

Springer

Volume Editors

Manuel Hermenegildo
University of New Mexico, Department of Computer Science
MSC 01 1130, Albuquerque, NM 87131, USA
E-mail: herme@unm.edu
and
Technical University of Madrid, Department of Computer Science
28660 Boadilla del Monte, Madrid, Spain
E-mail: herme@fi.upm.es

Daniel Cabeza
Technical University of Madrid, Department of Computer Science
28660 Boadilla del Monte, Madrid, Spain
E-mail: dcabeza@fi.upm.es

Library of Congress Control Number: 2004117186

CR Subject Classification (1998): D.3, D.1, F.3, D.2

ISSN 0302-9743
ISBN 3-540-24362-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11377474 06/3142 5 4 3 2 1 0

Preface

The International Symposium on Practical Aspects of Declarative Languages (PADL) is a forum for researchers and practioners to present original work emphasizing novel applications and implementation techniques for all forms of declarative concepts, including functional, logic, constraints, etc. Declarative languages build on sound theoretical foundations to provide attractive frameworks for application development. These languages have been successfully applied to a wide array of different real-world situations, including database management, active networks, software engineering, decision support systems, or music composition; whereas new developments in theory and implementation have opened up new application areas. Inversely, applications often drive the progress in the theory and implementation of declarative systems, as well as benefit from this progress.

The 7th PADL Symposium was held in Long Beach, California on January 10–11, 2005, and was co-located with ACM's Principles of Programming Languages (POPL). From 36 submitted papers, the Program Committee selected 17 papers for presentation at the symposium, based upon at least three reviews for each paper, provided from Program Committee members and additional referees.

Two invited talks were presented at the conference: one by Norman Ramsey (Harvard University) entitled "Building the World from First Principles: Declarative Machine Descriptions and Compiler Construction"; and a second by Saumya Debray (University of Arizona) entitled "Code Compression."

Following what has become a tradition in PADL symposia, the Program Committee selected one paper to receive the "Most Practical Paper" award. This year the paper judged the best in terms of practicality, originality, and clarity was "A Provably Correct Compiler for Efficient Model Checking of Mobile Processes," by Ping Yang, Yifei Dong, C.R. Ramakrishnan, and Scott A. Smolka. This paper presents an optimizing compiler for the pi-calculus that improves the efficiency of model-checking specifications in a logic-programming-based model checker.

The PADL symposium series is sponsored in part by the Association for Logic Programming (`http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/`) and COMPULOG Americas (`http://www.cs.nmsu.edu/~complog/`). Thanks are also due to the University of Texas at Dallas for its support. Finally, we want to thank the authors who submitted papers to PADL 2005 and all who participated in the conference.

November 2004

Manuel Hermenegildo
Daniel Cabeza

Program Chairs

Manuel Hermenegildo	University of New Mexico, USA *and* Technical University of Madrid, Spain
Daniel Cabeza	Technical University of Madrid, Spain

Program Committee

Kenichi Asai	Ochanomizu University, Japan
Manuel Carro	Technical University of Madrid, Spain
Bart Demoen	K.U.Leuven, Belgium
Robert Findler	University of Chicago, USA
John Gallagher	Roskilde University, Denmark
Hai-Feng Guo	University of Nebraska at Omaha, USA
Gopal Gupta	U. of Texas at Dallas, USA (General Chair)
Chris Hankin	Imperial College London, UK
Joxan Jaffar	National U. of Singapore, Singapore
Alan Mycroft	Cambridge University, UK
Gopalan Nadathur	U. of Minnesota, USA
Lee Naish	U. of Melbourne, Australia
Simon Peyton-Jones	Microsoft Research, USA
John Reppy	University of Chicago, USA
Morten Rhiger	Roskilde University, Denmark
Francesca Rossi	University of Padova, Italy
Vitor Santos-Costa	U. Federal do Rio de Janeiro, Brazil
Terrance Swift	S.U. of New York at Stony Brook, USA
David S. Warren	S.U. of New York at Stony Brook, USA

Referees

Maurice Bruynooghe
Ins de Castro Dutra
Chiyan Chen
Henning Christiansen
Gregory Cooper
Yifei Dong
Mrio Florido
David Greaves
ngel Herranz
Bharat Jayaraman
Siau-Cheng Khoo
Ricardo Lopes
Noelia Maya
Dale Miller
Rudradeb Mitra
Andrew Moss
Pablo Nogueira
Michael O'Donnell
Bernard Pope
Ricardo Rocha
Mads Rosendahl
Abhik Roychoudhury
Tom Schrijvers
David Scott
Mark Shinwell
Leon Sterling
Tom Stuart
Peter Stuckey
Eric Van Wyk
Kristen Brent Venable
Joost Vennekens
Razvan Voicu
Hongwei Xi

Table of Contents

Building the World from First Principles: Declarative Machine Descriptions and Compiler Construction (Abstract)

Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
`http://www.eecs.harvard.edu/~nr`

For at least 25 years, the most effective way to retarget systems software has been by using machine descriptions. But "machine description" doesn't mean what you think. A traditional machine description does contain information about the machine, but its utility is compromised in one of two ways:

– The description is useful only in support of a particular algorithm, such as instruction-set emulation, LR parsing, or bottom-up tree matching.
– Information about the machine is inextricably intertwined with information about a particular tool's internal representation, such as a compiler's intermediate code.

The result is that a machine description used to build one tool – a compiler, assembler, linker, debugger, disassembler, emulator, simulator, binary translator, executable editor, verification-condition generator, or what have you – is typically useless for any other purpose. Another difficulty is that to write a machine description, you have to be a double expert: for example, to write the machine description used to retarget a compiler, you must know not only about the target machine but also about the internals of the compiler.

My colleagues, my students, and I have been exploring an alternative: the *declarative machine description.*

– It tries to favor no algorithm over any other.
– It is independent of any tool's internal representation, and indeed, independent of any tool's implementation language.
– It describes only properties of the machine, preferably in a way that is designed for *analysis*, not for execution.

We are focusing on properties that are used in the construction of systems software. We have three long-term goals:

– Declarative machine descriptions should be *reusable.* That is, from just a few descriptions of a machine, we want to build *all* of the software needed to support that machine.

- Declarative machine descriptions should *decouple machine knowledge from tool knowledge.* That is, if you know all about a machine, you should be able to retarget a tool by writing a description of the machine, even if you know nothing about the tool.
- Declarative machine descriptions should *meet the hardware halfway.* That is, our descriptions should be provably consistent with the formal descriptions used to synthesize hardware.

We can realize these benefits only if we can solve a hard problem: instead of relying on a human programmer to apply machine knowledge to the construction of a particular tool, we must somehow build tool knowledge into a program generator that can read a machine description and generate the tool[1]. For example, in our machine-description language SLED, we specify encoding and decoding of machine instructions declaratively, by sets of equations. We then use an equation solver to generate encoders and decoders (assemblers and disassemblers) by applying two kinds of tool knowledge: knowledge about relocatable object code and knowledge about decoding algorithms.

All of which brings us to the title of this talk. What if, instead of writing a code generator in a domain-specific language and calling the result a machine description, we start with a true declarative machine description and build the code generator from first principles? What *are* the first principles? What kinds of tool knowledge are neded to generate a code generator? Why is the problem hard?

We start with a simple, declarative machine description that answers two questions:

- What is the mutable state of the machine?
- When an instruction is executed, how does that state change?

Given answers to these questions, building a simulator is straightforward. But to build a compiler, we must be able to take a source program, understand its semantics in terms of state change, then find a sequence of machine instructions implementing that state change. This problem lies at the hard of building not only a compiler but also many other tools: we must somehow generalize and invert the information in the machine description.

The inversion problem has lain fallow for years. The key insight we bring is that a code generator based on inversion need not produce *good* code – it is enough to produce *correct* code. We know this because of the work of Jack Davidson and his colleagues, who developed the following compilation strategy:

- Generate very naïve code
- Improve the code *under the invariant* that every node in the flow graph can be represented by a single instruction on the target machine.

[1] Program generators often dodge this problem by allowing a machine description to "escape" to hand-written code. But hand-written code used to build one tool is likely to be useless in building another, and especially if it contains library calls, hand-written code can be nearly impossible to analyze.

This simple strategy leads to very good machine code, and it has been applied successfully in the *po*, *vpo*, and *gcc* compilers.

Using Davidson's compilation strategy, we need to read a machine description and generate four components:

– A register allocator, to map temporaries to machine registers
– A "code expander," to select machine instructions
– A "recognizer," to maintain the single-instruction invariant
– An emitter, to emit assembly language for each instruction

The talk will describe these components and how we can hope to generate them from declarative machine descriptions. Our work is still very much in progress, but we have two reasons for optimism:

– We don't need descriptions of very many properties.
– We get a lot of mileage from one idea: *binding time.*

We also hope to be able to take machine-specific human knowledge and capture it as universal truths of mathematics, which will then enable us to apply that knowledge to new machines.

References

Manuel E. Benitez and Jack W. Davidson. 1988 (July). A portable global optimizer and linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 23(7):329–338.

Jack W. Davidson and Christopher W. Fraser. 1984 (October). Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526.

Lee D. Feigenbaum. 2001 (April). Automated translation: Generating a code generator. Technical Report TR-12-01, Harvard University, Computer Science Technical Reports.

Mary F. Fernández and Norman Ramsey. 1997 (May). Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering*, pages 326–336.

Norman Ramsey. 1996 (May)a. Relocating machine instructions by currying. *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):226–236.

Norman Ramsey. 1996 (April)b. A simple solver for linear equations containing nonlinear operators. *Software – Practice & Experience*, 26(4):467–487.

Norman Ramsey. 2003 (May). Pragmatic aspects of reusable program generators. *Journal of Functional Programming*, 13(3):601–646. A preliminary version of this paper appeared in *Semantics, Application, and Implementation of Program Generation*, LNCS 1924, pages 149–171.

Norman Ramsey and Cristina Cifuentes. 2003 (March). A transformational approach to binary translation of delayed branches. *ACM Transactions on Programming Languages and Systems*, 25(2):210–224.

Norman Ramsey and Jack W. Davidson. 1998 (June). Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 172–188. Springer Verlag.

Norman Ramsey, Jack W. Davidson, and Mary F. Fernández. 2001. Design principles for machine-description languages. Unpublished draft available from http://www.eecs.harvard.edu/ nr/pubs/desprin-abstract.html.

Norman Ramsey and Mary F. Fernández. 1995 (January). The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

Norman Ramsey and Mary F. Fernández. 1997 (May). Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524.

Kevin Redwine and Norman Ramsey. 2004 (April). Widening integer arithmetic. In *13th International Conference on Compiler Construction (CC 2004)*, volume 2985 of *LNCS*, pages 232–249.

Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004 (June). A generalized algorithm for graph-coloring register allocation. *ACM SIGPLAN '04 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 39(6): 277–288.

Code Compression
(Abstract)

Saumya Debray

Department of Computer Science
University of Arizona
Tucson, AZ 85721
debray@cs.arizona.edu

Increasingly, we see a trend where programmeable processors are incorporated into a wide variety of everyday devices, ranging from “smart badges,” copy and fax machines, phones, and automobiles to traffic lights and wireless sensor networks. At the same time, the functionality expected of the software deployed on such processors becomes increasingly complex (e.g., general-purpose operating systems such as Linux on cell phones, intrusion-detection and related security security measures on wireless sensor devices). The increasing complexity of such software, and the reliability expected of them, suggest a plausible application of declarative languages. However, programs in declarative languages very often experience a significant increase in code size when they are compiled down to native code. This can be a problem in situations where the amount of memory available is limited. This talk discusses a number of different techniques for reducing the memory footprint of executables.

We begin with a discussion of classical compiler optimizations that can be used to reduce the size of the generated code. While such optimizations have traditionally focused on improving execution speed, they can be adapted quite easily to use code size as the optimization criterion instead. Especially effective are optimizations such as dead and unreachable code elimination, as well as targeted function inlining (e.g., where the callee has exactly one call site, or where inlining a function results in the elimination of so many instructions that the resulting code is smaller than the original). These optimizations can be made even more effective via aggressive interprocedural constant propagation and alias analysis, since this can propagate information from the call sites of a function into its body, potentially allowing conditionals in the body to be evaluated statically, thus making it possible to identify more of the code as unreachable.

Further code size reduction is possible using various techniques for *code factoring*, which aims to reduce code size by getting rid of repeated code fragments. This is, in essence, simply an application of procedural abstraction: repeated occurrences of a code sequence at various locations in a program are replaced by a single instance of that code that is instead called from those locations. For this to be effective, it is necessary to be able to handle code sequences that are similar but may not be identical. We currently sue a low-level approach to dealing with this, via register renaming at the basic block level. An alternative would be to use some sort of partial tree matching on a higher level program representation such as syntax trees.

Classical optimizations, coupled with code factoring, gives code size reductions of around 30% on average. The main reason this value is not higher is the constraint that the code be maintained in executable form. We can relax this constraint by keeping code in a non-executable compressed form, and decompressing it on the fly into a runtime buffer when needed. The main drawback here is the runtime cost of decompression, which can be quite substantial. Fortunately, most programs follow the so-called "80-20 rule," which states in essence that most of a program's time is spent executing a small portion of its code; a corollary is that most of a program's code is executed only infrequently, if at all. Judicious use of profile information to guide the selection of which code is decompressed at runtime yields additional code size reductions of about 15% on average, with runtime overheads of around 4%.

An orthogonal direction to code size reduction involves dynamic code mutation. The idea here is to identify a set of "similar" code fragments and keep just one representative copy of their code. At runtime, we simply edit the text section of the executable to change the code of the representative appropriately to construct the code fragment that is needed. The runtime mutations are carried out by a "code editor" that is driven by an edit script that describes the edits necessary to change one code fragment into another. This is conceptually similar to classical sequence alignment, except that in our case the edits are carried out *in situ*, which makes insertion operations very expensive. We use clustering algorithms driven by a notion of "distance" between code fragments that aims to estimate the cost of editing one sequence to construct another. Initial experiments suggest that such an approach may be useful for constructs such as C++ templates.

Functional Framework for Sound Synthesis

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen, France
karczma@info.unicaen.fr

Abstract. We present an application of functional programming in the domain of sound generation and processing. We use the lazy language Clean to define purely functional stream generators, filters and other processors, such as reverberators. Audio signals are represented (before the final output to arrays processed by the system primitives) as co-recursive lazy streams, and the processing algorithms have a strong dataflow taste. This formalism seems particularly appropriate to implement the 'waveguide', or 'physically-oriented' sound models. Lazy programming allocates the dynamical memory quite heavily, so we do not propose a real-time, industrial strength package, but rather a pedagogical library, offering natural, easy to understand coding tools. We believe that, thanks to their simplicity and clearness, such functional tools can be also taught to students interested in audio processing, but with a limited competence in programming.

Keywords: Lazy streams, Sounds, DSP, Clean.

1 Introduction

The amplitude of a sound (for one channel) may be thought of as a real function f of time t, and it is fascinating how much structural information it may contain [1]. In order to produce some audible output, this function must be sampled, and transformed into a *signal*, and this is the basic data type we shall work on. Sound may be represented at many different levels, and if one is interested in the structure of sequences of musical events, chords, phrases, etc., there is no need to get down to the digital signal processing primitives. It may seem more interesting and fruitful to speak about the algebra of musical events, music combinators, etc. This was the idea of Haskore [2], whose authors used Haskell to define and to construct a whole spectrum of musical "implementable abstractions". Haskore deals with high-level musical structures, and consigns the low-level, such as the interpretation of the MIDI streams, or the spectral structure of sounds to some back-end applications, MIDI players or CSound [3].

We decided to use the functional approach for the specification and the coding of this "low end" sound generation process. This is usually considered a highly numerical domain involving filter and wave-guide design [4], Fourier analysis, some phenomenological "magic" of the Frequency Modulation approach [5], or some models based on simplified physics, such as the Karplus-Strong algorithm [6] for the plucked string, and its extensions. But the generation and transformation of sounds is a *constructive* domain, dealing with complex abstractions (such as timbre, reverberation, etc.), and it may be based on a specific algebra as well. A possible application of functional programming paradigms as representation and implementation tools seems quite natural.

Software packages for music and acoustics are *very* abundant, some of them are based on Lisp, such as CLM [7] or Nyquist [8], and their authors encourage the functional style of programming, although the functional nature of Lisp doesn't seem very important therein as the implementation paradigm. But the the interest of the usage of functional tools in the DSP/audio domain is definitely growing, see the article [9], and the HaskellDSP package of Matt Donadio [10].

We don't offer yet a fully integrated application, but rather a pedagogical library called *Clarion*[1], in the spirit of, e.g., the C++ package STK de Perry Cook [11], but – for the moment – much less ambitious, demonstrating mainly some models based on the waveguide approach [4]. It has been implemented in a pure, lazy functional language Clean [12], very similar to Haskell. Its construction focuses on:

- Co-recursive (lazy) coding of "data flow" algorithms with delay lines, filters, etc. presented declaratively. This is the way we build the sound samples.
- Usage of higher-order combinators. We can define "abstract" instruments and other sound processors at a higher level than what we see e.g., in Csound.

The dataflow approach to music synthesis, the construction of virtual synthesizers as collections of pluggable modules is of course known, such systems as Buzz, PD or Max/JMax [13–15] are quite popular. There is also a new sequential/time-parallel language Chuck [16] with operators permitting to transfer the sound data streams between modules. All this has been used as source of inspiration and of algorithms, sometimes easier to find in library sources than in publications...

Our implementation does not neglect the speed issues, but because of dynamic memory consumption the package is not suitable for heavy duty event processing. It is, however, an autonomous package, able to generate audible output on the Windows platform, to store the samples as `.wav` files, and to read them back. It permits to code simple music using a syntax similar to that of Haskore, and to depict graphically signals, envelopes, impulse responses, or Fourier transforms. The choice of Clean has been dictated by its good interfacing to lower-level Windows APIs, without the necessity of importing external libraries. The manipulation of 'unique' unboxed byte arrays which ultimately store the sound samples generated and processed as streams, seems sufficiently efficient for our purposes. The main algorithms are platform-independent, and in principle they could be coded in Haskell. The main purpose of this work is pedagogical, generators and transformers coded functionally are compact and readable, and their cooperation is natural and transparent.

1.1 Structure of This Article

This text can be assimilated by readers knowing typical functional languages such as Clean or Haskell, but possessing also some knowledge of the digital signal processing. While the mastery of filters, and the intricacies of such effects as the reverberation are not necessary in order to grasp the essential, some acquaintance with the signal processing techniques might be helpful.

[1] We acknowledge the existence of other products with this name.

We begin with some typical co-recursive constructions of infinite streams, and we show how a dataflow diagram representing an iterative signal processing is coded as a co-recursive, lazy stream. We show the construction of some typical filters, and we present some instruments, the plucked string implemented through the Karplus-Strong model, and simplified bowed string and wind instruments. We show how to distort the "flow of time" of a signal, and we show a simple-minded reverberator.

2 Co-recursive Stream Generators

A *lazy* list may represent a *process* which is activated when the user demands the next item, and then suspends again. This allows to construct infinite *co-recursive* sequences, such as $[0, 1, 2, 3, \ldots]$, coded as `ints = intsFrom 0 where intsFrom n=[n:intsfrom (n+1)]`, and predefined as `[0 .. ]`. Such constructions are useful e.g., for processing of infinite power series [18], or other iterative entities omnipresent in numerical calculi. The "run-away" co-recursion is not restricted to functions, we may construct also co-recursive ***data***. The sequence above can be obtained also as

```
ints = [0 : ints+ones] where ones=[1 : ones]
```

provided we overload the `(+)` operator so that it add lists element-wise. Here the sequences y_n for $n = 0, 1, 2, \ldots$ will represent samples of a discretized acoustic signal, and for the ease of processing they will be real, although during the last stage they are transformed into arrays of 8- or 16-bit integers. Already with `[0 .. ]` we can construct several sounds, e.g., a not very fascinating sinusoidal wave:

```
wave = map (\n -> sin(2*Pi*n*h)) [0 .. ]
```

where `h` is the sampling period, the frequency divided by the sampling rate: the number of samples per second. In order to cover the audible spectrum this sampling rate should be at least 32000, typically it is equal to 44100 or higher.

In the next section we shall show how to generate such a monochromatic wave by a recursive formula without iteration of trigonometric functions, but the essential point is not the actual recipe. What is important is the fact the we have a *piece of data*, that an infinitely long signal is treated declaratively, outside any loop or other syntactic structure localized within the program. For efficiency we shall use a Clean specific type: head-strict unboxed lists of reals, denoted by such syntax: `[# 1.0,0.8, ...]`.

3 Simple Periodic Oscillator

One of the best known algorithmic generators of sine wave is based on the recurrent formula: $\sin(n\omega h) = 2\cos(\omega h)\sin((n-1)\omega h) - \sin((n-2)\omega h)$, where h is some sampling interval, and $\omega = 2\pi f$ with f being the frequency. One writes the difference equation obeyed by the discrete sequence: $y_n = c \cdot y_{n-1} - y_{n-2}$ where $c = 2\cos(\omega h)$ is a constant. One may present this as a dataflow "circuit" shown on Fig. (1). Here z^{-1} denotes conventionally the unit delay box.

Of course, the circuit or the sequence should be appropriately initialized with some non-zero values for y_0 and y_1, otherwise the generator would remain silent. A Clean infinite list generator which implements the sequence $\sin(2\pi f t)$ with $t = n/$Sampling Rate may be given in a typically functional, declarative style as

```
oscil fr = y where   //  DpiSR is 2π/SR
  omh = DpiSR*fr     //  (divisor: global SamplingRate)
  y = [# 0.0 : v]
  v = [# sin omh : ((2.0*cos omh)*>v - y)]
```

The values $0.0 = \sin(0.0)$ and $\sin(\omega h)$ can be replaced by others if we wish to start with another initial phase. Note the cycle in the definition of variables, without laziness it wouldn't work! The operator `(*>)` multiplies a scalar by a stream, using the standard

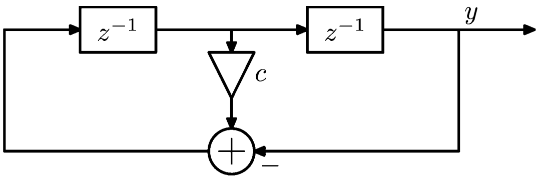


Fig. 1. Sine wave generator

`Map` functional (overloading of `map` for `[#]` lists): `(*>) x l = Map (\s->x*s) l`; we have also defined a similar operator `(+>)`, which raises the values in a stream by a constant, etc. The subtraction and other binary arithmetic operators have been overloaded element-wise for unboxed real streams, using the known functional `zipWith`. Some known list functionals such as `map`, `iterate` or `take` in this domain are called `Map`, `Iterate`, etc. This algorithm is numerically stable, but in general, in presence of feedback one has to be careful, and *know* the behaviour of iterative algorithms, unless one wants to exploit the imprevisible, chaotic generators.... Another stable co-recursive algorithm may be based on a *modified* Euler method of solving the oscillator differential equation: $\ddot{y} = -\omega^2 y$ (through $\dot{y} = \omega v$; $\dot{v} = -\omega y$). We get y equal to the sampled sine, with `y=[#0.0:y]+a*>v`; `v=[#1.0:v-a*>y]`. A pure sinusoidal wave is not a particularly fascinating example, but it shows already some features of our approach.

– As already mentioned, the sound signal is not a 'piece of algorithm' such as a loop over an array, but a first-class data item, although its definition contains a never-ending, recurrent code. We can add them in the process of additive synthesis, multiply by some scalars (amplify), replicate, etc. The modularity of the program is enhanced with respect to the imperative approach with loops and re-assignments. Some more complicated algorithms, such as the pitch shift are also easier to define.
– We have a *natural* way of defining data which rely on feedback. The coding of IIR filters, etc., becomes easy. In general, lazy streams seem to be a good, *readable* choice for implementing many 'waveguide' models for sound processing [4].

With the stream representation it is relatively easy to modulate the signal by an envelope, represented as another stream, typically of finite length, and normalized so that its maximum is equal to 1, it suffices to multiply them element-wise. An exponential decay (infinite) envelope can be generated on the spot as `expatt r = Iterate ((*) r) 1.0`. The *parameterization* of an arbitrary generator by an arbitrary envelope is more involved, and depends on the generator. But, often when in definitions written in this style we see `c*>s` where `c` is a constant, we may substitute `m*s` for it, where `m` is the modulating stream. Thus, in order to introduce a 5 Hz *vibrato* (see fig. 9) to our oscillator, it suffices to multiply `v` in `{2*cos omh*>v}` by, say, `{(1.0+>0.001*>oscil`

`5.0))}`, since the concerned gain factor determines the pitch, rather than the amplitude. This is not a universal way to produce a varying frequence, we just wanted to show the modularity of the framework. Some ways of generating *vibrato* are shown later.

It is worth noting that treating signals as data manipulated arithmetically permits to code in a very compact way the instruments defined through the additive synthesis, where the sound is a sum of many partials (amplitudes corresponding to multiples of the fundamental frequency), for example:

```
additive freq amplist = y
 where
  m = [1.0 .. toReal(length amplist)]
  y = Foldl (+) zeros          // an infinite stream of zeros
      (map (\(n,a)->a*>oscil(n*freq)) (zip2 m amplist))

oboe fr= additive fr [0.0021,0.0237,0.1,0.0513,0.045,0.061,0.0168]
tuba fr= additive fr [0.10095,0.15732,0.14992,0.09895,0.07178,...]
```

Envelopes, *vibrato*, etc. effects can be added during the post-processing phase.

4 Some Typical Filters and Other Simple Signal Processors

A filter is a *stream processor*, a function from signals to signals, $x \to y$. The most simple smoothing, low-pass filter is the averaging one: $y_n = 1/2(x_n + x_{n-1})$. This can be implemented directly, in an index-less, declarative style, as `y = 0.5*>(x + Dl x)`, where `Dl x = [#0.0 : x]`. If we are not interested in the first element, this is equivalent to `y = 0.5*>(x + Tail x)`, but this variant is less lazy. This is an example of a FIR (Finite Impulse Response), non-recursive filter. We shall also use the "Infinite Response", recursive, feedback filters (IIR), where y_n depends on y_{n-k} with $k > 0$, and whose stream implementation is particularly compact.

One of the standard high-pass filters, useful in DSP, the "DC killer", 1-zero, 1-pole filter, which removes the amplitude bias introduced e.g., by integration, is described by the difference equation $y_n = b \cdot (x_n - x_{n-1}) + a \cdot y_{n-1}$, and constitutes thus a (slightly renormalized) differentiator, followed by a "leaky" integrator, everything depicted in Fig. 2. Here a is close to 1, say 0.99, and $b = (1 + a)/2$.

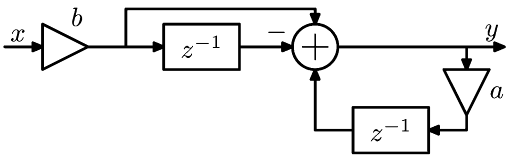


Fig. 2. DC blocker circuit

The effect of the filter on, say, a sample of brown (integrated white) noise is depicted on Fig. (3), with $b = 0.97$, and the equivalent Clean program is shown below, again, it is a co-recursive 1-liner, where a stream is defined through itself, delayed. This delay is essential in order to make the algorithm effective; our code, as usually in DSP, should not contain any "short loops", where a recursive definition of a stream cannot resolve the value of its head.

```
dcremove a (xs=:[# x0:xq]) = y where
  y=a*>Dl y+((1.0+a)/2.0)*>(xq-xs)
```

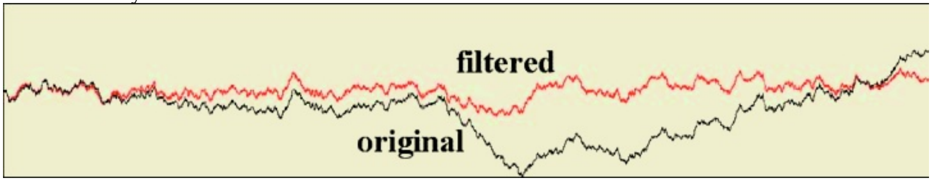


Fig. 3. DC removal from the "brown noise"

The damping sensitivity, and losses are bigger when a is smaller. Note the difference between the diagram and the code: The delay and the multiplication by b have been *commuted*, which is often possible for time-independent, linear modules, and here economises one operation. The parameter pattern syntax `a=:`α is equivalent to Haskell `a@`α, and permits to name and to destructure a parameter at the same time.

Another feedback transducer of general utility is an 'all-pass filter', used in reverberation modelling, and always where we need some dispersion – difference of speed between various frequency modes. The variant presented here is a combination of 'comb' forward and feedback filters, and it is useful e.g. for the transformation of streams generated by strings (piano simulators, etc.). The definition of a simple all-pass filter may be

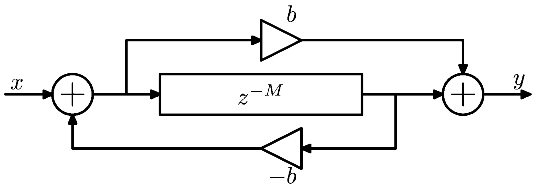


Fig. 4. An all-pass filter

```
allpass m b x = b*>z + v where
  v = delay m z   // Prepend m zeros to z
  z = x - b*>v
```

5 The Karplus-Strong Model of a Plucked String

Finally, here is another sound generator, based on a simplified 'waveguide' physical model [4]. A simplistic plucked string (guitar- or harpsichord-like sound) is constructed as a low-pass-filtered delay line with feedback. Initially the line is filled-up with any values, the (approximate) "white noise": a sequence of uncorrelated random values between ± 1, is a possible choice, standard in the original Karplus-Strong model.

On output the neighbouring values are averaged, which smooths down the signal, and the result is pumped back into the delay line. After some periods, only audible frequencies are the ones determined by the length of the delay line, plus some harmonics. Higher frequencies decay faster than the lower ones, which is the typical feature of the plucked string, so this incredibly simple model gives a quite realistic sound! The implementation is straightforward, the only new element is the usage of a finite-length segment and the overloaded concatenation operator `(++|)`.

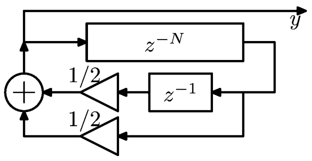


Fig. 5. Plucked string

```
karstr f = y where
  prfx = Take (toInt (SR/f)) whitenoise
  y = prfx ++| 0.5 *> (y + Tl y)  // or more delayed: y+Dl y
```

The noise signal is a stream generated by an iterated random number generator. The package disposes of several noise variants, brown, pink, etc., useful for the implementation of several instruments and effects.

A more complicated string (but equally easy to code from a given diagram), with external excitation, some slightly detuned parallel delay lines, additional all-pass filters etc., is used to simulate piano, harpsichord or mandolin, but also bowed instruments (violin), and some wind instruments, since mathematically, strings and air columns are similar. We shall omit the presentation of really complex instruments, since their principle remains the same, only the parameterization, filtering, etc. is more complex.

Since a looped delay line will be the basic ingredient of many instruments, it may be useful to know where the model comes from. If $y(x,t)$ denotes the displacement of air or metal, etc. as a function of position (1 dimension) and time, and if this displacement fulfils the wave equation without losses nor dispersion (stiffness): $\partial^2 y/\partial t^2 = c^2 \cdot \partial^2 y/\partial x^2$, its general solution is a superposition of two *any* travelling waves: $y(x,t) = y_r(x-ct) + y_l(x+ct)$, where c is the wave speed. For a vibrating string of length L, we have $c = 2f_0L$, where f_0 is the string fundamental frequency.

After the discretisation, the circuits contain two "waveguides" – delay lines corresponding to the two travelling waves. They are connected at both ends by filters, responsible for the wave reflection at both ends of the string (or the air column in the bore of a wind instrument). Because of the linearity of the delay lines and of typical filters, it is often possible to 'commute' some elements, e.g., to transpose a filter and a delay line, and to coalesce two delay lines into one. One should not forget that an open end of a wind instrument should be represented by a phase-inversion filter, since the outgoing low-pressure zone sucks air into the bore, producing a high-pressure incoming wave.

6 Digression: How to Make Noise

A typical random number generator is a 'stateful' entity, apparently not natural to code functionally, since it propagates the "seed" from one generation instance to another. But, if we want just to generate random *streams*, the algorithm is easy. We start with a simple congruential generator based on the fact that Clean operates upon standard 'machine' integers, and ignores overflows, which facilitates the operations modulo 2^{32}. It returns a random value between ± 1, and a new seed. For example:

```
rand1 seed
 # seed = 599479 + seed*25781083
 # r  = seed bitand 2147483647
 = (toReal r/2147483648.0,seed)
```

Despite its appearance, this is a purely functional construction, the `#` syntax is just a sequential variant of **let**, where the "reassignment" of a variable (here: `seed`) is just an introduction of a new variable, which for mnemotechnical purposes keeps the same name, screening the access to its previous, already useless instance. We write now

```
rndnoise seed
   # (z,seed) = rand1 seed
   = [#z : rndnoise seed]
```

This noise can be used in all typical manipulations, for example the “brown noise” which is the integral of the above “white noise” is just

```
brownnoise = z where   z=Dl (rndnoise someSeed + z)
```

In the generation of sounds the noise need not be perfect, the same samples can be reused, and if we need many streams it suffices to invoke several instances of `rndnoise` with different seeds. It is worth noticing that in order to generate noise in a purely functional way, *we don’t need an iterative generator with propagating seed*!. There exist *pure* functions which behave *ergodically*, the values corresponding to neighbouring arguments are uncorrelated, they vary wildly, and finally they become statistically independent of their arguments. A static, non-recursive pure function

```
ergodic n
  # n = (n<<13) bitxor n   // Use let n'= ... if you don't like #
  = toReal (n*(n*n*599479+649657)+1376312589)/2147483648.0
```

mapped through the list $[0, 1, 2, 3, 4, \ldots]$ gives the result shown in Fig. 6. We have also used this variant of noise.

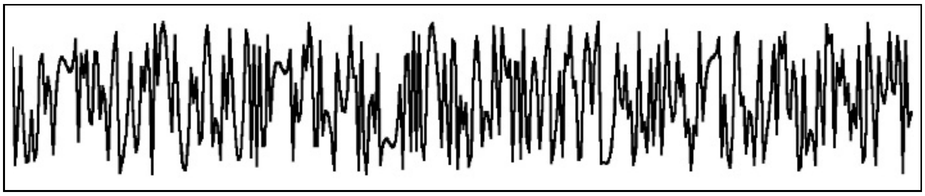


Fig. 6. An ergodic function

7 More General Filters, and Power Series

We have seen that a simple recurrence, say $y_n = b_0 \cdot x_n - a_1 \cdot y_{n-1}$ is easy to represent as a stream (here: `y = b0*>x - a1*>Dl y`), but a general m-zero, l-pole filter: $y_n = \sum_{k=0}^{m} b_k \cdot x_{n-k} - \sum_{k=1}^{l} a_k \cdot y_{n-k}$ would require a clumsily looking loop.

However, both terms are just convolutions. The DSP theory [17] tells us that they become simple products when we pass to the z transform of our sequences. Concretely, if we define $x(z) = \sum_{n=0}^{\infty} x_n z^{-n}$, and introduce appropriate power series for the sequences b, $a = [a_0 = 1, a_1, a_2, \ldots]$ and y, we obtain the equation $a(z) \cdot y(z) = b(z) \cdot x(z)$, or $y(z) = H(z) \cdot x(z)$, where $H = b/a$. The stream-based arithmetic operations on formal power series are particularly compact [18]. Element-wise adding and subtracting is trivial, uses `Map (+)` and `Map (-)`. Denoting $x = x_0 + z^{-1}\bar{x}$, where $\bar{x}$ is the tail of the sequence, etc., it is easy to show that

$$w \equiv (w_0 + z^{-1}\bar{w}) = b \cdot x = (b_0 + z^{-1}\bar{b})(x_0 + z^{-1}\bar{x}) \,, \tag{1}$$

reduces to the algorithm: $w_0 = b_0 x_0$, and $\bar{w} = b_0\bar{x} + \bar{b}x$.

The division $w = b/a$ is the solution for $[w_0 : \bar{w}]$ of the equation $b = a \cdot w$, and is given by the recurrence $w_0 = b_0/a_0$ and $\bar{w} = (\bar{b} - w_0\bar{a})/a$. So, the code for a general filtering transform is `y = (b<*>x)</>a` with

```
(<*>) [#b0:bq] a=:[#a0:aq] = [# b0*a0 : b0*>aq + bq<*>a]
(</>) [#b0:bq] a=:[#a0:aq] = [# w0 : (bq - w0*>aq)</>a]
      where w0 = b0/a0
```

It was not our ambition to include in the package the filter design utilities (see [10]), but since the numerator and the denominator of H are lists equipped with the appropriate arithmetic, it is easy to reconstruct their coefficients from the positions of the zeros, by expansions. But, since the division of series involves the feedback, the programmer must control the stability of the development, must know that all poles should lie within the unit disk in order to prevent the explosive growth, and be aware that the finite precision of arithmetics contributes to the noise. But the numerical properties of our algorithms are independent of the functional coding, so we skip these issues.

8 More Complex Examples

8.1 Flute

One of numerous examples of instruments in the library STK of Perry Cook [11] is a simple flute, which must contain some amount of noise, the noise gain g on the diagram depicted on fig. 7 is of about 0.04. The flute is driven by a "flow", the breath strength, which includes the envelope of a played note. There are two delays, the bore delay line, which determines the period of the resonance frequency, and a two times shorter embouchure delay block. The filter H is a reflection low-pass filter given by $y_n = 0.7x_n + 0.3y_{n-1}$. The gain coefficients are$c_1 = 0.5$, and $c_2 = 0.55$. The code is short.

```
flute freq flow = w where
 lpass1 x = y where y=0.7*>x + 0.3*>Dl y
 nlins xs = Map (\x -> x*(1.0 - x*x))xs
 u=0.04*>(flow*whitenoise) + flow
 v=u+0.5*>p
 p=tdelay (1.0/freq) w   // delay parameterized by real time
 w=lpass1 (0.55*>p + nlins (tdelay (0.5/freq) v))
```

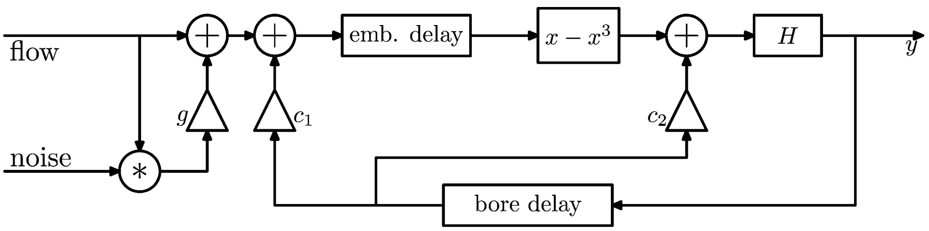


Fig. 7. A simple flute

An important ingredient of the instrument is the nonlinearity introduced by a cubic polynomial. Välimäki et al. [19] used a more complicated model. They used also an interpolated, fractional delay line, which is an interesting subject *per se*.

8.2 Bowed String

In Fig. 8 we see another example of a non-linearly driven, continuous sound instrument, a primitive "violin", with the string divided into two sections separated by the bow, whose movement pulls the string. Of course, after a fraction of second, the string slips, returns, and is caught back by the bow. The frequency is determined, as always, by the resonance depending on the string length.

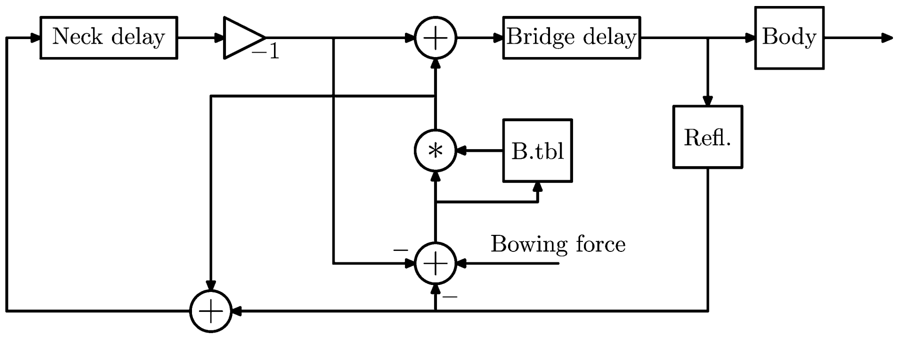


Fig. 8. A bowed string

```
bowed amp freq = y                  // amp is the bowing force
 where
  basedel = 2.0*SR/freq-4.0         // Base delay, split into neck and bridge
  p = 0.6 - 2205.0/SR               // Phenomenological filter pole position
  bowv = (0.03+0.2*amp)*>ones       // Max bow velocity; ones=[1,1,1,...]
  brefl= ~(bridge p 0.95 brdel)
  nrefl= ~(delay (toInt(0.872764*basedel)) (brefl+nvel)) // Neck
  vdiff = bowv-brefl-nvel           // Velocity difference; steering value
  nvel = vdiff*Map bowtable vdiff
  brdel= delay (toInt(0.127236*basedel)) (nrefl+nvel)
  y = biquad 500.0 0.85 0.6 brdel    // Output resonating filter
```

where `bridge` is the bridge reflection filter, and `bowtable` – the non-linearity (it could be table-driven, but here it is a local function). The function `biquad` is a filter. They are defined as in the source of STK:

```
bridge p g s = y where
  b0=(if(p>0.0) (1.0-p) (1.0+p))
  y = (g*b0)*>s + p*>Dl y
```

```
bowtable x
 # r=((abs(3.0*x)+0.75)^(-4.0))
 = if (r<1.0) r 1.0         // Conditional clipping
biquad freq rad g s = y    // Biquad resonating filter
 where
  a2=rad*rad
  v = (g*(0.5-0.5*a2))*>(s - D1 (D1 s))
  z = D1 y
  y = v + (2.0*rad*cos (freq*DpiSR))*>z - a2*>D1 z
```

Now, in order to have a minimum of realism, here, and also for many other instruments, the resonating frequency should be *precise*, which conflicts with the integer length of the delay line. We need a fractional delay. Moreover, it should be parameterized, in order to generate *vibrato* (or *glissando*); we know that the dynamic pitch changes are obtained by the continuous modification of the string length. Our stream-based framework, where the delay is obtained by a prefix inserted in a co-recursive definition seems very badly adapted to this kind of effects. We need more tools.

9 Fractional Delay and Time Stretching

Some sound effects, such as chorus or flanging [20], are based on dynamical delay lines, with controllable delay. For good tuning, and to avoid some discretisation artefacts is desirable to have delay lines capable of providing a non integer (in sampling periods) delay times. Such fractional delay uses usually some interpolation, linear or better (e.g., Lagrange of 4-th order). The simplest linear interpolation delay of a stream `u` by a fraction `x` is of course `v=(1.0-x)*>u + x*>D1 u`, where the initial element is an artefact; it is reduced if we replace the 0 of `D1` by the head of `u`. Smith proposes the usage of an interpolating all-pass filter, invented by Thiran [21], and co-recursively defined as

```
ifractd x s=:[#s0:_] = v + a*>u
 where
  a=(1.0-x)/(1.0+x)
  u = s - a*>v
  v = [#s0:u]
```

This module should be combined with a standard, integer delay. But how to change the big delay dynamically? The classical imperative solution uses a circular buffer whose length (distance between the read- and write pointers) changes during the execution of the processing loop. In our case the delay line will become a stream processor. We begin with a simple, static time distortion. The function `warp a x` where x is the input stream, and a – a real parameter greater than -1.0, shrinks or expands the discrete "time" of the input stream. If $a = 0$, then $y_n = x_n$, otherwise $y_0 = x_0$; $y_1 = x_{1+a}; \ldots\ y_n = x_{n\cdot(1+a)}$, where an element with fractional index is linearly interpolated. For positive α: $x_{n+\alpha} \equiv (1-\alpha)x_n + \alpha x_{n+1}$. Here is the code:

```
warp a [#x0 : xq] = [#x0 : wrp a x0 xq] where
  wrp g y0 ys=:[#y1:yq]|g>0.0 = wrp (g-1.0) y1 yq
                       =[#(1.0+g)*y1-g*y0 : wrp (g+a+1.0) y0 ys]
```

It is possible to vary the parameter a, the package contains a more complicated procedure, where the delay parameter a is itself a stream, consumed synchronously with x. It can itself be a periodic function of time, produced by an appropriate oscillator. In such a way we can produce the *vibrato* effect, or, if its frequency is high enough – an instrument based on the frequency modulation. Moreover, if warp is inserted into a feedback, e.g., `u=prefix ++| warp dx (filtered u)` with a rather very small `dx`, the recycling of the warped `u` produces a clear *glissando* of the prefix.

The technique exploited here is fairly universal, and plenty of other DSP algorithms can be coded in such a way. If we want that a "normal" function `f` from reals to reals, and parameterized additionally by `a`, transform a stream `x`, we write simply `Map (f a) x`. Now, suppose that after every `n` basic samples the parameter should change. This may be interesting if together with the basic audio streams we have *control* streams, which vary at a much slower rate, like in Csound. We put the sequence of parameters into a stream `as`, and we construct a generalized mapping functional

```
gmap f s n x = wm 0 s x where
  wm k as=:[#a0:aq] x=:[#x0:xq]|k<n = [#f a0 x0:wm (k+1) as xq]
                                    = wm 0 aq x
```

This may be used to modulate periodically the amplitude in order to generate the *tremolo* effect. A modified `warp` function, parameterized not by a constant but by a stream, if driven by an oscillator produces the *vibrato* effect. They are shown in Fig. 9.

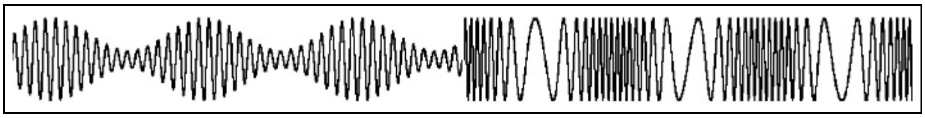


Fig. 9. Tremolo and vibrato

10 Reverberation

One of fundamental sound effects, which modifies the timbre of the sound is the composition of the original with a series of echos, which after some initial period during which the individual components are discernible, degrade into a statistical, decaying noise. A "unit", infinitely sharp sound represented by one vertical line in Fig. 10, is smeared into many. Fig. 10 corresponds to the model of John Chowning, based on the ideas of Schroeder [22], see also [23]. The reverberation circuit contains a series of all-pass filters as shown on fig. 4, with different delay parameters. This chain is linked to a parallel set of feed-forward comb filters, which are "halves" of the all-pass filters. They are responsible for the actual echo. Fig. 11 shows our reverberation circuit, whose code is given below. It is simple, what is worth noticing is a compact way of representing the splitting of a stream into components which undergo different transformations, as given by the auxiliary functional `sumfan`. The operator o denotes the composition. Thanks

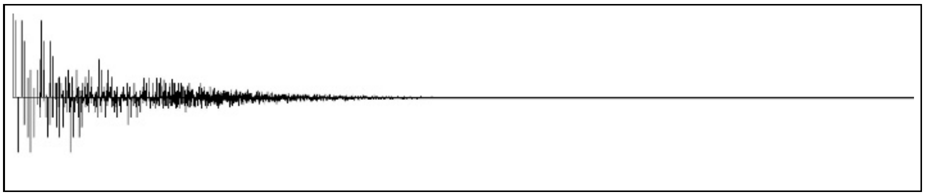


Fig. 10. Reverberation response

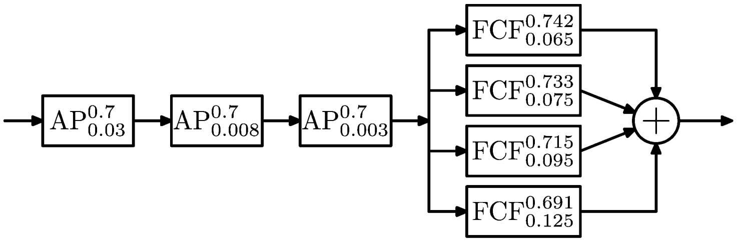


Fig. 11. Reverberation module

to the curried syntax of Clean, the definition of a processing module composed out of elements linked serially and in parallel, doesn't need the stream argument.

```
reverb =
 schr 0.03 o schr 0.008 o schr 0.003 o
   sumfan [fcf 0.065 0.742,fcf 0.075 0.733,
           fcf 0.095 0.715,fcf 0.125 0.691]
 where                                  // fcf is a forward comb filter
  fcf tm g x = x + g*>tdelay tm x       // Time, gain, stream

  schr tm = allpass tm 0.707            // Schröder allpass filter, below

  sumfan l x
   # [l1:lq]=map (\f->f x) l
   = 0.25*>foldl (+) l1 lq
  allpass tm b x = b*>z + v  where      // Co-recursive dispersion filter
    v = tdelay tm z
    z = x - b*>v
```

11 Conclusions and Perspectives

We have shown how to implement some useful algorithms for the generation and transformation of sound signals in a purely functional, declarative manner, using lazy streams which represented signals. The main advantage of the proposed framework is its compactness/simplicity and readability, making it a reasonable choice to *teach* sound processing algorithms, and to experiment with. We kept the abstraction level of the presentation rather low, avoiding excessive generality, although the package contains some other, high-level and parameterized modules. This is not (yet?) a full-fledged real-time generating library, we have produced rather short samples, and a decent user interface which would permit to pameterize and test several instruments in a way similar to STK [11] is under elaboration. The paper omits the creation of high-level objects: musical notes and phrases, not because it has not been done, or because it is not interesting, on the contrary. But this has been already treated in Haskore papers, and other rather *music*-oriented literature.

Our package can compute Fourier transforms, and contains other mathematical utilities, such as the complex number algebra, and a small power-series package. It has some graphic procedures (the signal plots presented in the paper have been generated with Clarion). We have coded some more instruments, as a clarinet, or a primitive harpsichord. Clarion can input and output `.wav` files, and, of course, it can play sounds, using the Windows specific `PlaySound` routine. We have generated streams up to 2000000 elements (about one minute of dynamically played sound at the sampling rate of 32000/sec.), and longer. In principle we might generate off-line sound files of much bigger lengths, but in order to play sound in real time it is necessary to split the audio stream in chunks. Our system uses heavily the dynamic memory allocation, and the garbage collection may become cumbersome. It is also known that the temporal efficiency of lazy programs (using the call-by-need protocol) is usually inferior to typical, call-by-value programs. We never tried to implement a full orchestra in Clarion...

Clarion remains for the moment rather a functional, but pedagogical feasibility study, than a replacement for Csound or STK, those systems are infinitely more rich. We wanted to show on a concrete example how the functional composition and lazy list processing paradigms may be used in practice, and we are satisfied. This work will continue.

Acknowledgements

I owe generous thanks to Henning Thielemann for interesting and useful comments, and for inspiring conversation.

References

1. John William Strutt (Lord Rayleigh), *The Theory of Sound*, (1877).
2. Paul Hudak, Tom Makucevich, Syam Gadde, Bo Whong, *Haskore Music Notation – an Algebra of Music*, J. Func. Prog. **6**:3, (1996), pp. 465–483. Also: Paul Hudak, *Haskore Music Tutorial*, in: *Advanced Functional Programming* (1996), pp. 38–67. The system is maintained at: `www.haskell.org/haskore/`.
3. Richard Boulanger, *The Csound Book*, (2000). See also the site `www.csounds.com`
4. Julius O. Smith, Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, Web publications at `www-ccrma.stanford.edu/~jos/`, (2003). Also: *Physical modelling using digital waveguides*, Computer Mus. Journal **16**, pp. 74–87, (1992).
5. J. Chowning, *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*, Journal of Audio Eng. Soc. **21**(7), (1973).
6. Kevin Karplus, A. Strong, *Digital Synthesis of Plucked Strings and Drum Timbres*, Computer Mus. Journal **7**:2, (1983), pp. 43–55. See also: D.A. Jaffe, J.O. Smith, *Extensions of the Karplus-Strong Plucked String Algorithm*, Comp. Mus. Journal **7**:2, (1983), pp. 56–69.

7. Bill Schottstaedt and others, *Common Lisp Music*, site `ccrma-www.stanford.edu/software/clm/`.
8. Roger B. Dannenberg, *Nyquist Reference Manual*, `www-2.cs.cmu.edu/~rbd/doc/nyquist/`.
9. Henning Thielemann, *Audio Processing Using Haskell*, Proc. 7th Int. Conf. on Digital Audio Effects (DAFx'04), Naples, pp. 201–206, (2004).
10. Matthew Donadio, HaskellDsp sources, site `haskelldsp.sourceforge.net`.
11. Perry Cook, site `www-ccrma.stanford.edu/CCRMA /Software/STK/`, see also the book: *Real Sound Synthesis for Interactive Applications*, A.K. Peters, (2002).
12. Rinus Plasmeijer, Marko van Eekelen, *Clean Language Report, version 2.1*, site `www.cs.kun.nl/~clean/`.
13. Oskari Tammelin, *Jeskola Buzz*, Modular software music studio, see e.g., `www.buzzmachines.com/`, or `www.jeskola.net/`.
14. Miller Puckette, *Pure Data: PD, Documentation*, `crca.ucsd.edu/~msp/Pd_documentation/`
15. François Déchelle, *Various IRCAM free software: jMax and OpenMusic*, Linux Audio Developers Meeting, Karlsruhe, (2003). See also `freesoftware.ircam.fr/`.
16. Ge Wang, Perry Cook, *ChucK: a Concurrent, On-the-fly, Audio Programming Language*, Intern. Comp. Music Conf., Singapore (2003). See also: `chuck.cs.princeton.edu/`.
17. Ken Steiglitz, *A DSP Primer: With Applications to Digital Audio and Computer Music*, Addison-Wesley, (1996).
18. Jerzy Karczmarczuk, *Generating power of Lazy Semantics*, Theor. Comp. Science **187**, (1997), pp. 203–219.
19. R. Hänninen, V. Välimäki, *An improved digital waveguide model of a flute with fractional delay filters*, Proc. Nordic Acoustical Meeting, Helsinki, (1996), pp. 437–444.
20. Sophocles Orphanidis, *Introduction to Signal Processing*, Prentice-Hall, (1995).
21. J.P. Thiran, Recursive digital filters with maximally flat group delay, IEEE Trans. Circuit Theiry 18 (6), Nov. 1971, pp. 659–664.
22. M.R. Schroeder, B.F. Logan, *Colorless artificial reverberation*, IRE Transactions, vol. AU-9, pp. 209–214, (1961).
23. J.O. Smith, *A new approach to digital reverberation using closed waveguide networks*, Proceedings, ICMC (1985).

Specializing Narrowing for Timetable Generation: A Case Study

Nadia Brauner, Rachid Echahed, Gerd Finke,
Hanns Gregor, and Frederic Prost

Institut d'Informatique et de Mathématiques Appliquées de Grenoble,
Laboratoire Leibniz, 46, av Félix Viallet, 38000 Grenoble, France
{Nadia.Brauner,Rachid.Echahed,Gerd.Finke,Hanns.Gregor,Frederic.Prost}
@imag.fr

Abstract. An important property of strategies used to solve goals in functional logic programming (FLP) languages is the complete exploration of the solution space. Integrating constraints into FLP proved to be useful in many cases, as the resulting constraint functional logic programming (CFLP) offers more facilities and more efficient operational semantics. CFLP can be achieved by means of conditional rewrite systems with a narrowing-based operational semantics. A common idea to improve the efficiency of such operational semantics is to use specific algorithms from operations research as constraint solvers. If the algorithm does not return a complete set of solutions, the property of completeness might be lost. We present a real world timetabling problem illustrating this approach. We propose an algorithm, obtained as an integration of three known optimization algorithms for the linear assignment problem (LAP), enumerating solutions to the LAP in order of increasing weight, such that resolution of goals is complete again. We show, how the narrowing process can be tailored to use this algorithm and provide an efficient way to solve the timetable generation problem.

Keywords: Functional-logic programming, Constraints, Narrowing, Timetable generation.

1 Introduction

In software development, it is often practical to use different programming paradigms for modeling different parts of the problem to solve. A system integrating various programming paradigms allows the programmer to express each part of the problem with the best suitable concepts. The formalization of the whole program also becomes easier and the resulting software design is often considerably less complex.

By using a conditional rewrite system with narrowing as operational semantics, it is possible to combine constraint logic programming [14] and functional logic programming [12] in a single constraint functional logic programming framework (e.g., [19]). A program in such a framework is composed of conditional rewrite rules of the form

$$L \to R \mid Q_i(s_1, \ldots, s_{m_i}), i \in \{0..n\}$$

where $Q_i(s_1, \ldots, s_{m_i})$ denotes a list (conjunction), possibly empty, of conditions. This rule can be applied as a classical rewrite rule only if the conditions Q_i hold. Typically, Q is an equation or an m-ary constraint (predicate). With appropriate narrowing strategies, it is possible to obtain a complete operational semantics for the interpretation of such conditional rewrite rules. Nevertheless, for some specific constraints the evaluation time of narrowing, which is essentially an exhaustive search among the possibilities, can be prohibitive. In order to cope with this problem under those specific conditions, it is possible to use operations research algorithms, which are much more efficient. In order to maintain the completeness of the calculus when specialized constraint solvers are integrated, each of these solvers needs to return a complete set of solutions. In this paper we focus on a particular case involving a solver for the linear assignment problem.

The linear assignment problem is efficiently solved by the Hungarian method [16]. This algorithm returns exactly one optimal solution to the stated problem. It thereby leads to an incomplete operational semantics. A first aim of this paper is to propose an algorithm for the linear assignment problem that enumerates all solutions to the assignment problem in order of decreasing quality (increasing cost). This is done by the combination of three known algorithms. The completeness of classical operational semantics of logic or functional logic programming languages with constraints is then recovered. A second aim is to illustrate the proposed algorithm by a case study: the timetable generation for medical staff.

We assume the reader familiar with narrowing-based languages (see e.g. [3]). Thus we rather start, in section 2, by introducing the three algorithms for the assignment problem which we propose to combine. Then we show in section 3 how a combination of these algorithms can be done in order to enumerate all possible (but non-optimal) assignments. Section 4 presents a practical case study, namely timetable generation of medical staff in a Parisian hospital. We illustrate how declarative programming can be efficiently used in such a case thanks to this new complete algorithm for LAP. Finally, we conclude and discuss further works in section 5. Due to lack of space, some presentations have been shortened. More details could be found in [4].

2 Linear Assignment Problem: Definition and Solutions

In this section we precise the linear assignment problem and give a brief overview of three algorithms from literature about this problem. The first one, the Hungarian method, gives exactly one answer, the second one enumerates all optimal assignments, while the final one considers suboptimal assignments. Detailed descriptions of the presented algorithms may be found in [4] or in the cited literature.

2.1 Definition

A graph $G = (V, E)$ with vertex set V and edge set E is said to be *bipartite*, if V can be partitioned into two sets S and T such that in each of these sets no two vertices are connected by an edge. A subset M of E is called a *matching*, if no vertex in V is incident to more than one edge in M. A *perfect matching* is a matching M such that every vertex of the graph is incident to exactly one edge in M. Such a matching can only exist, if $|S| = |T|$. A bipartite graph is said to be *complete*, if $(s_i, t_j) \in E$ for every pair of $s_i \in S$ and $t_j \in T$.

Given a complete weighted bipartite graph $G = (S \cup T, E)$ with integer weights w_{ij} (in this paper we consider positive weights) associated to every edge $(s_i, t_j) \in E$, the problem of finding a minimum weight perfect matching M is known as the *linear assignment problem* (LAP). This problem can be expressed in the form of a linear program by introducing variables x_{ij} with

$$x_{ij} = \begin{cases} 1 & \text{if } (s_i, t_j) \in M, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

The complete linear programming formulation of an LAP of size n herewith becomes

$$\text{Constraints:} \qquad \sum_{i=1}^{n} x_{ij} = 1 \qquad \sum_{j=1}^{n} x_{ij} = 1$$

$$\text{Objective:} \qquad \text{Minimize} \sum_{i,j=1}^{n} w_{ij} x_{ij}$$

The definition of an LAP can easily be extended to cover incomplete bipartite graphs and graphs in which $|S| \neq |T|$. In the first case, w_{ij} is set to ∞ whenever $(s_i, t_j) \notin E$, and a solution to an LAP only exists if $\sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_{ij} < \infty$. In a graph with $|S| > |T|$, a set T' of $|S| - |T|$ additional vertices can be introduced with weights $w_{ij} \equiv 0$ (or the minimum of weights if negative weights are allowed) for all $s_i \in S$. Figure 1 shows the graph for an LAP with $|S| = 4$, $|T| = 3$, and one additional vertex t_4.

2.2 Algorithms for Solving the Linear Assignment Problem

The so-called Hungarian method [16, 13] (or Kuhn-Munkres algorithm) solves the assignment problem in polynomial time. It is based on work of the Hungarian

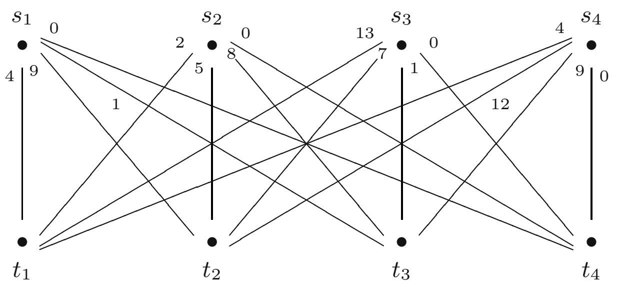


Fig. 1. Graph for an LAP of size 4.

mathematicians König and Egerváry and is an adapted version of the primal-dual algorithm for network flows. For an assignment between two vertex sets of cardinality n the time complexity of the algorithm is $O(n^3)$. It operates on the $n \times n$ weight matrix $W = w_{ij}$ (which is often referred to as cost matrix), and computes a permutation $\sigma = (j_1, \dots, j_n)$ of columns 1, …, n such that the sum $\sum_i w_{ij_i}$ is minimized. $\sum_i w_{ij_i}$ then equals the total weight of the assignment.

The Hungarian method finds an optimal assignment by incrementally applying transformations to the weight matrix. In each step the set of column permutations that correspond to optimal assignments remains unchanged. In the end, a matrix W' is obtained, in which an optimal assignment can be identified easily.

While the Hungarian method returns exactly one minimum cost perfect matching, more than one optimal solution might exist. Uno proposed an algorithm for the enumeration of the complete set of optimal solutions given one optimal assignment [23]. His algorithm is based on the fact that the matrix W' can be used to exclude all edges that cannot be part of any optimal assignment from G, thereby obtaining a graph G'. In G', every perfect matching corresponds to a minimum weight perfect matching in G. The time complexity for enumerating all of the N_p perfect matchings in G' is $O(nN_p)$.

The problem of enumerating all possible assignments in order of increasing cost is also known as the problem of finding K-best perfect matchings. Murty's original algorithm [22] has been improved and generalized by Lawler in [17] and last by Chegireddy and Hamacher [8]. The idea is to partition the solution space P of the minimum weight perfect matching problem for bipartite graphs iteratively into subspaces $P_1, \dots, P_k$. For each of these subspaces P_i, the optimal solution M_i and second best solution N_i to the perfect matching problem are known. The M_i are also known to represent the k-best perfect matchings within P. As the P_i are a partition of the solution space P, the next best solution in P has to be one of the second best matchings N_i, namely the matching with minimum weight w_{N_i}. Let N_j be this matching with minimum weight among all second best matchings N_i, and P_j the corresponding solution space. We partition the solution space P_j into two subspaces P'_j and P_{k+1} such that P'_j contains M_j and P_{k+1} contains N_j. Our $(k+1)$-best matching M_{k+1} is then set to N_j, and solution space P_j is replaced by P'_j. As $N_j (= M_{k+1})$ is no longer included in P_j, we calculate new second best matchings N_j and N_{k+1} for the solution spaces P_j and P_{k+1}. We are then prepared to proceed with finding the next best solution.

The algorithm requires a means to partition the solution space given two different solutions, and an algorithm to find the second best perfect matching in a bipartite graph, given a minimum weight perfect matching.

Chegireddy and Hamacher present an algorithm that takes $O(n^3)$ time to find a second best perfect matching N_i, given the best perfect matching in solution space P_i. In every iteration this algorithm is called two times, which yields an overall complexity of $O(Kn^3)$ for the $(K-1)$ iterations as stated by the authors.

But we also have to take into account the operation of finding the second best matching N_i with minimum weight. Using a priority queue, this is possible in $O(\log k)$ with k being the length of the queue. For the K iterations this yields $\sum_{k=1}^{K} \log k = \log K!$, which is actually growing faster than the factor K in Kn^3. Thus, for the time complexity of the algorithm as a function of n and K, we obtain $O(Kn^3 + \log K!)$. For most practical applications of the algorithm however, this difference is not very important, as K and its influence on the running time is relatively small: the dependency on the problem size n is of greater interest.

3 A Complete Constraint Solver for the Assignment Problem

In this section we explain how do we designed a complete algorithm for the LAP. Recall that completeness of a constraint solver (i.e., the ability of the algorithm to compute a set representing all solutions of given constraints) is required to ensure the completeness of the whole system integrating narrowing and the considered solver. In our case, in order to obtain a single homogeneous and complete solver for the LAP, all of the three algorithms presented in section 2.2 are combined. At first, an optimal solution to the LAP is computed, then all required information in order to calculate the following solutions is stored. The solver returns the solution and a handle to the stored information. Afterwards, the next best solution to the LAP based on the stored information is computed.

The Hungarian method is used to calculate the initial assignment required by the two other algorithms. As Uno's algorithm is a lot more efficient for enumerating optimal solutions than the algorithm by Chegireddy and Hamacher, this algorithm is used first. Then comes the enumeration of suboptimal solutions in order of increasing weight using the algorithm for enumerating K-best solutions.

The transition between the Hungarian method and Uno's algorithm only consists of the construction of the graph G' (see, section 2.2). The transition from Uno's algorithm to the algorithm of Chegireddy and Hamacher is more complicated, as we have to partition the solution space P into N_p subspaces, each containing one of the N_p optimal solutions, which can be seen as the N_p-best solutions to the stated LAP.

We proceed similarly with the algorithm for finding the K-best solutions. The solution space is partitioned by creating the two edge sets I_i and O_i for each optimal solution M_i, with $i \in 1, \ldots, N_p$. When the algorithm has finished, solution subspace P_i is defined as the subspace containing exactly those solutions that contain all edges in I_i, and none of the edges in O_i.

The computation of I_i and O_i is done incrementally. The sets I_1 and O_1 are initially empty. Now suppose that we have a list of k solutions $M_1, \ldots, M_k$ with corresponding solution spaces $P_1, \ldots, P_k$. The computation induced by another solution M_{k+1} is as follows. We first have to find out which solution space P_i $(i = 1, \ldots, k)$ M_{k+1} belongs to. This can be done by checking, if all edges in I_i are contained in M_{k+1} and none of the edges in O_i is contained in M_{k+1}. This process is repeated until the matching solution space P_i is found. We then proceed, as in the algorithm by Chegireddy and Hamacher for the insertion of the solution M_{k+1}. Once obtained the list of all N_p optimal solutions, the second best solution is computed for each of solution spaces $P_1, \ldots, P_{N_p}$.

We now address the complexity of this transition. By inserting a solution, two edges are added to the sets I_i and O_{k+1}; thus at the end, all of these sets together contain $2(N_p - 1)$ edges. This results in a time complexity of $O(N_p)$. Checking, whether a certain edge is contained in a solution, is possible in $O(1)$. This matching procedure must be carried out for each of the N_p solutions, which yields a total complexity of $O(N_p^2)$. Next, the difference between two solutions must be found, each time a solution is inserted. It results in a complexity of $O(N_p n)$ for inserting all of the N_p solutions. Finally, the second best solution has to be computed for each of the N_p solution spaces, which can be done in $O(N_p n^3)$. The total complexity for the transition is therefore $O(N_p^2 + N_p n + N_p n^3)$, which is the same as $O(N_p^2 + N_p n^3)$.

4 A Case Study: Timetabling of Medical Staff

Due to the complicated and often changing set of constraints, timetabling problems are very hard to solve, even for relatively small instances. This is why one of the more successful approaches in the area of timetabling is constraint logic programming. It is used to combine the advantages of declarative programming (facility to handle symbolic constraints) and efficient solver algorithms.

In this section we present a case study using our hybrid system. A first version of this case [5] was done with a standard evaluation strategy. In this older version the system was sometimes not able to find a solution even if one was theoretically possible. Thanks to our complete constraint solver no solution is missed anymore by our system.

The idea to get the best of both worlds, is to use a specific operations research algorithm for a particular constraint to be solved, and use the standard operational semantics for the other constraints. In our case study we use the algorithm from section 3 for LAP constraints and standard narrowing of conditional rewrite rules for the others.

4.1 Timetabling of Medical Staff

We have successfully tested the enhancement of narrowing with the solver we proposed in section 3 on a real world application. The problem consists in the generation of a timetable for the personnel of a Parisian hospital for one week. There are ten shifts (or blocks) for one week, a morning and a late shift for each of the five workdays. The staff is composed of physicians and interns. For each shift the available staff has to be assigned to the open services according to their qualifications. Such an assignment can be represented as follows:

	Echography	Stomatology	Emergency
Monday AM	Bill	Bob	Murray

This shift means that Bill is assigned to the service of Echography, Bob to the service of Stomatology and Murray to the service of Emergency on Monday morning.

In order to obtain an assignment of medical staff to the services, we use the algorithm of section 3. Therefore, the vertex sets of section 2.1 have to be interpreted as services and medical staff. Notice that since there are at least as many staff members as services (if it is not the case, then the assignment problem is unsolvable), and since the LAP is defined for $n \times n$ matrices, we introduce virtual "off work" services. The weights are computed following the suitability of some staff member to some service. I.e. some medical doctors may prefer, or may be more qualified for certain services. Weights also depend on the assignments in other shifts. For example, an infinite weight (i.e. a very large value) is set when the assignment is physically impossible (staff member not working for instance) and semi-infinite weights when the assignment would lead to violation of administrative constraints.

A timetable is acceptable if it fulfills a set of constraints of very different kind and importance. First, all of the open services have to be assigned appropriately qualified staff members. For some services, on a given day, the staff has to be different for the morning and late shifts. For other services it is the opposite, the staff has to be the same all day long. Policies also state that no one should work more than three times on the same service during a week. Also, the workload should be distributed as fairly as possible. An additional constraint is that every staff member should have two afternoons off work per week. These examples do not form an exhaustive list of all constraints (mandatory or not) required for an acceptable timetable.

At this point the reader may notice that there are two large classes of constraints. The first one gathers constraints which are local to a single shift. Basically, they consist in the LAP, i.e. once assigned to a service, a staff member cannot be assigned elsewhere. Pre-assignment, which is obtained by staff members when they give their schedule (day-off for the week), is also local. It consists mainly in the assignment (which is done by setting the weight to 0) of staff members to virtual services. The second class of constraints are the ones with inter-shifts scope (e.g. no more than three assignments per week on a same service). Those two classes are given a different treatment in our implementation and have interactions. Indeed, a locally optimal LAP solution may lead, because of inter-shifts constraints, to a globally inconsistent set of constraints. Suppose for instance that Bill has been assigned to the same service, say Echography, three times in a row from Monday AM, while Bob has been assigned to Stomatology. It is not possible anymore to assign Bill to the service of Echography. Now suppose that, later in the week, Bob is off work and that only Bill may be, due to other constraints, assigned to Echography. We get a set of constraints which is impossible to satisfy. Now, if on Monday AM we switch Bill and Bob, a global solution may be possible, even if the assignment for the shift on Monday AM is sub-optimal. Therefore, it can be useful to find another locally optimal solution, or even a sub-optimal one, in order to build a global solution. Hence, the use of the algorithm from section 3 to enumerate all solutions is mandatory.

4.2 Implementation: Modifying Narrowing by Integrating a Complete LAP Solver

Our implementation is based on constraint functional logic programming. Programs are sets of conditional rewrite rules of the form $l \rightarrow r \mid c$. Informally, any instance, say $\sigma(l)$, of l can be rewritten into the right-hand side under the same instance, say $\sigma(r)$, provided that the instance of the condition $\sigma(c)$ is satisfied, see e.g., [19] for a detailed presentation. A goal is a multiset of basic conditions (usually equations). The operational semantics solves goals using narrowing: it produces a substitution of variables that occur in the goal.

A cost is associated with the violation of every constraint imposed on an acceptable timetable. The more important the constraint is, the higher is the cost. For every shift a cost matrix is computed. It contains the costs for assigning each of the available staff members to the open services. The LAP solver is then used to find an assignment of the staff members to the services, which minimizes the total cost, using the Hungarian method. If an acceptable assignment for a shift is found, the cost matrix for the following shift will be calculated based on previous assignments. This process continues until a timetable for the entire week is obtained.

It is possible that for some shift, no acceptable assignment can be found by the Hungarian method due to assignments in previous shifts. In this case, the LAP solver needs to be called again in order to find alternative assignments for previously assigned shifts. This backtracking mechanism is integrated into the narrowing process.

Within the conditional rewrite system used to represent the constraints of an acceptable timetable [4], two rules are essential for timetable generation:

```
timetable tt → true | (tt=timetable_scheme) and
                      (is_timetable tt)
is_timetable empty → true
is_timetable (cons (shift day period assignments) tail)
             → true |
               (is_timetable tail) and
               (linear_assignment (compute_matrix
                           tail) assignments).
```

The first rule (`timetable`) contains two conditions stating that an acceptable timetable, say `tt`, consists of ten shifts (`tt = timetable_scheme`), and that these shifts have to contain assignments that satisfy the imposed constraints (`is_timetable tt`). The second rule (`is_timetable`) is defined recursively: An empty list of shifts represents a valid timetable. A nonempty list of shifts represents a valid timetable, if the tail of the list contains assignments that do not violate any of the constraints, and if the first shift contains valid assignments. The latter is ensured by the constraint `linear_assignment`, which calls our algorithm for the LAP with the cost matrix that is calculated based on the previous assignments by `compute_matrix tail`, and other constraints that are represented very naturally using rewrite rules. Sample rules are:

```
present (physician Gerd) Monday PM → true
open Echography Friday AM → false
unqualified (physician Fred) Emergency → true
```

The first rule declares that Gerd is present on Monday PM, and therefore can be assigned to a service. The second rule declares that a service is closed, so no working physician for this shift should be assigned to this service. Finally the last rule declares that Fred is not qualified for the service of Emergency. Such rules are used to compute the weights of assignments.

Computation in constraint functional logic programming consists of goal solving. A *goal* – like the condition in a conditional rewrite rule – thereby consists of a list of equations and constraints. A goal is *solved*, if a substitution of the variables occurring in the goal is found such that all equations hold and all constraints are satisfied.

The goal to generate a timetable is of the form `timetable x = true?` Thus, the narrowing process has to find a satisfying substitution for variable `x`. For this purpose a stack of goals is maintained as well as partial substitutions that might lead to a complete satisfying substitution. This corresponds to a depth first traversal of the search space generated by narrowing. At the start, this stack only contains the initial goal entered. Then, for each narrowing step, the goal and the partial substitution on top of the stack are removed. A satisfying substitution for the first equation or constraint in the goal is looked for. For each such substitution, a new goal and partial substitution are put on top of the stack. Whenever a conditional rewrite rule is used in the narrowing step, its conditions have to be added to the goal. If a satisfying substitution for all of the variables has been found, it is displayed as a possible solution to the initial goal.

When computing a timetable, the goal `timetable x = true` is replaced with `x = timetable_scheme` and `is_timetable x = true`. Once the first equation has been solved by substituting `x` with a timetable scheme that contains variables for each service to fill, the rule `is_timetable` is applied recursively. It goes on until the goal becomes a list of ten constraints `linear_assignment`; one for each shift of the week.

A constraint solver that only uses the Hungarian method to find a solution to the assignment problem would solve these constraints one at a time. It computes a substitution that represents an optimal assignment and then removes the constraint from the goal, before it is put back on top of the stack for the next narrowing step. This strategy, however, is not complete, and in our example, as shown earlier, acceptable timetables might not be found, even though they exist.

In our current implementation, the Hungarian method is used initially to compute a solution to the constraint `linear_assignment`. Instead of simply removing the constraint from the goal that is put back on top of the stack, two goals are put on the stack. The first goal still contains the constraint `linear_assignment` for the shift that has just been solved, but it is put onto the stack together with the handle to the stored information that the LAP solver returned after the first call. The next time that this goal is removed from the stack, the LAP algorithm will return the next best assignment, if another acceptable assignment exists. The second goal to be put on top of the stack is the same as in an implementation only based on the Hungarian method. This goal is the next to be removed from the stack in the following narrowing step. The narrowing process in the case that acceptable assignments are found by the Hungarian method for every shift is thus the same as if only the Hungarian method was used as a constraint solver.

The difference between the two only becomes important, if for one of the shifts, no assignment satisfying all of the constraints that an acceptable timetable has to fulfill could be found. In this case, an implementation only based on the Hungarian method for solving the LAP would fail, as it could not solve the goal on top of the stack, and the stack would be empty. In our current implementation, there would be no acceptable assignment for the shift on top of the stack, either, but afterwards the stack would not be empty. Instead, the next goal on top would yield the next best solution for the previous shift.

After a solution has been found for all ten shifts, the program offers to search for alternative timetables by continuing the narrowing process. By first trying to find an assignment for each shift, and backtracking to the previous shift if no such assignment exists, the described strategy implements a simple depth first search of the solution space. At this point we have to point out that this strategy does not guarantee an optimal solution from a global point of view. Indeed, the best choice for a shift may lead to a branch in which global cost is actually higher than if a less optimal solution would have been chosen.

In the following, we assume that we can decide if a given timetable is valid or not by adding the cost for the violated constraints, even without knowing the order, in which the shifts have been assigned[1]. In this case, we obtain a complete timetabling algorithm that finds all acceptable schedules, if only we consider suboptimal solutions for the blocks (shifts) until the cost for a block gets higher than the maximum cost for a timetable. If the optimal assignment for a block has a higher cost than the maximum cost for a timetable, we can be sure that the subset of assigned shifts is already violating more constraints than allowed. Hence, regardless of later assignments, we can never obtain a valid timetable and thus have to step back and consider an alternative assignment for a previous block.

4.3 The Solution Search Tree

With these observations, we are now ready to define the backtracking policy. First, we use a fixed order of assigning the shifts. We define two constants `failure_bound` and `backtrack_bound`. The meaning of the first constant is that there is not enough qualified staff for a shift, if the minimum cost of assignment

[1] This is actually our intention, even though we could construct examples, in which the total cost differs slightly depending on the order of assignment

is at least equal to `failure_bound`. This problem cannot be solved by trying different assignments for previous shifts, and so we can stop the search for a valid timetable, as we can be sure that there is none. If the minimum cost for a certain shift equals or exceeds the second constant, `backtrack_bound`, this means that the assignments that have been found so far cannot be part of any acceptable timetable, even though with different assignments there might exist one.

The search for a solution is then carried out in the following way. For every shift, we calculate the optimal assignment based on the previously assigned blocks. If for one shift the cost of this optimal assignment is greater than or equal to `backtrack_bound`, we return to the previous block and try an alternative optimal solution and continue with the next block. If no more optimal solutions exist, we try suboptimal solutions until the cost exceeds `backtrack_bound`, in which case we step back another block. If for some shift the minimum cost of assignment equals or exceeds `failure_bound`, we stop the search.

We give an example to illustrate the search of a timetable using the presented backtracking scheme. Figure 2 shows a possible search tree for a timetable consisting of five shifts. A solution A_i stands for an assignment for one shift. The indices represent the order in which these solutions are found. For each solution, we state if the solution is optimal with respect to the previously assigned blocks (o), suboptimal (s) or if the assignment is not acceptable (f).

We start by finding an assignment A_1 of minimum cost for the first block. We find optimal assignments A_2 and A_3 for blocks 2 and 3. But in the fourth block it turns out that there is no valid timetable for the whole week containing the assignments A_1–A_3, because the least expensive solution A_4 already exceeds the value of `backtrack_bound`. We step back to block three, find assignments A_5 and A_7, which also cannot be extended to a valid timetable. A_3, A_5 and A_7 represent the only optimal solutions for block three given the assignments A_1 and A_2. So we have to try suboptimal assignments. We find the next cheapest assignment A_9, which already exceeds `backtrack_bound`. We now know that the assignments A_1 and A_2 cannot be extended to an acceptable timetable, so we have to consider alternative substitutions for block 2. It turns out that A_2 was the only optimal assignment for block 2, hence we try suboptimal solutions. The next solution cannot be extended to a valid timetable either, but assignment A_{12} can. Finally, we find the acceptable solution consisting of assignments A_1, A_{12}, A_{13}, A_{14} and A_{15}. We could now try to find alternative solutions by first re-assigning one of the five blocks, and then completing the timetable by finding assignments for the subsequent shifts.

The solution search strategy is complete with respect to finding acceptable timetables whenever they exist. For the first block, we try all possible assignments in order of increasing cost, until the cost of a solution exceeds the value of the constant `backtrack_bound`. Thus, every assignment for the first shift, that could be part of a timetable for the whole week, is considered. For each of these assignments, we try all assignments for the next shift that are consistent with those of the first block and could possibly yield a valid timetable for the whole week. This strategy is pursued for each of the ten blocks. Thus, we only cut

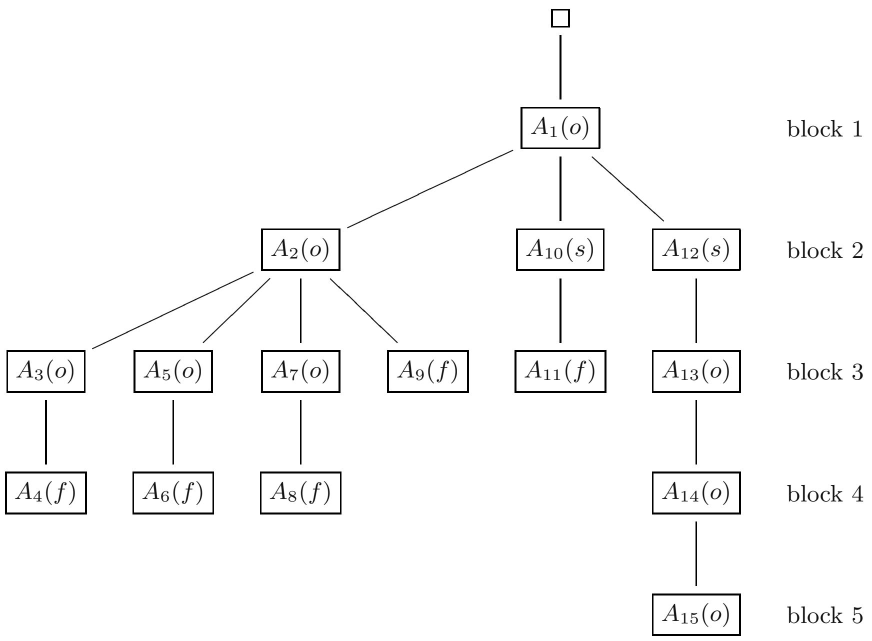


Fig. 2. A possible search tree for a timetable for five shifts

branches of the search tree, if the assignments in the current subset of shifts already violate too many constraints. We can therefore be sure not to ignore any acceptable solution in our search.

For a more precise presentation of the complete set of inference rules for goal transformation we refer to [4]. Let G be a goal. Our proposition consists in giving a particular instance of the following rule where $C_{sol}(p(t_1, \ldots, t_n))$ is computed by the algorithm from section 3:

$$\frac{[p(t_1, \ldots, t_n)] \cup G}{\sigma(G)} \quad \begin{array}{l} \text{if } \sigma \in C_{sol}(p(t_1, \ldots, t_n)) \\ \text{with } C_{sol}(p(t_1, \ldots, t_n)) \text{ being} \\ \text{a complete set of solutions for} \\ \text{constraint } p(t_1, \ldots, t_n) \end{array}$$

where p is an n-ary constraint.

4.4 Practical Results

We have tested the implementation on several generated test cases with real data from the planning of a week for a Parisian hospital. The real world example involved around 20 staff members that had to be assigned to 11 services. An implementation only using the Hungarian method fails in general to find a solution for this problem. The implementation using the complete LAP algorithm was able to find an acceptable timetable with one backtracking step in less than 20 seconds on a 500MHz Pentium III PC. For the generated cases, the program found a solution to a part of the problems within less than five minutes. For others, no solution was found within reasonable time. This was partly due to overconstrained problems, for which no acceptable timetables existed, and partly due to the backtracking strategy. The simple depth-first search that we have implemented performs very well for assignment conflicts that are caused by the assignments made in the previous few shifts. But it is bad for resolving constraints that require re-assignments for shifts that have been assigned relatively early in the goal solving process.

For the problem of timetable generation, the program has to be tested on more real data in order to determine its performance in practice. Also, it would give some material to try other heuristics for a better traversal of the search space. Random test cases are not sufficient for evaluating performance on real data. In cases, where a timetable cannot be found in reasonable time, the program can still be used to generate a timetable by allowing more constraints to be violated. This might already facilitate the process of finding acceptable timetables, if the program is used as a supporting tool.

5 Conclusion

We have proposed an algorithm for LAP, obtained as an integration of three known optimization algorithms, which enumerates a complete set of solutions in order of increasing cost. We have discussed its integration into a constraint functional logic programming language. This system has been tested on a real world problem of timetabling for medical staff. It improves earlier approaches of this kind of problems (with a highly complicated hierarchy of constraints) by combining the expressive facility of declarative programming and the efficiency of operations research algorithms.

The presented depth-first traversal of the solution space performs well for constraints that can be resolved by re-assigning a shift that has been assigned only a few steps before, whereas it does not perform very well for global constraints. Thus, the search for solutions satisfying all of the stated constraints might be improved by employing different backtracking heuristics and strategies. Finding a generally applicable heuristic might be difficult, if not impossible, task.

In the recent proposals for the solution of timetabling problems one can distinguish between two main approaches ([7, 18]). One is based on optimization techniques from operation research like simulated annealing, genetic algorithms and tabu search (e.g. [2, 6, 15, 24]). The second approach is based on constraint programming (e.g. [1, 20, 21]). But to our knowledge our proposition mixing declarative programming and the use of an extended version of the Hungarian method enumerating all solutions (thus retaining completeness) is a novelty.

Several directions for future works are possible. For instance taking into account dynamic constraints as in the on-line management of a fleet of vehicles. For this we would have to modify the operational semantics of a system combining declarative programming and concurrency such as [9]. Another direction would be to enhance the techniques for an efficient traversal of the search space. As we mentioned in section 4.4, we need more practical tests to develop such heuristics. It could be useful to consider other works in this field, for instance [11, 10].

References

1. S. Abdennadher and H. Schlenker. Nurse scheduling using constraint logic programming. In B. McKay, X. Yao, C. S. Newton, J.-H. Kim, and T. Furuhashi, editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference, IAAI-99, Orlando, Florida, July 1999*, pages 838–843. AAAI Press/MIT Press, 1999.
2. U. Aickelin and K. A. Dowsland. An indirect genetic algorithm for a nurse scheduling problem. *Computers and Operations Research*, 31:761–778, 2004.
3. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776 – 822, July 2000.
4. N. Brauner, R. Echahed, G. Finke, H. Gregor, and F. Prost. A complete assignment algorithm and its application in constraint declarative languages. Technical Report 111, Cahier du Laboratoire Leibniz, http://www-leibniz.imag.fr/NEWLEIBNIZ/LesCahiers/, September 2004.
5. N. Brauner, R. Echahed, G. Finke, F. Prost, and W. Serwe. Intégration des méthodes de réécriture et de recherche opérationnelle pour la modélisation et la résolution de contraintes : application à la planification de personnel médical. In *GISEH 2003*, Lyon, January 2003.
6. E. K. Burke, P. D. Causmaecker, and G. V. Berghe. A hybrid tabu search algorithm for the nurse rostering problem. In B. McKay, X. Yao, C. S. Newton, J.-H. Kim, and T. Furuhashi, editors, *Simulated Evolution and Learning, Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL '98, Canberra, Australia, November 24–27 1998, Selected Papers*, volume 1585, pages 93–105. Springer, Berlin, 1999.
7. A. Caprara, F. Focacci, E. Lamma, P. Mello, M. Milano, P. Toth, and D. Vigo. Integrating constraint logic programming and operations research techniques for the crew rostering problem. *Software Practice and Experience*, 28:49–76, 1998.
8. C. R. Chegireddy and H. W. Hamacher. Algorithms for finding k-best perfect matchings. *Discrete Applied Mathematics*, 18:155–165, 1987.
9. R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In J. Lloyd et al., editors, *Proceedings of the* 1^{st} *International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 300 – 314, London, july 2000. Springer Verlag.
10. T. Eiter, W. Faber, C. Koch, N. Leone, and G. Pfeifer. Dlv – a system for declarative problem solving. In C. Baral and M. Truszczynski, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.
11. W. Faber, N. Leone, and G. Pfeifer. Optimizing the computation of heuristics for answer set programming systems. In T. Eiter, W. Faber, and M. Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01,*, volume 2173 of *Lecture Notes in AI (LNAI)*, Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, September 2001. Springer Verlag.

12. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19 & 20:583 – 628, 1994.
13. A. Holland and B. O'Sullivan. Efficient vickrey-pricing for grid service providers. In *Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, July 2003.
14. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *In Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages, POPL'87*, pages 111–119, 1987.
15. L. Kragelund and B. Mayoh. Nurse scheduling generalised. `citeseer.nj.nec.com/kragelund99nurse.html`, 1999.
16. H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.
17. E. L. Lawler. A procedure for computing the k-best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 7(18):401–405, 1972.
18. W. Legierski. Search strategy for constraint-based class–teacher timetabling. In E. Burke and P. D. Causmaecker, editors, *Practice and Theory of Automated Timetabling IV, Fourth International Conference, Gent, Belgium, August 2002, Selected Revised Papers*, volume 2740, pages 247–261. Springer, Berlin, 2003.
19. F.-J. López-Fraguas. A general scheme for constraint functional logic programming. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, volume 632 of *lncs*, pages 213–227. Springer Verlag, 1992.
20. A. Meisels, E. Gudes, and G. Solotorevsky. Employee timetabling, constraint networks and knowledge-based rules: A mixed approach. In E. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling, First International Conference, Edinburgh, UK, August 29–September 1, 1995, Selected Papers*, volume 1153, pages 93–105. Springer, Berlin, 1996.
21. T. Moyaux, B. Chaib-draa, and S. D'Amours. Satisfaction distribuée de contraintes et son application à la génération d'un emploi du temps d'employés. In *Actes du 5e Congrès International de Génie Industriel, Québec, QC, Canada, 26–29 Octobre*, 2003.
22. K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16:682–687, 1968.
23. T. Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In H. W. Leong, , H. Imai, and S. Jain, editors, *Algorithms and Computation*, volume 1350 of *lncs*, pages 92–101. Springer Verlag, 1997.
24. C. A. White and G. M. White. Scheduling doctors for clinical training unit rounds using tabu optimization. In E. Burke and P. D. Causmaecker, editors, *Practice and Theory of Automated Timetabling IV, Fourth International Conference, Gent, Belgium, August 2002, Selected Revised Papers*, volume 2740, pages 120–128. Springer, Berlin, 2003.

Character-Based Cladistics and Answer Set Programming

Daniel R. Brooks[1], Esra Erdem[2], James W. Minett[3], and Donald Ringe[4]

[1] Department of Zoology, University of Toronto, Toronto, Canada
[2] Institute of Information Systems, Vienna University of Technology, Vienna, Austria
[3] Department of Electronic Engineering, Chinese University of Hong Kong, Shatin, Hong Kong
[4] Department of Linguistics, University of Pennsylvania, Philadelphia, USA

Abstract. We describe the reconstruction of a phylogeny for a set of taxa, with a character-based cladistics approach, in a declarative knowledge representation formalism, and show how to use computational methods of answer set programming to generate conjectures about the evolution of the given taxa. We have applied this computational method in two domains: to historical analysis of languages, and to historical analysis of parasite-host systems. In particular, using this method, we have computed some plausible phylogenies for Chinese dialects, for Indo-European language groups, and for *Alcataenia* species. Some of these plausible phylogenies are different from the ones computed by other software. Using this method, we can easily describe domain specific information (e.g. temporal and geographical constraints), and thus prevent the reconstruction of some phylogenies that are not plausible.

1 Introduction

Cladistics (or phylogenetic systematics), developed by Willi Henig [17], is the study of evolutionary relations between species based on their shared traits. Represented diagrammatically, these relations can form a tree whose leaves represent the species, internal vertices represent their ancestors, and edges represent the genetic relationships between them. Such a tree is called a "phylogenetic tree" (or a "phylogeny"). In this paper, we study the problem of reconstructing phylogenies for a set of taxa (taxonomic units) with a character-based cladistics approach[1].

In character-based cladistics, each taxonomic unit is described with a set of "(qualitative) characters" – traits that every taxonomic unit can instantiate in a variety of ways. The taxonomic units that instantiate the character in the same way are assigned the same "state" of that character. Here is an example from [31]. Consider the languages English, German, French, Spanish, Italian, and Russian. A character for these languages is the basic meaning of 'hand':

English	German	French	Spanish	Italian	Russian
hand	*Hand*	*main*	*mano*	*mano*	*ruká*

[1] See [12] for a survey on the other methods for phylogeny reconstruction.

Since the English and German words descended from the same word in their parent language, namely Proto-Germanic **handuz*, by direct linguistic inheritance, those languages must be assigned the same state for this character. The three Romance languages must likewise be assigned a second state (since their words are all descendants of Latin *manus*) and Russian must be assigned a third:

English	German	French	Spanish	Italian	Russian
1	1	2	2	2	3

In character-based cladistics, after describing each taxonomic unit with a set of characters, and determining the character states, the phylogenies are reconstructed by analyzing the character states. There are two main approaches: one is based on the "maximum parsimony" criterion [7], and the other is based on the "maximum compatibility" criterion [3]. According to the former, the goal is to infer a phylogeny with the minimum number of character state changes along the edges. With the latter approach, the goal is to reconstruct a phylogeny with the maximum number of "compatible" characters. Both problems are NP-hard [14, 5]. In this paper we present a method for reconstructing a phylogenetic tree for a set of taxa, with the latter approach.

Our method is based on the programming methodology called answer set programming (ASP) [26, 33, 21]. It provides a declarative representation of the problem as a logic program whose answer sets [15, 16] correspond to solutions. The answer sets for the given formalism can be computed by special systems called answer set solvers. For instance, CMODELS [20] is one of the answer set solvers that are currently available.

We apply our method of reconstructing phylogenies using ASP to historical analysis of languages, and to historical analysis of parasite-host systems.

Histories of individual languages give us information from which we can infer principles of language change. This information is not only of interest to historical linguists but also of interest to archaeologists, human geneticists, physical anthropologists as well. For instance, an accurate reconstruction of the evolutionary history of certain languages can help us answer questions about human migrations, the time that certain artifacts were developed, when ancient people began to use horses in agriculture [24, 25, 32, 35].

Historical analysis of parasites gives us information on where they come from and when they first started infecting their hosts. The phylogenies of parasites, with the phylogenies of their hosts, and with the geographical distribution of their hosts, can be used to understand the changing dietary habits of a host species, to understand the structure and the history of ecosystems, and to identify the history of animal and human diseases. This information allows predictions about the age and duration of specific groups of animals of a particular region or period, identification of regions of evolutionary "hot spots" [2], and thus can be useful to make more reliable predictions about the impacts of perturbations (natural or caused by humans) on ecosystem structure and stability [1].

With this method, using the answer set solver CMODELS, we have computed 33 phylogenetic trees for 7 Chinese dialects based on 15 lexical characters, and 45 phylogenetic trees for 24 Indo-European languages based on 248 lexical, 22 phonological and 12 morphological characters. Some of these phylogenies are plausible from the point of view of historical linguistics. We have also computed 21 phylogenetic trees for 9 species of *Alcataenia* (a tapeworm genus) based on their 15 morphological characters, some of which are plausible from the point of view of coevolution – the evolution of two or more interdependent species each adapting to changes in the other, and from the point of view of historical biogeography – the study of the geographic distribution of organisms.

We have also computed most parsimonious trees for these three sets of taxa, using PARS (available with PHYLIP [13]). Considering also the most parsimonious trees published in [30] (for Indo-European languages), [27] (for Chinese dialects), and [18, 19] (for *Alcataenia* species), we have observed that some of the plausible trees we have computed using the compatibility criterion are different from the most parsimonious ones. This shows that the availability of our computational method based on maximum compatibility can be useful for generating conjectures that can not be found by other computational tools based on maximum parsimony.

As for related work, one available software system that can compute phylogenies for a set of taxa based on the maximum compatibility criterion is CLIQUE (available with PHYLIP), which is applicable only to sets of taxa where a taxonomic unit is mapped to state 0 or state 1 for each character. This prevents us from using CLIQUE to reconstruct phylogenies for the three sets of taxa mentioned above since, in each set, there is some taxonomic unit mapped to state 2 for some character. Another system is the Perfect Phylogeny software of [31], which can compute a phylogeny with the maximum number of compatible characters only when all characters are compatible. Otherwise, it computes an approximate solution. In this sense, our method is more general than the existing ones that compute trees based on maximum compatibility.

Another advantage of our method over the existing ones mentioned above is that we can easily include in the program domain specific information (e.g. temporal and geographical constraints) and thus prevent the reconstruction of some trees that are not plausible.

We consider reconstruction of phylogenies as the first step of reconstructing the evolutionary history of a set of taxa. The idea is then to reconstruct (temporal) phylogenetic networks, which also explain the contacts (or borrowings) between taxonomic units, from the reconstructed phylogenies. The second step is studied in [29, 9, 10].

For more information on the semantics of the ASP constructs used in the logic program below, and on the methodology of ASP, the reader is referred to [22].

2 Problem Description

A *phylogenetic tree* (or *phylogeny*) for a set of taxa is a finite rooted binary tree $\langle V, E\rangle$ along with two finite sets I and S and a function f from $L \times I$ to S, where L is the set of leaves of the tree. The set L represents the given taxonomic units whereas the set V describes their ancestral units and the set E describes the genetic relationships between them. The elements of I are usually positive integers ("indices") that represent, intuitively, qualitative characters, and elements of S are possible states of these characters. The function f "labels" every leaf v by mapping every index i to the state $f(v, i)$ of the corresponding character in that taxonomic unit.

For instance, Fig. 1 is a phylogeny with $I = \{1, 2\}$ and $S = \{0, 1\}$; $f(v, i)$ is represented by the i-th member of the tuple labeling the leaf v.

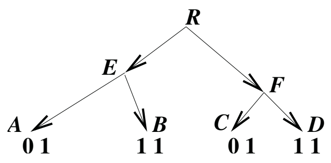


Fig. 1. A phylogeny for the languages A, B, C, D.

A character $i \in I$ is *compatible* with a phylogeny (V, E, L, I, S, f) if there exists a function $g : V \times \{i\} \mapsto S$ such that

(i) for every leaf v of the phylogeny, $g(v, i) = f(v, i)$;
(ii) for every $s \in S$, if the set

$$V_{is} = \{x \in V : \ g(x, i) = s\}$$

is nonempty then the digraph $\langle V, E \rangle$ has a subgraph with the set V_{is} of vertices that is a rooted tree.

A character is *incompatible* with a phylogeny if it is not compatible with that phylogeny. For instance, Character 2 is compatible with the phylogeny of Fig. 1, but Character 1 is incompatible.

The computational problem we are interested in is, given the sets L, I, S, and the function f, to build a phylogeny (V, E, L, I, S, f) with the maximum number of compatible characters. This problem is called the *maximum compatibility problem.* It is NP-hard even when the characters are binary [5].

To solve the maximum compatibility problem, we consider the following problem: given sets L, I, S, a function f from $L \times I$ to S, and a nonnegative integer n, build a phylogeny (V, E, L, I, S, f) with at most n incompatible characters if one exists.

3 Describing the Problem as a Logic Program

We formalize the problem of phylogeny reconstruction for a set of taxa (as described in Section 2) as a logic program. The inputs to this problem are

- a set L of leaves $0, \ldots, k$ ($k > 0$), representing a set of taxa,
- a set I of (qualitative) characters,
- a set S of (character) states,
- a function f mapping every leaf, for every character, to a state, and
- a nonnegative integer n .

The output is a phylogeny (V, E, L, I, S, f) for L with at most n incompatible characters, if one exists.

The logic program describing the problem has two parts. In the first part, rooted binary trees whose leaves represent the given taxa are generated. In the second part, the rooted binary trees according to which there are more than n incompatible characters are eliminated.

Part 1. First note that a rooted binary tree $\langle V, E \rangle$ with leaves L has $2k + 1$ vertices, since $|L| = k+1$. Then V is a set of $2k+1$ vertices. We identify the vertices in V by the numbers $0, \ldots, 2k$. For a canonical representation of a rooted binary tree, i.e., a unique numbering of the internal vertices in V, we ensure that (1) for every edge $(x, y) \in E$, $x > y$, and (2) for any two internal vertices x and y, $x > y$ iff the maximum of the children of x is greater than the maximum of the children of y. We call such a canonical representation of a rooted binary tree an *ordered binary tree*, and describe it as follows.

Suppose that the edges (x, y) of the tree, i.e., elements of E, are described by atoms of the form $edge(x, y)$. The sets of atoms of the form $edge(x, y)$ are "generated" by the rule[2]

$$2 \leq \{edge(x, y) : y \in V, x > y\}^c \leq 2 \leftarrow \qquad (x \in V \setminus L). \tag{1}$$

Each set describes a digraph where there is an edge from every internal vertex to two other vertices with smaller numbers, thus satisfying condition (1). Note that, due to the numbering of the internal vertices above, the in-degree of Vertex $2k$ is 0. Therefore, Vertex $2k$ is the root of the tree.

These generated sets are "tested" with some constraints expressing that the set describes a tree: (a) the set describes a connected digraph, and (b) the digraph is acyclic.

To describe (a) and (b), we "define" the reachability of a vertex y from vertex x in $\langle V, E \rangle$:

$$\begin{array}{ll} reachable(x, y) \leftarrow edge(x, y) \qquad (x, y \in V) \\ reachable(x, y) \leftarrow edge(x, z), reachable(z, y) \qquad (x, y, z \in V). \end{array} \tag{2}$$

For (a), we make sure that every vertex is reachable from the root by the constraint

$$\leftarrow not\ reachable(2k, x) \qquad (x \in V \setminus \{2k\}). \tag{3}$$

For (b), we make sure that no vertex is reachable from itself:

$$\leftarrow reachable(x, x) \qquad (x \in V). \tag{4}$$

To make sure that condition (2) above is satisfied, we first "define" $max_Y(x, y)$ ("Child y of vertex x is larger than the sister of y")

$$max_Y(x, y) \leftarrow edge(x, y), edge(x, y_1) \qquad (x, y, y_1 \in V, y > y_1) \tag{5}$$

and express that a vertex x is larger than another vertex x_1 if the maximum child of x is larger than that of x_1:

$$\leftarrow max_Y(x, y), max_Y(x_1, y_1) \qquad (x, x_1, y, y_1 \in V, y > y_1, x < x_1). \tag{6}$$

Part 2. We eliminate the rooted binary trees $\langle V, E \rangle$, generated by Part 1 above, with more than n incompatible characters as follows. First we identify, for a rooted binary tree $\langle V, E \rangle$, the characters such that, for some function $g : V \times I \mapsto S$, condition (i) holds but condition (ii) does not. Then we eliminate the rooted binary trees for which the number of such characters is more than n.

[2] Rule (1) describes the subsets of the set $\{edge(x, y) : y \in V, x > y\}$ with cardinality 2.

Take any such function g. According to condition (i), g coincides with f where the latter is defined:

$$g(x,i,s) \leftarrow \qquad (x \in L, f(x,i) = s). \tag{7}$$

The internal vertices are labeled by exactly one state for each character by the rule

$$1 \leq \{g(x,i,s) : s \in S\}^c \leq 1 \leftarrow \qquad (x \in V \setminus L, i \in I). \tag{8}$$

To identify the characters for which condition (ii) does not hold, first we pick a root x for each character i and for each state s such that $V_{is} \neq \emptyset$ by the choice rule

$$\{root_{is}(x,i,s)\}^c \leftarrow g(x,i,s) \qquad (x \in V, i \in I, s \in S). \tag{9}$$

We make sure that exactly one root is picked by the constraints

$$\leftarrow root_{is}(x,i,s), root_{is}(y,i,s) \qquad (x,y \in V, x \neq y, i \in I, s \in S) \tag{10}$$

$$\leftarrow \{root_{is}(x,i,s) : x \in V\}\ 0,\ g(y,i,s) \qquad (y \in V, i \in I, s \in S), \tag{11}$$

and that, among the vertices in V_{is}, this root is the closest to the root of the tree by the constraint

$$\leftarrow root_{is}(x,i,s), g(y,i,s), reachable(y,x) \qquad (x,y \in V, i \in I, s \in S). \tag{12}$$

After defining the reachability of a vertex in V_{is} from the root:

$$reachable_{is}(x,i,s) \leftarrow root_{is}(x,i,s) \qquad (x \in V, i \in I, s \in S) \tag{13}$$

$$\begin{aligned} reachable_{is}(x,i,s) \leftarrow g(x,i,s), reachable_{is}(z,i,s), edge(z,x) \\ (x,z \in V, i \in I, s \in S) \end{aligned} \tag{14}$$

we identify the characters for which condition (ii) does not hold:

$$\begin{aligned} incompatible(i) \leftarrow g(x,i,s), not\ reachable_{is}(x,i,s) \\ (x \in V, i \in I, s \in S). \end{aligned} \tag{15}$$

We make sure that there are at most n incompatible characters by the constraint

$$\leftarrow n+1 \leq \{incompatible(i) : i \in I\}. \tag{16}$$

The following theorem shows that the program above correctly describes the maximum compatibility problem stated as a decision problem.

Let Π be the program consisting of rules (1)–(16). Let E_k denote the set of all atoms of the form $edge(x,y)$ such that $0 \leq y < x \leq 2k$.

Correctness Theorem for the Phylogeny Program. *For a given input (L, I, S, f, n), and for a set E of edges that is a rooted binary tree with the leaves L, E describes a phylogeny (V, E, L, I, S, f) with at most n incompatible characters iff E can be represented by the ordered binary tree $Z \cap E_k$ for some answer set Z for Π. Furthermore, every rooted binary tree with the leaves L can be represented like this in only one way.*

The proof is based on the splitting set theorem and uses the method proposed in [11].

Note that constraints (11) and (12) can be dropped from Π, if the goal is to find the minimum n such that Π has an answer set. In our experiments, we drop constraint (11) for a faster computation.

4 Useful Heuristics

We can use the answer set solver CMODELS with the phylogeny program described above to solve small instances of the maximum compatibility problem. Larger data sets, like the Indo-European dataset (Section 7), require the use of some heuristics.

Sometimes the problem for a given input (L, I, S, f, n) can be simplified by making the set I of characters smaller. In particular, we can identify the characters that would be compatible with any phylogeny constructed for the given taxa. For instance, if every taxonomic unit is mapped to a different state at the same character, i.e., the character does not have any "essential" state[3], then we do not need to consider this character in the computation. Similarly, if every taxonomic unit is mapped to the same state at the same character then the character has only one essential state, and that character can be eliminated. Therefore, we can consider just the characters with at least 2 essential states. Such a character will be called *informative* since it is incompatible for some phylogeny. For instance, for the Indo-European languages, out of 275 characters, we have found out that 21 are informative.

Due to condition (ii) of Section 2, every nonempty V_{is} forms a tree in $\langle V, E\rangle$. In each such tree, for every pair of sisters x and y, such that $x, y \in V_{is}$, x and y are labeled for character i in the same way as their parent is labeled. Therefore, to make the computation more efficient, while labeling the internal vertices of the rooted binary tree in Part 2, we can propagate common labels up. For instance, for the *Alcataenia* species, this heuristic improves the computation time by a factor of 2.

In fact, as described in [9, Section 5], we can use partial labelings of vertices, considering essential states, instead of a total one. For instance, for the Indo-European languages, this heuristic improves the computation time by a factor of 3.

Due to the definition of a (partial) perfect network in [9], a character i is compatible with respect to a phylogeny (V, E, L, I, S, f) iff there is a partial mapping g from $V \times \{i\}$ to S such that $(V, E, \emptyset, g)$ is a partial perfect network built on the phylogeny $(V, E, L, \{i\}, S, f\,|_{L\times\{i\}})$. Then, Propositions 4 and 5 from [9] ensure that no solution is lost when the heuristics above are used in the reconstruction of a phylogeny with the maximum number of compatible characters.

5 Computation and Evaluation of Phylogenetic Trees

We have applied the computational method described above to three sets of taxa: Chinese dialects, Indo-European languages, and *Alcataenia* (a tapeworm genus) species. Our experiments with these taxa are described in the following three sections.

To compute phylogenies, we have used the answer set solver CMODELS with the programs describing a set of taxa, preprocessing of the taxa, and reconstruction of a phylogeny. Since the union of these programs are "tight" on their models of completion [8], CMODELS transforms them into a propositional theory [23], and calls a SAT solver to compute the models of this theory, which are identical to the answer sets for the given programs [20]. In our experiments, we have used CMODELS (Version 2.10)

[3] Let (V, E, L, I, S, f) be a phylogeny, with $f : L \times I \mapsto S$. A state $s \in S$ is *essential* with respect to a character $j \in I$ if there exist two different leaves l_1 and l_2 in L such that $f(l_1, j) = f(l_2, j) = s$.

Character	Xiang	Gan	Wu	Mandarin	Hakka	Min	Yue
‘feather’	1	2	2	1	2	1	2
‘give’	1	1	2	3	4	5	2
‘grease’	1	2	1	3	2	2	2
‘know’	1	1	1	2	2	2	2
‘say’	1	3	2	2	1	1	1

Fig. 2. The character states of some informative characters for seven Chinese dialects.

with the SAT solver ZCHAFF (Version Z2003.11.04) [28], on a PC with a 733 MHz Intel Pentium III processor and 256MB RAM, running SuSE Linux (Version 8.1).

In the following, we present the computed trees in the Newick format, where the sister subtrees are enclosed by parentheses. For instance, the tree of Fig. 1 can be represented in the Newick format as ((A, B), (C, D)).

We compare the computed phylogenetic trees with respect to three criteria. First, we identify the phylogenies that are plausible. For the Chinese dialects and Indo-European languages, the plausibility of phylogenies depends on the linguistics and archaeological evidence; for *Alcataenia*, the plausibility of the phylogeny we compute is dependent on the knowledge of host phylogeny (e.g. phylogeny of the seabird family *Alcidae*), chronology of the fossil record, and biogeographical evidence. Since our method is based on maximum compatibility, the second criterion is the number of incompatible characters: the more the number of compatible characters the better the trees are. As pointed out earlier in Section 1, we view reconstructing phylogenies as the first step of reconstructing the evolutionary history of a set of taxonomic units. The second step is then, to obtain a perfect (temporal) phylogenetic network from the reconstructed phylogeny by adding some lateral edges, in the sense of [29, 9, 10]. Therefore, the third criteria is the minimum number of lateral edges (denoting contacts such as borrowings) required to turn the phylogeny into a phylogenetic network.

We also compare these trees to the ones computed by a maximum parsimony method. Usually, to compare a set of trees with another set, “consensus trees” are used. A consensus tree “summarizes” a set of trees by retaining components that occur sufficiently often. We have used the program CONSENSE, available with PHYLIP [13], to find consensus trees.

6 Computing Phylogenetic Trees for Chinese Dialects

We have applied the computational method described above to reconstruct a phylogeny for the Chinese dialects Xiang, Gan, Wu, Mandarin, Hakka, Min, and Yue. We have used the dataset, originally gathered by Xu Tongqiang and processed by Wang Feng, described in [27]. In this dataset, there are 15 lexical characters, and they are all informative. Each character has 2–5 states. For some characters, their states are presented in Fig. 2. After the inessential states are eliminated as explained in Section 4, each character has 2 essential states.

With this dataset, we have computed 33 phylogenies with 6 incompatible characters and found out that there is no phylogeny with less than 6 incompatible characters, in less than an hour. The sub-grouping of the Chinese dialects is not yet established. However,

	Phylogenies	m
15	((Hakka, Min), (Yue, (Gan, (Xiang, (Wu, Mandarin)))))	2
18	((Yue, (Hakka, Min)), (Mandarin, (Wu, (Xiang, Gan))))	3
23	((Hakka, Min), (Yue, ((Xiang, Gan), (Wu, Mandarin))))	3
24	((Yue, (Hakka, Min)), (Gan, (Xiang, (Wu, Mandarin))))	2
27	((Hakka, Min), (Yue, (Mandarin, (Wu, (Xiang, Gan)))))	3

Fig. 3. Phylogenies computed for Chinese dialects, using CMODELS, that are plausible from the point of view of historical linguistics. Each of these trees has 6 incompatible characters, and requires m lateral edges to turn into a perfect phylogenetic network.

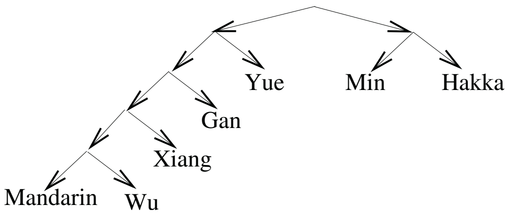


Fig. 4. A plausible phylogeny for Chinese dialects, constructed by CMODELS.

many specialists agree that there are a Northern group and a Southern group. That is, for the dialects we chose in our study, we would expect a (Wu, Mandarin, Gan, Xiang) Northern grouping and a (Hakka, Min) Southern grouping. (It is not clear which group Yue belongs to.) Out of the 33 trees, 5 are more plausible with respect to this hypothesis. One of these plausible trees, Phylogeny 15, is presented in Fig. 4. Among these 5 plausible phylogenies, 2 require at least 2 lateral edges (representing borrowings) to turn into a perfect phylogenetic network; the others require at least 3 edges.

With the dataset above, we have constructed 5 most parsimonious phylogenies using the phylogeny reconstruction program PARS, and observed that none of these phylogenies is consistent with the hypothesis about the grouping of Northern and Southern Chinese dialects.

Using the program CONSENSE, we have computed the majority-consensus tree for our 33 phylogenies: ((Yue, (Hakka, Min)), ((Gan, Xiang), (Wu, Mandarin))). Both this tree and the majority-consensus tree for the 55 most parsimonious trees of [27] are consistent with the more conventional hypothesis above, grouping Yue with the Southern dialects.

All of the 33 phylogenies we have computed correspond to the trees of Types I–III in [27]. Each of the remaining 22 trees of [27] has 7 incompatible characters, but they have the same degree of parsimony as the other 33 trees. This highlights the difference between a maximum parsimony method and a maximum compatibility method.

7 Computing Phylogenetic Trees for Indo-European Languages

We have applied the computational method described above to reconstruct a phylogeny for the Indo-European languages Hittite, Luvian, Lycian, Tocharian A, Tocharian B, Vedic, Avestan, Old Persian, Classical Armenian, Ancient Greek, Latin, Oscan, Um-

Character	Ancient Greek	Old Church Slavonic	Old English	Old High German	Latin	Old Persian
'child'	3	8	10	18	12	15
'father'	2	1	2	2	2	2
'free'	3	8	10	10	3	14
'laugh'	2	7	9	9	11	14
'tear'	2	4	2	2	2	7

Fig. 5. The character states of some informative characters for six Indo-European languages.

brian, Gothic, Old Norse, Old English, Old High German, Old Irish, Welsh, Old Church Slavonic, Old Prussian, Lithuanian, Latvian, and Albanian. We have used the dataset assembled by Don Ringe and Ann Taylor, with the advice of other specialist colleagues. This dataset is described in [31].

There are 282 informative characters in this dataset. Out of 282 characters, 22 are phonological characters encoding regular sound changes that have occurred in the prehistory of various languages, 12 are morphological characters encoding details of inflection (or, in one case, word formation), and 248 are lexical characters defined by meanings on a basic word list. For each character, there are 2–24 states. Some of the character states for some Indo-European languages are shown in Fig. 5.

To compute phylogenetic trees, we have treated as units the language groups Balto-Slavic (Lithuanian, Latvian, Old Prussian, Old Church Slavonic), Italo-Celtic (Oscan, Umbrian, Latin, Old Irish, Welsh), Greco-Armenian (Ancient Greek, Classical Armenian), Anatolian (Hittite, Luvian, Lycian), Tocharian (Tocharian A, Tocharian B), Indo-Iranian (Old Persian, Avestan, Vedic), Germanic (Old English, Old High German, Old Norse, Gothic), and the language Albanian.

For each language group, we have obtained the character states by propagating the character states for languages up, similar to the preprocessing of [9]. After propagating character states up, we have found out that grouping Baltic and Slavic makes 1 character incompatible, and grouping Italic and Celtic makes 6 characters incompatible. (For the purposes of this experiment we accept the Italo-Celtic subgroup as found in [31] largely on the basis of phonological and morphological characters.) Other groupings do not make any character incompatible. Therefore, we have not considered these 7 characters while computing a phylogenetic tree, as we already know that they would be incompatible with every phylogeny.

Then we have identified the characters that would be compatible with every phylogeny built for these 7 language groups and the language Albanian. By eliminating such characters as explained in Section 4, we have found out that, out of $282 - 7$ characters, 21 characters are informative. Out of those 21, 2 are phonological ('P2' and 'P3') and 1 is morphological ('M5'). Each character has 2–3 essential states.

While computing phylogenetic trees for the 7 language groups and the language Albanian, we have ensured that each tree satisfies the following domain-specific constraints: Anatolian is the outgroup for all the other subgroups; within the residue, Tocharian is the outgroup; within the residue of that, Italo-Celtic, and possibly Albanian are outgroups, but not necessarily as a single clade; Albanian cannot be a sister of Indo-Iranian or Balto-Slavic.

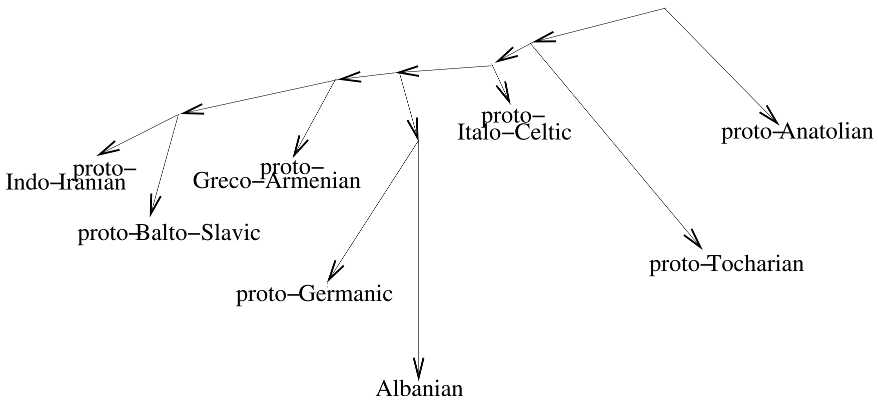


Fig. 6. A plausible phylogeny computed for Indo-European languages, using CMODELS.

The domain-specific information above can be formalized as constraints. For instance, we can express that Anatolian is the outgroup for all the other subgroups by the constraint

$$\leftarrow not\ edge(2k, 6)$$

where $2k$ is the root of the phylogeny, and 6 denotes proto-Anatolian.

Another piece of domain-specific information is about the phonological and morphological characters. The phonological and morphological innovations (except 'P2' and 'P3') considered in the dataset are too unlikely to have spread from language to language, and that independent parallel innovation is practically excluded. Therefore, while computing phylogenetic trees, we have ensured that these characters are compatible with them. This is achieved by adding to the program the constraint

$$\leftarrow incompatible(i) \qquad (i \in IC \cap MP)$$

where MP is the set of all morphological and phonological characters except 'P2' and 'P3'.

With 21 informative characters, each with 2–3 essential states, we have computed 45 phylogenetic trees for the 7 language groups above and the language Albanian, in a few minutes. Out of the 45 phylogenies computed using CMODELS, 34 are identified by Don Ringe as plausible from the point of view of historical linguistics. Fig. 6 shows the most plausible one with 16 incompatible characters. This phylogeny is identical to the phylogeny presented in [31], which was computed with a greedy heuristic using the Perfect Phylogeny software in about 8 days (Don Ringe, personal communication), and used in [29, 9, 10] to build a perfect phylogenetic network for Indo-European.

With the same Indo-European dataset obtained after preprocessing (with 21 informative characters, each with 2–3 essential states), we have also computed a most parsimonious phylogeny using the computational tool PARS: (Anatolian, Tocharian, (Greco-Armenian, ((Albanian, ((Italo-Celtic, Germanic), Balto-Slavic)), Indo-Iranian))). Some other most parsimonious phylogenies constructed for Indo-European languages are due to [30], where the authors use PAUP [34] with the dataset Isidore Dyen [6] to generate

Character	*A. Longicervica*	*A. Cerorhincae*	*A. Pygmaeus*	*A. Meinertzhageni*	*A. Campylacantha*
uterus	1	1	1	1	1
size of hooks	1	0	1	2	2
position in host	1	0	1	1	0
position of hooks	1	0	0	2	1

Fig. 7. The character states of some characters for five *Alcataenia* species.

phylogenies. None of these most parsimonious trees is consistent with the domain-specific information described above, and thus none is plausible from the point of view of historical linguistics. On the other hand, we should note that Dyen's dataset is not very reliable since it is a purely lexical database from modern languages.

8 Computing Phylogenetic Trees for *Alcataenia* Species

With the computational method presented above, we can also infer phylogenies for some species, based on some morphological features. Here we have considered 9 species of *Alcataenia* – a tapeworm genus whose species live in alcid birds (puffins and their relatives): *A. Larina* (LA), *A. Fraterculae* (FR), *A. Atlantiensis* (AT), *A. Cerorhincae* (CE), *A. Pygmaeus* (PY), *A. Armillaris* (AR), *A. Longicervica* (LO), *A. Meinertzhageni* (ME), *A. Campylacantha* (CA). We have used the dataset described in [19].

In this dataset, there are 15 characters, each with 2–3 states. For some characters, their states are presented in Fig. 7. After preprocessing, we are left with 10 informative characters, each with 2 essential states.

According to [19], the outgroup for all *Alcataenia* species is *A. Larina*. We have expressed this domain-specific information by the constraint

$$\leftarrow not\ edge(2k, 0)$$

where $2k$ is the root of the phylogeny, and 0 denotes *A. Larina*.

With the dataset obtained after preprocessing, we have found out that, for *Alcataenia*, there is no phylogeny with less than 5 incompatible characters. Then we have computed 18 phylogenies, with 5 incompatible characters, for *Alcataenia*, in less than 30 minutes. One of these phylogenies is presented in Fig. 8.

For the plausibility of the phylogenies for *Alcataenia*, we consider the phylogenies of its host *Alcidae* (a seabird family) and the geographical distributions of *Alcidae*. This information is summarized in Table 3 of [19]. For instance, according to host and geographic distributions over the time, diversification of *Alcataenia* is associated with sequential colonization of puffins (parasitized by *A. Fraterculae* and *A. Cerorhincae*), razorbills (parasitized by *A. Atlantiensis*), auklets (parasitized by *A. Pygmaeus*), and murres (parasitized by *A. Armillaris*, *A. Longicervica*, and *A. Meinertzhageni*). This pattern of sequential colonization is supported by the phylogeny of *Alcidae* in [4]. Out of the 18 trees we have computed, only two are consistent with this pattern. (One of

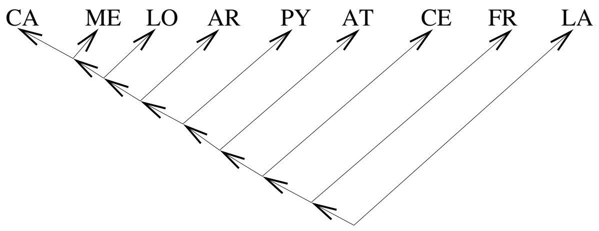


Fig. 8. A plausible phylogeny computed, using CMODELS, for *Alcataenia* species.

them is shown in Fig. 8.) Both trees are plausible also from the point of view of historical biogeography of *Alcataenia* in *Alcidae*, summarized in [19]. Each plausible tree needs 3 lateral edges to turn into a perfect phylogenetic network.

With the *Alcataenia* dataset described above, we have computed a most parsimonious tree using PARS, which is very similar to the phylogeny of Fig. 8, and to the most parsimonious phylogeny computed for the *Alcataenia* species above (except *A. Atlantiensis*) by Eric Hoberg [18][Fig. 1].

According to [18, 19], a more plausible phylogeny for *Alcataenia* is the variation of the phylogeny of Fig. 8 where *A. Armillaris* and *A. Longicervica* are sisters. We can express that *A. Armillaris* and *A. Longicervica* are sisters by the constraint

$$\leftarrow not\ sister(2, 4)$$

where 2 and 4 denote *A. Armillaris* and *A. Longicervica* respectively. By adding this constraint to the problem description, we have computed 3 phylogenies, each with 6 incompatible characters, in less than 10 minutes; their strict consensus tree is identical to the one presented in Fig. 5 of [19]. It is not the most parsimonious tree.

9 Conclusion

We have described how to use answer set programming to generate conjectures about the phylogenies of a set of taxa based on the compatibility of characters. Using this method with the answer set solver CMODELS, we have computed phylogenies for 7 Chinese dialects, and for 24 Indo-European languages. Some of these trees are plausible from the point of view of historical linguistics. We have also computed phylogenies for 9 *Alcataenia* species, and identified some as more plausible from the point of view of coevolution and historical biogeography.

Some of the plausible phylogenies we have computed (e.g. the ones computed for Indo-European) using CMODELS are different from the ones computed using other software, like PARS of PHYLIP, based on maximum parsimony. This shows that the availability of our computational method based on maximum compatibility can be useful for generating conjectures that can not be found by other computational tools.

One software that can compute phylogenies for a set of taxa based on the maximum compatibility criterion is CLIQUE (available with PHYLIP), which is applicable only to sets of taxa where a taxonomic unit is mapped to state 0 or state 1 for each character.

Another one is the Perfect Phylogeny software of [31], which can compute a phylogeny with the maximum number of compatible characters only when all characters are compatible. Our method is applicable to sets of taxa (like the ones we have experimented with) where a taxonomic unit can be mapped to multiple states. Also, it guarantees to find a tree with the maximum number of compatible characters, if one exists, when all characters may not be compatible. In this sense, our method is more general than the existing ones that compute trees based on maximum compatibility.

Another advantage of our method over the existing ones mentioned above is due to answer set programming. Its declarative representation formalism allows us to easily include in the program domain specific information, and thus to prevent the reconstruction of some phylogenetic trees that are not plausible. Moreover, well-studied properties of programs in this formalism allow us to easily prove that the maximum compatibility problem is correctly described as a decision problem by the phylogeny program.

Acknowledgments

We have had useful discussions with Selim Erdoğan and Vladimir Lifschitz on the formalization of the problem, and with Wang Feng on the plausibility of phylogenies for Chinese dialects. Eric Hoberg, Luay Nakhleh, William Wang, and Tandy Warnow have supplied relevant references. Brooks was supported by an NSERC Discovery Grant to DRB. Ringe was supported by NSF BCS 03-12911. Erdem was supported in part by FWF P16536-N04; part of this work was done while she visited the University of Toronto, which was made possible by Hector Levesque and Ray Reiter.

References

1. D. R. Brooks, R. L. Mayden, and D. A. McLennan. Phylogeny and biodiversity: Conserving our evolutionary legacy. *Trends in Ecology and Evolution*, 7:55–59, 1992.
2. D. R. Brooks and D. A. McLennan. *Phylogeny, Ecology, and Behavior: A Research Program in Comparative Biology*. Univ. Chicago Press, 1991.
3. J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:311–326, 1965.
4. R. M. Chandler. *Phylogenetic analysis of the alcids*. PhD thesis, University of Kansas, 1990.
5. W. H. E. Day and D. Sankoff. Computational complexity of inferring phylogenies by compatibility. *Systematic Zoology*, 35(2):224–229, 1986.
6. I. Dyen, J. B. Kruskal, and P. Black. An Indoeuropean classification: a lexicostatistical experiment. *Transactions of the American Philosophical Society*, 82:1–132, 1992.
7. A. W. F. Edwards and L. L. Cavalli-Sforza. Reconstruction of evolutionary trees. *Phenetic and Phylogenetic Classification*, pp. 67–76, 1964.
8. E. Erdem and V. Lifschitz. Tight logic programs. *TPLP*, 3(4–5):499–518, 2003.
9. E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the evolutionary history of Indo-European languages using answer set programming. In *Proc. of PADL*, pp. 160–176, 2003.
10. E. Erdem, V. Lifschitz, and D. Ringe. Temporal phylogenetic networks and answer set programming. In progress, 2004.
11. S. T. Erdoğan and V. Lifschitz. Definitions in answer set programming. In *Proc. of LPNMR*, pp. 114–126, 2004.

12. J. Felsenstein. Numerical methods for inferring evolutionary trees. *The Quarterly Review of Biology*, 57:379–404, 1982.
13. J. Felsenstein. PHYLIP (Phylogeny inference package) version 3.6.
14. L. R. Foulds and R. L. Graham. The Steiner tree problem in Phylogeny is NP-complete. *Advanced Applied Mathematics*, 3:43–49, 1982.
15. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of ICLP/SLP*, pp. 1070–1080, 1988.
16. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
17. W. Hennig. *Grundzuege einer Theorie der Phylogenetischen Systematik*. Deutscher Zentralverlag, 1950.
18. E. P. Hoberg. Evolution and historical biogeography of a parasite-host assemblage: *Alcataenia* spp. (Cyclophyllidea: Dilepididae) in Alcidae (Chradriiformes). *Canadian Journal of Zoology*, 64:2576–2589, 1986.
19. E. P. Hoberg. Congruent and synchronic patterns in biogeography and speciation among seabirds, pinnipeds, ans cestodes. *J. Parasitology*, 78(4):601–615, 1992.
20. Yu. Lierler and M. Maratea. Cmodels-2: SAT-based answer sets solver enhanced to non-tight programs. In *Proc. of LPNMR*, pp. 346–350, 2004.
21. V. Lifschitz. Answer set programming and plan generation. *AIJ*, 138:39–54, 2002.
22. V. Lifschitz. Introduction to answer set programming. Unpublished draft, 2004.
23. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.
24. V.H. Mair, editor. *The Bronze Age and Early Iron Age Peoples of Eastern Central Asia*. Institute for the Study of Man, Washington, 1998.
25. J.P. Mallory. *In Search of the Indo-Europeans*. Thames and Hudson, London, 1989.
26. V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pp. 375–398, 1999.
27. J. W. Minett and W. S.-Y. Wang. On detecting borrowing: distance-based and character-based approaches. *Diachronica*, 20(2):289–330, 2003.
28. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC*, 2001.
29. L. Nakhleh, D. Ringe, and T. Warnow. Perfect phylogenetic networks: A new methodology for reconstructing the evolutionary history of natural languages. *Language*, 2005. To appear.
30. K. Rexova, D. Frynta, and J. Zrzavý. Cladistic analysis of languages: Indo-European classification based on lexicostatistical data. *Cladistics*, 19:120–127, 2003.
31. D. Ringe, T. Warnow, and A. Taylor. Indo-European and computational cladistics. *Transactions of the Philological Society*, 100(1):59–129, 2002.
32. R.G. Roberts, R. Jones, and M.A. Smith. Thermoluminescence dating of a 50,000-year-old human occupation site in Northern Australia. *Science*, 345:153–156, 1990.
33. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *AIJ*, 138:181–234, 2002.
34. D.L. Swofford. PAUP* (Phylogenetic analysis under parsimony) version 4.0.
35. J.P. White and J.F. O'Connell. *A Prehistory of Australia, New Guinea, and Sahul*. Academic Press, New York, 1982.

Role-Based Declarative Synchronization for Reconfigurable Systems[⋆]

Vlad Tanasescu and Paweł T. Wojciechowski

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{Vlad.Tanasescu,Pawel.Wojciechowski}@epfl.ch

Abstract. In this paper we address the problem of encoding complex concurrency control in reconfigurable systems. Such systems can be often reconfigured, either statically, or dynamically, in order to adapt to new requirements and a changing environment. We therefore take a declarative approach and introduce a set of high-level programming abstractions which allow the programmer to easily express complex synchronization constraints in multithreaded programs. The constructs are based on our model of *role-based synchronization (RBS)* which assumes attaching roles to concurrent threads and expressing a synchronization policy between the roles. The model is illustrated by describing an experimental implementation of our language as a design pattern library in OCaml. Finally, we also sketch a small application of a web access server that we have implemented using the RBS design pattern.

1 Introduction

Our motivating example of reconfigurable systems are networked applications, such as modular group communication middleware protocols [7, 14] and web services [27]. Software components in these applications interact with clients and process network messages. Due to efficiency reasons, different components (or objects) may be accessed concurrently, with possible dependencies on other components. To provide data consistency and a quality of service, complex synchronization policies of object accesses are required. Unfortunately, a given composition of application components may often change in order to adapt to a new environment or changing user requirements. This means that the synchronization policy may need to be revised as well, and the corresponding code changed accordingly, making programming of such systems a difficult task.

Developing multithreaded systems is considerably more difficult than implementing sequential programs due to several reasons:

– traditional concurrency constructs, such as monitors and conditional variables, are used to express synchronization constraints at the very low level of individual accesses to shared objects (thread safety);

⋆ Research supported by the Swiss National Science Foundation under grant number 21-67715.02 and Hasler Stiftung under grant number DICS-1825.

- embedding the implementation of a synchronization policy in the main code compromises both a good understanding of the application logic, i.e. we are not sure from the first look what the application code does, and also an understanding of the policy expressed;
- the notions of *thread roles* such as producer or consumer, which are essential for the understanding of a given policy, tend to disappear beyond an accumulation of lines of code, just as the logical essence of a sequential program gets lost when expressed in, say, an assembly language;
- synchronization constructs are usually entangled with instructions of the main program, which means that the correctness of concurrent behaviour is bound to the correctness of the entire application; this feature complicates maintenance and code reuse – some of the most advocated reasons for using components.

We therefore study *declarative synchronization*, which assumes a separation of an object's functional behaviour and the synchronization constraints imposed on it. Such an approach enables to modify and customize synchronization policies constraining the execution of concurrent components, without changing the code of component objects, thus making programming easier and less error-prone.

While some work on such separation of concerns exists (see [8, 5, 23, 22, 16] among others) and example languages have been built (see [21, 22, 15, 16]), as far as we know, our solution is novel. It shares a number of design features with these languages, such as support for "declarative style" and "separation of synchronization aspects". However, there are also important differences in the model and implementation (we characterize them briefly in §2). Some of our motivations and goals are different, too.

In this paper, we propose a *role-based synchronization (RBS)* model with a constraint language to express concurrency control between the roles. Our design has been guided by two main requirements:

- to keep the semantics of synchronization control attached to roles involved in the specification of a concurrent problem rather than to instances of components and objects (or their abstractions);
- to allow expressing concurrent strategies independently from the main code of applications, with a possibility to switch between different strategies on-the-fly (with some control on the moment of switching).

Our long term goal is to develop support of declarative synchronization for component-based systems that can be dynamically reconfigured or adapted to changing requirements or a new environment. For instance, when a mobile device is moved from outside to inside a building, it may reconfigure its suite of network protocols on-the-fly. Another example are mobile collaborative systems, such as the *Personal Assistant (PA)* application [29]; the PA service components may need to switch between user interfaces at runtime, depending on a given device accessed by the user at a time (e.g. a hand-held device or PC workstation).

In our previous work, Wojciechowski [28] has studied typing and verification of synchronization policies expressed using *concurrency combinators* – a simple language for declaring constraints between static modules (instead of dynamic roles). This paper provides an alternative model and new insight into the implementation aspects of declarative synchronization.

To illustrate our approach, we describe an example RBS design pattern package that we have implemented in OCaml [17]. Notably, our experimental implementation tackles both aspects of our design (i.e. separation of concerns and expressiveness) without using any precompilation tools. We believe however that more traditional programming languages, such as Java [6] and C++ [26], could be also used.

Our current implementation of RBS can only switch between synchronization policies that have been predefined by a given RBS package. This is sufficient for the above example applications of reconfigurable systems. Ultimately, we would like to be able to download a new policy dynamically. The OCaml distribution does not support dynamic class loading, however. We intend therefore to experiment with programming languages that support dynamic data types with dynamic typing and binding, such as Python [20] and Acute [24]. The latter is an extension of the OCaml language with dynamic loading and controlled rebinding to local resources. We leave this for future work.

The paper is organized as follows. §2 contains related work, §3 introduces the RBS model and constraint language, §4 describes an example RBS package, §5 discusses dynamic switching between synchronization policies, §6 illustrates our approach using a small web application, and §7 concludes.

2 Related Work

There have been recently many proposals of concurrent languages with novel synchronization primitives, e.g. Polyphonic C# [1] and JoCaml [3] that are based on the join-pattern abstraction [4], and Concurrent Haskell [10], Concurrent ML [18], Pict [19] and Nomadic Pict [25, 29], with synchronization constructs based on channel abstractions. They enable to encode complex concurrency control more easily than when using standard constructs, such as monitors and locks.

The above work is orthogonal to the goals of this paper. We are primarily focused on a declarative way of encoding synchronization through separation of concerns (see [8, 12, 11, 9] among others). The low-level details of the RBS implementation resemble the idea of aspect-oriented programming. Below we discuss example work in these two areas.

Separation of Concurrency Aspects. For a long time, the object-oriented community has been pointing out, under the term *inheritance anomaly* [13], that concurrency control code interwoven with the application code of classes, can represent a serious obstacle to class inheritance, even in very simple situations.

For instance, consider a library class `SimpleBuffer` implementing a bounded buffer shared between concurrent producers and consumers. The class provides public methods `input()` and `output()` which can be used to access the buffer. The implementation of these methods would normally use some conditional variables like `is_buffer_full` or `is_buffer_empty` in order to prevent the buffer from being accessed when it is full or empty. Suppose we want to implement a buffer in which we would like to add a condition that nobody can access the buffer after a call to a method `freezeBuffer()` has been made. But in this case, we are not able to simply extend the class `SimpleBuffer`. We also need to rewrite the code of both methods `output()` and `input()` in order to add the new constraint on the buffer usage!

Milicia and Sassone [15, 16] address the inheritance anomaly problem and propose an extension of Java with a linear temporal logic to express synchronization constraints on method calls. Their approach is similar to ours (although our motivation is the ease of programming and expressiveness). However, it requires a precompilation tool in order to translate a program with temporal logic clauses into Java source code, while our approach uses the facilities provided by the host language. Also, their language does not allow for expressing synchronization constraints that require access to a program's data structures.

Ramirez *et al.* [21, 22] have earlier proposed a simple constraint logic language for expressing temporal constraints between "marked points" of concurrent programs. The approach has been demonstrated using Java, extended with syntax for marking. Similarly to the approach in [15, 16], the language has however limited expressiveness. While our constraint declarations can freely access data in a thread-safe way, and call functions of the application program, their constraints are not allowed to refer to program variables. Also, composite synchronization policies (on groups of threads) are not easily expressible.

The previous work, which set up goals similar to our own is also by Ren and Agha [23] on separation of an object's functional behaviour and the timing constraints imposed on it. They proposed an actor-based language for specifying and enforcing at runtime real-time relations between events in a distributed system. Their work builds on the earlier work of Frølund and Agha [5] who developed language support for specifying multi-object coordination, expressed in the form of constraints that restrict invocation of a group of objects.

In our previous work, Wojciechowski [28] has studied declarative synchronization in the context of a calculus of concurrency combinators. While in this paper we propose a language for expressing dynamic constraints between role-oriented *threads*, the concurrency combinators language is used to declare synchronization constraints between static *code fragments*. The calculus is therefore equipped with a static type system that can verify if the matching of a policy and program is correct. Typable programs are guaranteed to make progress.

Aspect-Oriented Programming. *Aspect-Oriented Programming (AOP)* is a new trend in software engineering. The approach is based on separately specifying the various *concerns* (or *aspects*) of a program and some description of their relationship, and then relying on the AOP framework to *weave* [9] or compose them together into a coherent program. For instance, error handling or security checks can be separated from a program's functional core. Hürsch and Lopes [8] identify various concerns, including synchronization. Lopes [12] describes a programming language D, that allows thread synchronization to be expressed as a separate concern. More recently, AOP tools have been proposed for Java, such as AspectJ [11]. They allow aspects to be encoded using traditional languages and weaved at the intermediate level of Java bytecode. The programmer writes aspect code to be executed *before* and *after* the execution of *pointcuts*, where a pointcut usually corresponds to invocations of an application method.

The code weaving techniques can be, of course, applied to synchronization aspects too, and by doing so, we can achieve separation of concurrency concerns. However, by using a pure AOP approach, we are not getting yet more expressiveness. In this paper, we propose a set of language abstractions that are used to *declare* an arbitrary synchronization policy. The policy is then effectuated automatically at runtime by a concurrency controller implementing a given RBS design pattern. Our current implementation of RBS design patterns in OCaml does not resort to external precompilation tools.

In §3, we describe the RBS model and the constraint language. Then we explain the implementation details in §4, using an example RBS design pattern.

3 Role-Based Synchronization

In this section, we describe our simple but expressive model of Role-Based Synchronization (RBS). By looking at the classical concurrency problems, such as Producer-Consumer, Readers-Writers, and Dining Philosophers, we can identify two essential semantic categories which are used to describe these problems, i.e. roles and constraints imposed on the roles. Below we characterize these two categories.

3.1 Thread Roles

Threads in concurrent programs are spawned to perform certain *roles*, e.g. producers, consumers, readers, writers, and philosophers. Below we confuse roles and threads unless otherwise stated, i.e. a role means one, or possibly many concurrent threads, that are logically representing the role.

Roles can execute *actions*, e.g. to output a value in the buffer, to write a value to a file, to eat rice. Roles can be in different *states* during program execution. Some actions are allowed only in certain states, i.e. in order to execute an action a role must first *enter* a state that allows the action to be executed, unless the role is already in such a state.

The common synchronization problems are usually concerned with accessing *shared resources* (or *objects*) by roles in an exclusive manner, e.g. a buffer, a file, a fork and a rice bowl. We can therefore identify two role states (more refined states can also be defined): the state of being able to call methods of a shared object (and execute the corresponding action) and the state of waiting to be able to do so. We denote by `In` the former state (where "in" means being *in* the position to execute actions) and by `Wait` the latter state.

For instance, a producer is waiting if the buffer is full, a writer is waiting when another writer is writing, a philosopher is waiting if at least one fork is missing. Otherwise, these roles are in the position to execute all role's actions which have been defined on the buffer, file, and rice bowl.

3.2 Synchronization Constraints

Synchronization constraints describe conditions on when roles are allowed to enter a state and execute the role's actions allowed by the state.

For instance, a consumer can only input a value if neither producer nor other consumer is currently accessing the buffer *and* there is actually some value in the buffer to be obtained; many concurrent readers can read a file concurrently if there is no writer writing to the file at the same time; and a philosopher can eat rice only if two forks are available.

Thus, synchronization constraints specify a synchronization problem – if an implementation obeys all the constraints defined, then it provides a correct solution to the problem. Failure to satisfy any constraint in accessing shared objects by roles may cause incorrect program behaviour. Below we describe two parts that jointly define a synchronization constraint: a synchronization policy and a logical condition.

Synchronization Policy. A *synchronization policy* (or *policy* in short) defines a constraint between different roles and their states. We say that a policy is *satisfied* if the constraint always holds, and it is *violated* otherwise. Essentially, a policy specifies when a role is permitted or forbidden to enter a given state.

A popular policy is "mutual exclusion", which simply states that some roles cannot be simultaneously in the critical section defined by access to a shared object (method, data field, etc.). For instance, a producer and consumer cannot access a buffer at the same time, two writers are not allowed to simultaneously write to the file, and two philosophers cannot share the same fork. More precisely, they are not allowed at any time to be both in the same state `In`.

Logical Conditions. Satisfying the synchronization policy is the necessary but often not sufficient condition to solve a synchronization problem. For instance, a consumer cannot input a value if there is no value in the buffer. Thus, we also introduce a *logical condition* which specifies a constraint on an object requested by a role; this condition must be satisfied in order to allow a method of the object to be called by the role.

3.3 Constraint Language

The above observations led us to a simple constraint language, defined in Fig. 1. The language is expressive enough to describe most of the common concurrency problems. Below we describe the syntactic constructs and example policy types. We use $[\overline{x}]$ to denote a list of elements $\overline{x} = x_1, ..., x_n$, where $[]$ is the empty list.

Objects. Shared objects, denoted o, are declared as lists of pairs $(a, [\overline{S}])$, where the meaning of each pair is that the object method a (action a) can be called (executed) by a role only if the role is in one of the states mentioned in the list $[\overline{S}]$ (an empty list $[]$ is used if the method can be called in *any* state).

For instance, `Buffer = [(output, [In]); (is_full, [])]` declares an object `Buffer` which has a method `output`, which can be called only by a role which is in state `In`, and a method `is_full` which can be called in any state.

Constraint Language:

Thread roles	$R \in \mathit{Roles}$
Objects	$o \in \mathit{Objects}$
Actions	$a \in \mathit{Actions}$
States	$S \in \mathit{States}$
Families	$F \in \mathit{Families}$
Constraints	$K \in \mathit{Constraints}$
Sync policies	$P \in \mathit{Policies}$
Policy types	$T \in \mathit{Types}$
Conditions	$C \in o.a \rightarrow \mathit{boolean}$

$$\begin{aligned}
o &::= [(a, [\overline{S}]); ...; (a, [\overline{S}])] \\
F &::= ([\overline{R}], [\overline{K}]) \\
K &::= \texttt{enter}(R, S) = (P, C) \\
P &::= (T, [(S, [\overline{R}]); ...; (S, [\overline{R}])]) \\
T &::= \texttt{Excluded} \mid \texttt{Allowed} \mid \texttt{Required} \mid ...
\end{aligned}$$

Example Policy Types:

$$\frac{\exists R' \in \overline{R}_i.\ R' \text{ in } S_i \quad \text{for all} \quad i = 1..n}{(\texttt{Allowed}, [(S_1, [\overline{R}_1]); ...; (S_n, [\overline{R}_n])]) \quad \text{satisfied}}$$

$$\frac{\forall R' \in \overline{R}_i.\ R' \text{ in } S_i \quad \text{for all} \quad i = 1..n}{(\texttt{Required}, [(S_1, [\overline{R}_1]); ...; (S_n, [\overline{R}_n])]) \quad \text{satisfied}}$$

$$\frac{\forall R' \in \overline{R}_i.\ R' \text{ not in } S_i \quad \text{for all} \quad i = 1..n}{(\texttt{Excluded}, [(S_1, [\overline{R}_1]); ...; (S_n, [\overline{R}_n])]) \quad \text{satisfied}}$$

Fig. 1. The Role-Based Synchronization Model

Constraints and Families. We define a *constraint* on entering a state S by a role R, written `enter`(R, S), to be a policy P which regulates switching of the role to this state, paired with a logical condition C on all objects accessible by the role in state S. The role R can enter the state S if the policy P is satisfied *and* the condition C is true.

Policy P is expressed as a policy type T paired with a list L of *policy rules*, i.e. tuples $(S, [\overline{R}])$ of a state and a list of roles; the meaning of a policy rule will be explained below. The condition C has the form of a boolean function with no arguments which returns **true** if calling methods by role R will actually make sense if role R would now enter the state S, and **false** otherwise. What "makes sense" or not depends, of course, on the program semantics. Programmers define function C as part of the main code of the application, and so any variables visible in the scope of the function can be used to express the condition.

We define a *role family*, denoted F, to be a list of roles of a single synchronization problem, paired with a list of constraints, e.g. `Produce-Consumer-Family = ([Producer; Consumer], [enter(Producer, In); enter(Consumer, In)]).` Roles are globally unique.

Policy Types. We have identified two categories of policies: permission and denial (or refusal). Intuitively, a *permission policy* describes what must happen in order to *permit* a role to enter a state, while a *denial policy* describes what *forbids* a role to enter a state. Two example permission policy types and one denial policy type have been defined in the bottom of Figure 1:

Consider a constraint `enter`$(R, S) = (P, C)$, where $P = (T, L)$. We have the following policy types T:

- The $T =$ `Allowed` policy says that role R can enter state S only if for each tuple $(S_i, [\overline{R_i}])$ in L, at least one role in $\overline{R_i}$ is in state S_i; the empty list of roles means *any* role.
- The $T =$ `Required` policy says that role R can enter state S only if for each tuple $(S_i, [\overline{R_i}])$ in L, all roles $\overline{R_i}$ are in state S_i; the empty list of roles means *all* roles.
- The $T =$ `Excluded` policy says that role R can enter state S only if for each tuple $(S_i, [\overline{R_i}])$ in L, all roles in $\overline{R_i}$ are *not* in state S_i; the empty list of roles means *all* roles.

Note that satisfying policy `Excluded` means that policy `Allowed` is violated (and vice versa); similarly we can define the fourth denial policy by negation of `Required`.

Example Specifications of Constraints. For instance, consider a Producer-Consumer problem with the priority of producers. The policy of accessing a buffer by a consumer is such that the consumer is forbidden to enter a state that allows the buffer to be accessed, if there is already a producer accessing the buffer or there are some producers waiting in the queue to access it. Moreover, a consumer can enter this state only when the buffer is not empty.

We can specify the above constraint using an exclusion policy and a logical condition, as follows: `enter(Consumer, In) = (Excluded, [(In, [Producer]), (Wait, [Producer])], not (buffer.empty))`.

On the other hand, a constraint on entering the state `In` by producer is: `enter(Producer, In) = (Excluded, [(In, [Consumer])], not (buffer.full))`.

The simplicity of this formalism suggests that it can be indeed useful to encode synchronization constraints at the level of thread roles, instead of individual actions executed by the threads. The advantage is that the constraints can be expressed *declaratively*, as a set of policy rules. The rules are intuitively easier to understand than the low-level synchronization code, thus aiding design and proofs of correctness.

In the next section, we demonstrate how the separation of concerns defined by our model can be achieved in practice.

4 Example 'RBS' Package

We illustrate our approach using an example role-based synchronization package that we have implemented in the OCaml programming language [17] – an object-oriented variant of ML. OCaml has abstract types and pattern matching over types, which allowed us to use the syntax of the constraint language exactly as it has been defined in §3.3. We think however that the approach described in this paper can be easily adapted to any object-oriented language.

The package *Readers-Writers (RW)* implements a role family of the Readers-Writers synchronization problem [2], defined as follows. Two kinds of threads – readers and writers – share an object. To preclude interference between readers and writers, a writer must have exclusive access to the object. Assuming no writer is accessing the object, any number of readers may concurrently access the object. We assume that writers have priority over readers, i.e. new readers are delayed if a writer is writing or waiting, and a delayed reader is awakened only if no writer is waiting. To demonstrate expressiveness of our language, we have slightly extended the problem by adding a new role `Sysadmin` that represents a system administrator. The `Sysadmin` role is to periodically turn the computer – i.e., the shared object in our example – off and on for maintenance; turning the computer off prevents both writers and readers from executing their actions.

The example code below is taken almost verbatim from the source code in OCaml. However, those readers who are familiar with any object-oriented language should not find the programs too difficult to understand.

Application Program. Imagine a programmer who wants to use the RW package to implement an application that requires the Readers-Writers synchronization pattern. We assume that the programmer has already defined all application classes but no synchronization code is provided yet. Below is an example class:

```
class computer =
  object
    val mutable screen = 0
    val mutable on = true
    method read = screen
    method write x = screen <- x    (* Assign a new value to 'screen' *)
    method is_working = on
    method turn = on <- (not on)    (* Invert the boolean value 'on' *)
  end
```

The class implements a "computer object" that will be accessed by concurrent readers and writers. The class defines methods `read` and `write` for accessing the object's shared state `screen`, and methods `turn` and `is_working` that are used to turn the computer on/off and check its current status (on/off).

To use the RW synchronization package, the programmer has to create instances of *role objects*, where each role object (`r`, `w`, `s`) represents a particular role and "wraps" method calls on objects accessible by the role (in our case it is the computer object only):

```
let o = new computer;
let r = new rw_Reader o;
let w = new rw_Writer o;
let s = new rw_Sysadmin o;
```

Now we just require that concurrent threads (roles) in the application program do not access shared objects directly, but via the role objects defined above. For instance, a thread representing role `Reader` must read on the (screen of) computer using a method `r.read` instead of `o.read`, similarly for other roles.

Below is an example code which creates a thread of role Writer that writes in a loop a random integer (the notation w#write in OCaml corresponds to w.write in languages like Java and C++, i.e. "call a method write of object w").

```
let thread_writer_no1 = Thread.create (fun () ->
  while true do
    w#write (Random.int 10)
  done)
();
```

The invocation of the method by thread thread_writer_no1 is guaranteed to satisfy all the synchronization constraints imposed on this call, i.e. the call will be suspended (and the thread blocked) if the constraints cannot be satisfied, and automatically resumed after actions made by other threads will allow the constraints to be satisfied.

Synchronization Package. Now, let us explain how the RW package has been implemented. Below are declarations of roles and states:

```
(* Roles defined in Readers/Writers Package *)
type role = Reader | Writer | Sysadmin

(* States visited by roles *)
type action = Wait | In
```

Below is a class which implements "wrapping" of the shared object for the role Writer (the classes and methods have polymorphic types, expressed in OCaml with a type variable 'a):

```
class ['a] rw_Writer o =
  object
    (* Use object 'synchronizer' to evaluate constraints *)
    inherit ['a] rbs Writer synchronizer as sync

    method write (x : 'a) : unit =
      (* Message sent to synchronizer before the call *)
      sync#try_cond ([(Begin, Wait)], [(End, Wait);
                                       (Begin, In)]);
      (* Call the actual method (only one here) *)
      let result = o#write x in
      (* Message sent to synchronizer after the call *)
      sync#notify [(End, In)];
      (* Return any method result *)
      result
  end
```

Essentially, each call of an object method by a role is preceded and followed by a call to a synchronizer object (which has been bound to a name sync). Note that this resembles the AOP weaving principle. The synchronizer is the core part of the package – it implements an evaluation engine parameterized over constraint declarations (which will be explained below).

The effect of calling `sync#try_cond` above is that the engine will evaluate constraints declared for the role `Writer`. If all constraints are satisfied then a thread calling the `write` method will enter the state `In`, and the method will be invoked on the actual object `o`. Otherwise, the thread enters the state `Wait` (and it will be awaken once possible). After the call returns, the thread leaves state `In` and enters a default state `Idle`.

The method `sync#try_cond` takes as arguments two lists of events to be triggered by the role's threads, respectively at the beginning of a critical section, and inside the critical section, where an *event* is a tuple of time delimiters (`End` and `Begin`) and the states defined by a given design pattern (in our example, these are `Wait` and `In`). Triggering events increase and decrease some counter variables of the package, that are used by the concurrency controller to enforce user-defined synchronization policies. By using this mechanism, our approach allows logical conditions in declared policies to be parameterized by the number of threads that are in a given state (as illustrated in the following example).

Below are synchronization constraints for the `Reader` and `Writer` roles predefined by the package. (Of course, the application programmer who uses the package could always extend this set and express some specialized constraints.)

```
synchronizer#define_constraint
  Reader [{
    enter  = In;
    policy = (Excluded, [(In, [Writer; Sysadmin], []);
                         (Wait, [Writer], [])]);
    check  = [(fun () -> computer#is_working)]
  }];

synchronizer#define_constraint
  Writer [{
    enter  = In;
    policy = (Allowed, [(In, [], [])]);
    check  = [(fun () -> computer#is_working)]
  }];
```

Each constraint of a role consists of fields `enter`, `policy`, and `check`, which define respectively the state to enter by the role, the policy and the condition. Otherwise the syntax matches the one that we have used in §3.3, with a small extension.

To allow policies that require a quantitative information about threads, we have added a third field in each rule (tuple) of the policy list. This field may contain a number of threads of the corresponding role from which the rule begins to apply (the default value is 1), e.g. replacing `[]` by `[2]` in the `(Wait, [Writer], [])` policy (see the first constraint) would mean that the `Reader` will be excluded only if there are at least two `Writers` waiting.

5 Dynamic Policy Switching

Our constraint language could be used to extend the separation-of-concerns principle to systems that must be reconfigured without stopping them. For such systems, we consider a *dynamic switching* of synchronization constraints.

Each constraint is guarded by a boolean function (defined in the application program). Only a constraint whose guard has returned **true** is chosen for synchronization. If several guards have returned **true**, then the order of applying the corresponding policies depends on the RBS package. The RBS package implementation makes sure that the transition between different policy types is atomic and a new policy is correctly satisfied. We could easily extend the model in §3.3 accordingly, by requiring the second component of a constraint declaration to be a triple `(C,P,C')`, where `C` is a guard (we omitted it for clarity).

Below are two example constraints on a role `Reader` to enter a state `In`. The guard functions examine a content of a boolean variable `happy_hour`. The policy switching is "triggered" on-the-fly by the program execution state. Depending on the current value stored in `happy_hour` (returned with the ! construct that is used in OCaml to read mutable data) the synchronizer will either evaluate the first constraint, if `happy_hour` contains **true**, or the second one (otherwise).

```
synchronizer#define_constraint
  Reader [{
    enter  = In;
    guard  = [(fun () -> !happy_hour)];
    policy = (Excluded, [(In, [Writer; Sysdmin], [])]);
    check  = [(fun () -> computer#is_working)]
  };
  {
    enter  = In;
    guard  = [(fun () -> not !happy_hour)];
    policy = (Excluded, [(In, [Writer; Sysadmin], []);
                         (Wait, [Writer], [])]);
    check  = [(fun () -> computer#is_working)]
  }];
```

The policy in the first constraint removes the clause about the priority of writers from the second constraint (explained in §4) and so it equals the rights of readers and writers to access the computer's screen during a "happy hour".

6 Example Application

To facilitate experimentation, we have prototyped a small web access server that provides access to remote services to a large number of clients. The clients may have quite different workload profiles and priorities, simultaneous access of some client types can be forbidden. Accessing a distant resource such as a remote web service or database can be both expensive and time consuming. Therefore such accesses should be well controlled and fine-tuned, considering different client

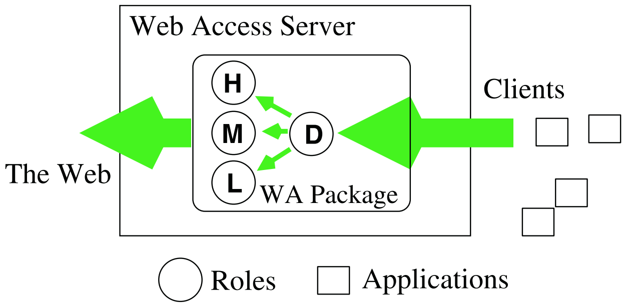


Fig. 2. The Web Access Server

profiles, and external factors such as the current traffic, the time of the day, etc. We also require that:

- to meet any future expectations without reverse-engineering the server's components, the code must be flexible and easy to maintain and modify;
- it should be possible to change the control policy dynamically.

Our solution is to use a generic *Web Access (WA)* synchronization package to implement a component that is used to access the web. We present a schema showing the main components of our implementation in Figure 2. Roles predefined by the WA package are the `Low` (`L`), `Medium` (`M`), and `High` (`H`) *access priority* roles and the `Dispatcher` role `D`.

The access priority roles define different policies of accessing the web based on the client's priority (low, medium, high). The general policy constraint *between* access priority roles is that roles `M` and `L` are blocked if role `H` is accessing the web, while `L` is blocked if either `M` or `L` or both are accessing the web. Internally, each priority role may declare several variants of the policy which may change dynamically, e.g. once moving from the peak time of the day to the evening, or when some traffic fluctuation has been observed.

The `Dispatcher` role recognizes a client contacting the server and dynamically selects a suitable access priority role, based on the dynamically calculated access rate of the application and the recent client's history (e.g. exceeding the time of calls beyond the agreed time threshold during the previous call, calling the server too often etc.). When the client application is calling the server for the first time, the server provides it with a fresh ID that must be presented in the following calls.

The WA design pattern package enables rapid prototyping of synchronization constraints. It is easy to customize the system by simply extending a list of policies that can be switched to at runtime.

7 Conclusions and Future Work

We have proposed a constraint language for separation of synchronization concerns from functional ones while keeping visible the role-oriented aspects of synchronization problems. Contrary to similar implementations, our solution allows the programmer to declare complex constraints, which can inspect (at runtime) the dynamic content of program variables and data.

We have demonstrated that it is possible to implement our approach in a general purpose programming language (OCaml) without using external tools. We can achieve this by analyzing and expressing common concurrency problems in the form of a design pattern; the patterns are then encoded in the host language as synchronization packages. We have implemented two example synchronization packages, where each package defines a given set of roles. One package has been used to implement a small application, which can dynamically switch between declared quality-of-service policies for accessing a web service.

We believe that our approach is language independent. However, the type system of OCaml certainly helped in representing policies in a way that resembles our formal specification of the constraint language in §3. In the future work, we would like to experiment with dynamically type-checked scripting languages like Python, or extensions of the OCaml language, such as Acute. Their dynamic binding features could make it possible to "plug-in" code of synchronization constraints at runtime. It may be also worthwhile to investigate the possibility of extending Java-like languages with the RBS approach, and using extensible compilers for the implementation of the constraint language.

We hope that our demonstration of declarative synchronization can be instructive, especially in the context of adaptive, non-stop systems, which may occasionally need to change their *modus operandi*.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proc. ECOOP '02 (16th European Conference on Object-Oriented Programming)*, LNCS 2374, June 2002.
2. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, Oct. 1971.
3. F. L. Fessant and L. Maranget. Compiling join-patterns. In *Proc. HLCL '98 (Workshop on High-Level Concurrent Languages)*, 1998.
4. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. CONCUR '96 (7th Conference on Concurrency Theory)*, LNCS 1119, Aug. 1996.
5. S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. ECOOP '93 (7th European Conference on Object-Oriented Programming)*, LNCS 627, July 1993.
6. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison Wesley, 2000.
7. M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.
8. W. Hursch and C. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Feb. 1995.
9. R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. ECOOP '03 (17th European Conference on Object-Oriented Programming)*, LNCS 2743, July 2003.

10. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL '96 (23rd ACM Symposium on Principles of Programming Languages)*, Jan. 1996.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
12. C. V. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, Dec. 1997 (1998).
13. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
14. S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. Middleware 2003*, LNCS 2672, 2003.
15. G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *Proc. ACM Java Grande/ISCOPE Conference*, Nov. 2002.
16. G. Milicia and V. Sassone. Jeeg: Temporal constraints for the synchronization of concurrent objects. Technical Report RS-03-6, BRICS, Feb. 2003.
17. Objective Caml. *http://caml.inria.fr.*
18. P. Panangaden and J. Reppy. The Essence of Concurrent ML. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation, and Application*, pages 5–29. Springer, 1997.
19. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.
20. Python. *http://www.python.org/.*
21. R. Ramirez and A. E. Santosa. Declarative concurrency in Java. In *Proc. HIPS 2000 (5th IPDPS Workshop on High-Level Parallel Programming Models and Supportive Environments)*, May 2000.
22. R. Ramirez, A. E. Santosa, and R. H. C. Yap. Concurrent programming made easy. In *Proc. ICECCS (6th IEEE International Conference on Engineering of Complex Computer Systems)*, Sept. 2000.
23. S. Ren and G. A. Agha. RTsynchronizer: Language support for real-time specifications in distributed systems. In *Proc. ACM Workshop on Languages, Compilers, & Tools for Real-Time Systems*, 1995.
24. P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, Oct. 2004. Also published as INRIA RR-5329.
25. P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, LNCS 1686, pages 1–31, 1999.
26. B. Stroustrup. *The Design and Evolution of C++.* Addison Wesley, 1994.
27. W3C. Web Services Architecture. *http://www.w3.org/TR/ws-arch/.*
28. P. T. Wojciechowski. Concurrency combinators for declarative synchronization. In *Proc. APLAS 2004 (2nd Asian Symposium on Programming Languages and Systems)*, volume 3302 of *LNCS*. Springer, Nov. 2004.
29. P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency. The Computer Society's Systems Magazine*, 8(2):42–52, April-June 2000.

Towards a More Practical Hybrid Probabilistic Logic Programming Framework

Emad Saad and Enrico Pontelli

Department of Computer Science
New Mexico State University
{emsaad,epontell}@cs.nmsu.edu

Abstract. The hybrid probabilistic programs framework [1] allows the user to explicitly encode both logical and statistical knowledge available about the dependency among the events in the program. In this paper, we extend the language of hybrid probabilistic programs by allowing *disjunctive composition functions* to be associated with heads of clauses, and we modify its semantics to make it more suitable to encode real-world applications. The new semantics is a natural extension of standard logic programming semantics. The new semantics of hybrid probabilistic programs also subsumes the implication-based probabilistic approach proposed by Lakshmanan and Sadri [12]. We provide also a sound and complete algorithm to compute the least fixpoint of hybrid probabilistic programs with annotated atomic formulas as rule heads.

1 Introduction

Reasoning with uncertain knowledge is an important issue in most real-world applications, including those in AI domains. The literature is rich of proposals for extensions of the *Logic Programming (LP)* framework with different notions of uncertainty (e.g., [1, 4–6, 8, 9, 11–14, 16, 18–20, 22, 24]), providing the ability to represent both logical as well as probabilistic knowledge about a domain. The semantics of such frameworks provide ways to systematically derive logical conclusions along with their associated probabilistic properties. The differences between these frameworks rely mainly on the underlying formalism of uncertainty adopted and on the way certainty values are associated to the rules and facts of the logic programs. These formalisms include *fuzzy set theory* [22, 24], *possibilistic logic* [4], *hybrid formalisms* [11, 13], *multi-valued logic* [5, 6, 8, 9], and formalisms based on *probability theory* [1, 12, 18–20].

In the context of frameworks based on probability theory, Dekhtyar and Subrahmanian [1] proposed the notion of *Hybrid Probabilistic Programs (HPP)*. Hybrid probabilistic programs are built upon the idea of *annotated logic programs*, originally introduced in [23] and extensively studied in [8, 9, 15, 17]. Uncertainty is encoded in the form of *probability intervals*, representing *constraints* on a set of possible probability distributions associated to the elements of the Herbrand Universe. The key idea in hybrid probabilistic programs is to enable the user to *explicitly* encode his/her knowledge about the type of dependencies existing between the probabilistic events being described by the programs. HPP is a generalization of the *probabilistic annotated logic programming approach* – originally proposed in [18] and further extended in [19] – where different probabilistic strategies are allowed instead of a fixed probabilistic strategy [18, 19]. Observe that these approaches differ from the probabilistic logic programming schemes (e.g., [7, 21]) where a single probability distribution over some logical entities (e.g., possible worlds) is defined (see [25] for a general discussion of these latter schemes).

The aim of probabilistic logic programming in general, and of the HPP framework in particular, is to allow reasoning and decision making with uncertain knowledge; in the context of the probabilistic framework, uncertainty is represented using probability theory. Nevertheless, to achieve this goal and to be able to perform reasoning tasks in a more practical way and with cognitively meaningful results, a more sophisticated syntax and semantics for HPP than the one previously proposed [1] is required. Let us consider the following example.

Example 1. Consider the following robot planning task, adapted from [10]. A robot's grasping operation is not always successful, especially if the robot's gripper is wet. The robot is able to grasp a block with a probability 0.95 after executing the *pickup* action in the state of the world in which the gripper is dry, while the probability decreases to 0.5 in the state of the world where the gripper is wet. Assume that initially the block is not held, and the gripper is dry with probability 0.7. There are two world states in which the robot can hold the block after executing the action *pickup*. The state in which the gripper is dry and the robot holds the block has probability $0.7 \times 0.95 = 0.665$. If the gripper is initially wet, then the second resulting state where the robot holds the block has probability $0.3 \times 0.5 = 0.15$. Hence, the robot is successfully holding the block with probability $0.665 + 0.15 = 0.815$. This planning domain problem can be represented in the HPP framework as follows:

$$\begin{array}{lcl}
holdBlock : [0.95 \times V, 0.95 \times V] & \leftarrow & pickup : [1,1], gripperDry : [V,V] \\
holdBlock : [0.5 \times V, 0.5 \times V] & \leftarrow & pickup : [1,1], not\_gripperDry : [V,V] \\
not\_gripperDry : [1-V, 1-V] & \leftarrow & gripperDry : [V,V] \\
pickup : [1,1] & \leftarrow & \\
gripperDry : [0.7, 0.7] & \leftarrow &
\end{array}$$

Example 1 demonstrates a shortcoming in the semantical characterization of HPP. In spite of the intuitive correctness and simplicity of the above encoding, the semantics of HPP fails to assign the correct probability interval – $[0.815, 0.815]$ – to the formula *holdBlock*. This can be shown by computing the least fixpoint of this program. The least fixpoint assigns $[1, 1]$ to *pickup*, $[0.7, 0.7]$ to *gripperDry*, $[0.3, 0.3]$ to *not_gripperDry*, and $\emptyset$ to *holdBlok*. $\emptyset$ is assigned to *holdBlock* because, by applying the first rule in the program, *holdBlock* is derived with the probability interval $[0.665, 0.665]$, and by applying the second rule *holdBlock* is derived with the interval $[0.15, 0.15]$. Therefore, according to HPP's semantics, *holdBlock* is assigned the probability interval: $\cap\{[0.665, 0.665], [0.15, 0.15]\} = \emptyset$. Similar situations occur frequently in HPP encodings.

This shortcoming arises, mainly, from the chosen ordering between probability intervals – i.e., set inclusion order – employed in the HPP semantics [1]. Under this ordering, given the probability intervals $[a_1, a_2], [b_1, b_2] \subseteq [0, 1]$ for a certain event e, $[a_1, a_2] \leq [b_1, b_2]$ iff $[b_1, b_2] \subseteq [a_1, a_2]$. This set inclusion order is known as the *knowledge order* [18–20, 1]. This means that $[b_1, b_2]$ provides a more precise probabilistic knowledge about the probability of e than $[a_1, a_2]$. However, when reasoning is performed to make decisions, the outcome usually does not depend on how more knowledgeable we are about the probability of the various events, but it depends on how likely these events are. Therefore, it is more reasonable to make use of an ordering which describes the likelyhood of a certain event e to guide the reasoning task. This intuition can be captured by employing the natural ordering $\leq_t$ – called *truth order* – described in [12, 1]. The truth order $\leq_t$ asserts that if $[a_1, a_2], [b_1, b_2] \subseteq [0, 1]$ are two probability intervals for the events e_1 and e_2 respectively, then $[a_1, a_2] \leq_t [b_1, b_2]$ iff $a_1 \leq b_1$ and $a_2 \leq b_2$. Hence, the event e_2 is more likely to occur than the event e_1.

The problem of having unintuitive or incorrect probability intervals in the above example could be avoided in Dekhtyar and Subrahmanian's approach [1], by modifying the HPP encoding of the problem. This is accomplished by disallowing *holdBlock* to appear as head of more than one clause – thus, removing the need to perform intersection between intervals – and by using more complex annotation functions. Consider the following encoding of Example 1 using the approach in [1].

Example 2.

$$\begin{aligned}
&\mathit{holdBlock} : [0.95 \times V + 0.5 \times V_1, 0.95 \times V + 0.5 \times V_1] \\
&\qquad\qquad \leftarrow \mathit{pickup} : [1, 1], \mathit{gripperDry} : [V, V], \\
&\qquad\qquad\quad \mathit{not\_gripperDry} : [V_1, V_1]. \\
&\mathit{not\_gripperDry} : [1 - V, 1 - V] \leftarrow \mathit{gripperDry} : [V, V]. \\
&\mathit{pickup} : [1, 1] \leftarrow . \qquad\qquad\qquad \mathit{gripperDry} : [0.7, 0.7] \leftarrow .
\end{aligned}$$

The above encoding ensures the correct solution of the problem described in Example 1 under the semantics of HPP [1]. Nevertheless, the encoding is fairly unintuitive, since two clauses are combined into one – rather complex – clause, with a fairly complex annotation function. This encoding strategy is not feasible in general; in presence of many alternative ways to determine the probability interval of an event, we would obtain very complex clauses and annotation functions. Providing a different semantics for HPP to sustain simple and intuitive encodings is considered essential to make the framework practical.

In this work we propose to develop an alternative semantics for HPP to accommodate for the truth order instead of the knowledge order. The introduction of a different ordering between probability intervals for events requires significant changes at the syntactic and semantics level. The changes include the use of disjunctive composition functions, similar to those used in [12], for combing the probability intervals derived from different clauses and associated to the same events.

The main contribution of this work is the definition of a new syntax and semantics for the HPP framework presented in [1], to appropriately reason probabilistically about real-world applications by employing the truth order. The new framework provides more intuitive and accurate probability intervals. We show that problems such as the one described in Example 1 are properly addressed in the new framework. The new semantics is a natural extension of standard logic programming semantics. We also show that the new hybrid probabilistic programming framework can be easily generalized to subsume Lakshmanan and Sadri's [12] approach for probabilistic logic programming – a feature which was not available under the previous semantics for HPP. This shows, in general, any appropriately defined annotation based logic programming framework can *subsumes* any implication based logic programming framework. We provide sound and complete algorithm to compute the least fixpoint of hybrid probabilistic programs with annotated atomic formulas as heads of their rules.

2 Preliminaries

In this section we review the foundations of Hybrid Probabilistic Programs (HPPs) [1].

2.1 Probabilistic Strategies

Let $C[0,1]$ denote the set of all closed intervals in $[0,1]$. In the context of HPP, probabilities are assigned to primitive events (atoms) and compound events (conjunctions and disjunctions of atoms) and encoded as intervals in $C[0,1]$. The type of dependency among the simple events within a compound event are determined according to *probabilistic strategies* selected by the programmer. This section introduces the notion of probabilistic strategy and presents some of its properties [1] – opportunely modified to accommodate for the truth order between intervals.

Definition 1 (Truth Order). *Let* $[a_1, b_1], [a_2, b_2] \in C[0,1]$. *Then* $[a_1, b_1] \leq_t [a_2, b_2]$ *iff* $a_1 \leq a_2$ *and* $b_1 \leq b_2$.

Intuitively, an interval is preferred if it provides greater evidence of truth.

Lemma 1. *The set* $C[0,1]$ *and the relation* $\leq_t$ *form a complete lattice. The join* ($\oplus_t$) *is defined as* $[a_1, a_2] \oplus_t [b_1, b_2] = [max\{a_1, b_1\}, max\{a_2, b_2\}]$ *and the meet* ($\otimes_t$) *is defined as* $[a_1, a_2] \otimes_t [b_1, b_2] = [min\{a_1, b_1\}, min\{a_2, b_2\}]$.

A *probabilistic strategy* is composed of two functions; the *composition function* is used to combine the probability intervals of two events, while the *maximal interval function* is used to provide an estimate of the probability of a primitive event e given the probability of a compound event containing e.

Definition 2. *A probabilistic strategy (p-strategy)* ρ *is a pair of functions* $\langle c, md \rangle$. c *is a* probabilistic composition function $c : C[0,1] \times C[0,1] \to C[0,1]$ *that satisfies the following properties:*

a. Commutativity: $c([a_1, b_1], [a_2, b_2]) = c([a_2, b_2], [a_1, b_1])$
b. Associativity: $c(c([a_1, b_1], [a_2, b_2]), [a_3, b_3]) = c([a_1, b_1], c([a_2, b_2], [a_3, b_3]))$

c. *Monotonicity: if* $[a_1, b_1] \leq_t [a_3, b_3]$ *then* $c([a_1, b_1], [a_2, b_2]) \leq_t c([a_3, b_3], [a_2, b_2])$

d. *Separation: there exist two functions* c_1 *and* c_2 *such that*

$$c([a_1, b_1], [a_2, b_2]) = (c_1([a_1, a_2]), c_2([b_1, b_2]))$$

$md : C[0, 1] \rightarrow C[0, 1]$ *is called the* maximal interval function.

Given the probability range of a complex event, the maximal interval function md returns the best estimate of the probability range of a primitive event. The composition function c returns the probability range of a conjunction or disjunction of two or more events. The above definition is different from the one adopted in [1] – being based on the truth order instead of knowledge order to establish monotonicity. Since the composition functions in the various p-strategies are commutative and associative, the order in which a composition function is applied to its arguments is irrelevant. For the sake of convenience, we introduce the following simplified notation:

Definition 3. *Let* $M = \{[a_1, b_1], \ldots, [a_n, b_n]\}$ *be a set of distinct probability intervals. The notation* cM *is defined as follows:*

- $cM = [0, 0]$ *if* $n = 0$.
- $cM = [a_1, b_1]$ *if* $n = 1$.
- $cM = c([a_1, b_1], c([a_2, b_2], \ldots, c([a_{n-1}, b_{n-1}], [a_n, b_n]) \ldots)$ *if* $n \geq 2$.

According to the type of combination among events, p-strategies are classified into *conjunctive* p-strategies and *disjunctive* p-strategies [1]. Conjunctive strategies are employed to compose events belonging to conjunctive formulae, while disjunctive strategies are employed with disjunctions of events.

Definition 4. *Let* $\langle c, md \rangle$ *be a p-strategy.*

- $\langle c, md \rangle$ *is a* conjunctive p-strategy *if the following properties hold:*
  a. *Bottomline:* $c([a_1, b_1], [a_2, b_2]) \leq_t [\min(a_1, a_2), \min(b_1, b_2)]$
  b. *Identity:* $c([a, b], [1, 1]) = [a, b]$
  c. *Annihilator:* $c([a, b], [0, 0]) = [0, 0]$
  d. *Maximal interval:* $md([a, b]) = [a, 1]$
- $\langle c, md \rangle$ *is a* disjunctive p-strategy *if the following properties hold:*
  a. *Bottomline:* $[\max(a_1, a_2), \max(b_1, b_2)] \leq_t c([a_1, b_1], [a_2, b_2])$
  b. *Identity:* $c([a, b], [0, 0]) = [a, b]$
  c. *Annihilator:* $c([a, b], [1, 1]) = [1, 1]$
  d. *Maximal interval:* $md([a, b]) = [0, b]$.

Example 3. The following are examples of p-strategies [1]:

- Independence p-strategy (in):
  - Conjunctive (inc): $c_{inc}([a_1, b_1], [a_2, b_2]) = [a_1 a_2, b_1 b_2]$.
  - Disjunctive (ind): $c_{ind}([a_1, b_1], [a_2, b_2]) = [a_1 + a_2 - a_1 a_2, b_1 + b_2 - b_1 b_2]$.

- Ignorance p-strategy (ig):
  - Conjunctive (igc): $c_{igc}([a_1, b_1], [a_2, b_2]) = [\max(0, a_1 + a_2 - 1), \min(b_1, b_2)]$.
  - Disjunctive (igd): $c_{igd}([a_1, b_1], [a_2, b_2]) = [\max(a_1, a_2), \min(1, b_1 + b_2)]$.
- Positive correlation p-strategy (pc):
  - Conjunctive (pcc): $c_{pcc}([a_1, b_1], [a_2, b_2]) = [\min(a_1, a_2), \min(b_1, b_2)]$.
  - Disjunctive (pcd): $c_{pcd}([a_1, b_1], [a_2, b_2]) = [\max(a_1, a_2), \max(b_1, b_2)]$.
- Negative correlation p-strategy (nc):
  - Disjunctive (ncd): $c_{ncd}([a_1, b_1], [a_2, b_2]) = [\min(1, a_1 + a_2), \min(1, b_1 + b_2)]$.

2.2 Annotations

Let L be an arbitrary first-order language with finitely many predicate symbols, constants, and infinitely many variables. Let $S = S_{conj} \cup S_{disj}$ be an arbitrary set of p-strategies, where S_{conj} is the set of all conjunctive p-strategies in S and S_{disj} is the set of all disjunctive p-strategies in S. The notions of term, atom, and literal are defined in the usual way. The Herbrand base of L is denoted by B_L.

Definition 5. *An annotation function of arity n is a computable total function* $f : [0, 1]^n \to [0, 1]$.

Definition 6 (Annotations). *An* annotation item *is*

- *A constant in* $[0, 1]$*, or*
- *A variable ranging over* $[0, 1]$ *– called* annotation variable*, or*
- $f(\alpha_1, \ldots, \alpha_n)$ *where* f *is an annotation function of arity* n *and* $\alpha_1, \ldots, \alpha_n$ *are annotation items.*

Annotation *are expressions of the form* $[\alpha_1, \alpha_2]$*, where* α_1, α_2 *are annotation items.*

Example 4. $[0.91, 1], [V_1 * V_2, V_1]$, and $[0, V]$ are annotations where V, V_1, V_2 are annotation variables.

The following definition introduces the notion of *hybrid basic formula*, which describes how p-strategies are associated to conjunctions and disjunctions of atoms.

Definition 7 (Hybrid Basic Formula). *Let us consider a collection of atoms* $A_1, \ldots, A_n$*, a conjunctive p-strategy* ρ*, and a disjunctive p-strategy* ρ'*. Then* $A_1 \wedge_\rho \ldots \wedge_\rho A_n$ *and* $A_1 \vee_{\rho'} \ldots \vee_{\rho'} A_n$ *are called* hybrid basic formulae. $bf_S(B_L)$ *is the set of all ground hybrid basic formulae formed using distinct atoms from* B_L *and p-strategies from* S.

Finally, the following definition introduces a simplified notation used to describe generic partitioning of hybrid basic formulae.

Definition 8. *Let* $F, G, H \in bf_S(B_L)$*, and let* $F = F_1 *_\rho \ldots *_\rho F_n$*,* $G = G_1 *_\rho \ldots *_\rho G_k$*, and* $H = H_1 *_\rho \ldots *_\rho H_m$*, where* $* \in \{\wedge, \vee\}$ *and* $\rho \in S$*. Then* $G \oplus_\rho H = F$ *iff* $k > 0$*,* $m > 0$*,* $\{G_1, \ldots, G_k\} \cup \{H_1, \ldots, H_m\} = \{F_1, \ldots, F_n\}$*, and* $\{G_1, \ldots, G_k\} \cap \{H_1, \ldots, H_m\} = \emptyset$.

3 A New Semantics for Hybrid Probabilistic Programs

In the new semantics, the composition functions for the disjunctive p-strategies are used to combine the probability intervals of the same hybrid basic formula derived from different hybrid probabilistic clauses. For example, if a hybrid probabilistic program consists of the clauses

$$\begin{array}{ll} a:[0.5,0.6] \leftarrow b:[0.7,0.7] & \quad a:[0.4,0.7] \leftarrow c:[0.5,0.8] \\ b:[0.7,0.7] \leftarrow & \quad c:[0.5,0.8] \leftarrow \end{array}$$

and it is known that deriving a from the first clause with probability $[0.5,0.6]$ is positively correlated to deriving a with probability $[0.4,0.7]$ from the second clause, then a can be concluded with probability $c_{pcd}([0.5,0.6],[0.4,0.7]) = [\max(0.5,0.4),\max(0.6,0.7)] = [0.5,0.7]$ (see also Example 3). A similar idea has been used in [12] in the context of an implication-based approach to probabilistic LP.

3.1 Syntax

In this subsection, we provide a new syntax for hybrid probabilistic programs. This is a modification of the notion of hybrid probabilistic programs of [1], by allowing the user to encode his/her knowledge about how to combine the probability intervals for a hybrid formula derived from *different* clauses.

Definition 9 ([1]). *A hybrid probabilistic rule (h-rule) is an expression of the form*

$$F:\mu \leftarrow F_1:\mu_1,\ldots,F_n:\mu_n$$

where $F, F_1, \ldots, F_n$ are hybrid basic formulae and $\mu, \mu_1, \ldots, \mu_n$ are annotations. $F:\mu$ is called the head *of the h-rule, while $(F_1:\mu_1,\ldots,F_n:\mu_n)$ is its* body.

Definition 10. *A hybrid probabilistic program (h-program) over S is a pair $P = \langle R, \tau\rangle$ where R is a finite set of h-rules involving only p-strategies from S, and τ is a mapping $\tau : bfs(B_L) \rightarrow S_{disj}$.*

The mapping τ in Definition 10 associates to each ground hybrid basic formula F a disjunctive p-strategy, that will be used to combine the intervals obtained from different rules that have F as head.

Definition 11. *Let $P = \langle R, \tau\rangle$ be an h-program. P is said to be a ground h-program iff all h-rules in R do not contain neither variables nor annotation variables.*

Example 5. The following is a typical h-program $P = \langle R, \tau\rangle$ where R

$$\begin{array}{ll} a:[0.5,0.6] \leftarrow b:[0.7,0.7] & \quad a:[0.4,0.7] \leftarrow c:[0.5,0.8] \\ b:[0.7,0.7] \leftarrow & \quad c:[0.5,0.8] \leftarrow \end{array}$$

and $\tau(a) = ncd, \tau(b) = \tau(c) = \pi$, where π is an arbitrary disjunctive p-strategy.

3.2 Declarative Semantics

The concept of *probabilistic model* is based on the notion of hybrid formula function.

Definition 12. *A hybrid formula function h is a mapping $h : bf_S(B_L) \to C[0,1]$ which satisfies the following conditions:*

1. Commutativity: $h(F) = h(G_1 *_\rho G_2)$ *if* $F = G_1 \oplus_\rho G_2$.
2. Composition: $c_\rho(h(G_1), h(G_2)) \leq_t h(F)$ *if* $F = G_1 \oplus_\rho G_2$.
3. Decomposition: *For any hybrid basic formula F and for all $\rho \in S$ and $G \in bf_S(B_L)$, we have that $md_\rho(h(F *_\rho G)) \leq_t h(F)$.*

We denote with HFF the set of all hybrid formula functions for a given HPP language.

Let us extend the notion of truth order to the case of hybrid formula functions and investigate some properties of the resulting partial order relation.

Definition 13. *If h_1 and h_2 are hybrid formula functions, then*

$$(h_1 \leq_t h_2) \Leftrightarrow (\forall F \in bf_S(B_L) : h_1(F) \leq_t h_2(F))$$

Lemma 2. *The set of all hybrid formula functions HFF and the truth order $\leq_t$ form a complete lattice. The meet $\otimes_t$ and the join $\oplus_t$ operations with respect to $\leq_t$ are defined as follows: for all $F \in bf_S(B_L)$*

$$(h_1 \otimes_t h_2)(F) = h_1(F) \otimes_t h_2(F) \qquad (h_1 \oplus_t h_2)(F) = h_1(F) \oplus_t h_2(F)$$

The top element of the lattice $\langle HPP, \leq_t \rangle$ is the mapping $bf_S(B_L) \to [1,1]$ and the bottom element is the mapping $bf_S(B_L) \to [0,0]$.

Let us now describe how formula functions are employed as models of hybrid probabilistic programs.

Definition 14 (Formula Satisfaction). *Let h be a hybrid formula function, $F \in bf_S(B_L)$, and $\mu \in C[0,1]$. Then*

- *h is a p-model of $(F : \mu)$ (denoted by $h \models F : \mu$) iff $\mu \leq_t h(F)$.*
- *h is a p-model of $F_1 : \mu_1, \ldots, F_n : \mu_n$ (denoted by $h \models F_1 : \mu_1, \ldots, F_n : \mu_n$) iff for all $1 \leq i \leq n$, h is a p-model of $F_i : \mu_i$.*
- *h is a p-model of the h-rule $F : \mu \leftarrow Body$ iff h is a p-model of $F : \mu$ or h is not a p-model of $Body$.*

Since probability intervals of the same hybrid basic formula F derived from different h-rules are combined together to strengthen the overall probability interval of F, more conditions need to be imposed to define the concept of p-models of h-programs. The following definitions are needed. The intermediate operator S_P makes use of the disjunctive p-strategy $\tau(F)$ to combine the probability intervals for the hybrid basic formula F *directly* derived from different h-rules.

Definition 15. *Let $P = \langle R, \tau \rangle$ be a ground h-program and h a hybrid formula function. The intermediate operator S_P is the mapping $S_P : HFF \to HFF$ such that $S_P(h)(F) = c_{\tau(F)}\, M$ where $M = \{\mu | F : \mu \leftarrow Body \in R \wedge h \models Body\}$.*

Lemma 3. *The S_P operator is monotonic.*

The following example shows that S_P is in general not continuous.

Example 6. Let P is an h-program of the form

$$q : [0.3, 0.3] \leftarrow p : [1, 1]$$

Let $h_j(p) = [1 - (\frac{1}{2})^j, 1 - (\frac{1}{2})^j]$ and $h_j(q) = [0,0]$ where $0 \leq j \leq \infty$ be hybrid formulae functions for P. $lub(h_j)(p) = [1, 1]$ and hence, $S_P(lub(h_j))(q) = [0.3, 0.3]$. However, for all $j, S_P(h_j)(q) = [0, 0]$ and hence $lub(S_P(h_j)(q)) = [0, 0]$.

Observe that if $M = \emptyset$ – i.e., there are no active h-rules in P having F as head – then $S_P(h)(F) = [0, 0]$. However, it is possible to have h-rules in P where the hybrid basic formulae in their heads contains F as a constituent, e.g., $F *_\rho G$. Therefore, the probability interval associated to $F *_\rho G$ (see definition 16) can be used to extract the probability interval of F by employing the maximal interval function, md_ρ, which in turn will be used to strengthen the overall probability interval of F (Definition 17). An additional contribution to the probability of F can be derived when F is not atomic and the information about F's probability can be inductively obtained using h for every pair G, H of formulae such that $G \oplus_\rho H = F$ and using c_ρ to combine these intervals.

Definition 16. *Let $P = \langle R, \tau \rangle$ be a ground h-program and h be a hybrid formula function. Then for each $F \in bf_S(B_L)$:*

- *If F is atomic then*
  $M_1^h(F) = \{\langle \mu, \rho \rangle | (F *_\rho G) : \mu \leftarrow Body \in R, * \in \{\vee, \wedge\}, \rho \in S, h \models Body\}$
- *$F = F_1 *_\rho \ldots *_\rho F_n$ is not atomic then:*

$$M_2^h(F) = \left\{ \begin{array}{r} \langle \mu, \rho \rangle \mid (D_1 *_\rho \ldots *_\rho D_k) : \mu \leftarrow Body \in R, h \models Body, \\ \{F_1, \ldots, F_n\} \subset \{D_1, \ldots, D_k\}, n < k \end{array} \right\}$$

Definition 17. *Let $P = \langle R, \tau \rangle$ be a ground h-program and h be a hybrid formula function. Then, h is a* p-model *of P iff h is a p-model of every h-rule in P and for all $F \in bf_S(B_L)$ the following conditions are satisfied:*

- *If F is atomic then $c_{\tau(F)}(c_{\tau(F)}\{md_\rho(\mu) | \langle \mu, \rho \rangle \in M_1^h(F)\}, S_P(h)(F)) \leq_t h(F)$*
- *If $F = F_1 *_\rho \ldots *_\rho F_n$ is not atomic then:*

$$c_{\tau(F)} \left(S_P(h)(F), c_{\tau(F)} \left(\begin{array}{l} \{c_\rho(h(G), h(H)) | G \oplus_\rho H = F\} \cup \\ \{md_\rho(\mu) | \langle \mu, \rho \rangle \in M_2^h(F)\} \end{array} \right) \right) \leq_t h(F)$$

The following properties allow us to provide a simple definition of the *least p-model* of a hybrid probabilistic program.

Proposition 1. *Let P be a ground h-program and let h_1, h_2 be two p-models of P. Then, $h_1 \otimes_t h_2$ is also a p-model of P.*

Corollary 1 (The Least Model h_P). *Let P be a ground h-program and let H_P be the set of all p-models of P. Then, $h_P = \otimes_t\{h | h \in H_P\}$ is the least p-model of P.*

3.3 Fixpoint Semantics

Associated to each h-program, P, is a *fixpoint operator* T_P, that takes a hybrid formula function as an argument and returns a hybrid formula function.

Definition 18. *Let $P = \langle R, \tau \rangle$ be a ground h-program and h a hybrid formula function. The fixpoint operator T_P is a mapping $T_P : HFF \rightarrow HFF$ defined as follows:*

- *If F is atomic then $T_P(h)(F) = c_{\tau(F)}(c_{\tau(F)}\{md_\rho(\mu) | \langle \mu, \rho \rangle \in M_1^h(F)\}, S_P(h)(F))$*
- *If $F = F_1 *_\rho \ldots *_\rho F_n$ is not atomic then*

$$T_P(h)(F) = c_{\tau(F)} \left(\begin{array}{l} S_P(h)(F), \\ c_{\tau(F)} \left(\begin{array}{r} \{c_\rho(T_P(h)(G), T_P(h)(H)) | G \oplus_\rho H = F\} \\ \cup \{md_\rho(\mu) | \langle \mu, \rho \rangle \in M_2^h(F)\} \end{array} \right) \end{array} \right)$$

The intuition in the above definition is that, in order to compute the least fixpoint of an h-program P, for any $F \in bf_S(B_L)$, we need to consider: *(i)* the h-rules having F in their heads, *(ii)* the h-rules where F appears as a component of the head, and *(iii)* the h-rules whose heads contain constituents of F. Let us now proceed in the construction of the least p-model as repeated iteration of the fixpoint operator T_P:

Definition 19. *Let P be a ground h-program. Then:*

- *$T_P \uparrow 0 = h_\perp$ where $h_\perp$ is the mapping $h_\perp : bf_S(B_L) \rightarrow [0, 0]$*
- *$T_P \uparrow \alpha = T_P(T_P \uparrow (\alpha - 1))$ if α is a successor ordinal.*
- *$T_P \uparrow \lambda = \oplus_t\{T_P \uparrow \alpha | \alpha < \lambda\}$ if λ is a limit ordinal.*

Lemma 4. *The T_P operator is monotonic.*

The properties of the T_P operator guarantee the existence of a least fixpoint and its correspondence to the least model of an h-program.

Proposition 2. *Let P be an h-program and h be hybrid formula function. Then h is a model of P iff $T_P(h) \leq_t h$.*

Theorem 1. *Let P be an h-program. Then, $h_P = lfp(T_P)$.*

The T_P operator for annotated logic programming in general, and probabilistic annotated logic programming in particular, is not continuous [1,18,19, 23]. The only case when T_P is continuous is whenever the annotations appear in the body of every h-rule are *variable annotations* [9]. This extends to the T_P operator defined in this paper.

To illustrate how the new semantics is more sophisticated and intuitive, let us reconsider the robot gripper planning example, introduced Section 1, which can be encoded as the following h-program $P = \langle R, \tau \rangle$ where R is

$$\begin{array}{ll} holdBlock : [0.95 \times V, 0.95 \times V] & \leftarrow pickup : [1, 1], gripperDry : [V, V]. \\ holdBlock : [0.5 \times V, 0.5 \times V] & \leftarrow pickup : [1, 1], not\_gripperDry : [V, V] \\ not\_gripperDry : [1 - V, 1 - V] & \leftarrow gripperDry : [V, V] \\ pickup : [1, 1] \leftarrow & \\ gripperDry : [0.7, 0.7] \leftarrow & \end{array}$$

and τ is defined as $\tau(holdBlock) = ncd$ and $\tau(x) = \pi$ for $x \neq holdBlock$, where π is an arbitrary disjunctive p-strategy. According to our semantics, the least fixpoint of P assigns $[1,1]$ to *pickup*, $[0.7, 0.7]$ to *gripperDry*, $[0.3, 0.3]$ to *not_gripperDry*, and $[0.815, 0.815]$ to *holdBlok*. This happens because, by applying the first h-rule, *holdBlock* is derived with the probability interval $[0.665, 0.665]$ and by applying the second h-rule, *holdBlock* is derived with the probability interval $[0.15, 0.15]$. Since the disjunctive negative correlation p-strategy is associated to *holdBlock*, *holdBlock* is assigned

$$\begin{array}{ll}
c_{ncd}\{[0.665, 0.665], [0.15, 0.15]\} & = \\
c_{ncd}([0.665, 0.665], [0.15, 0.15]) & = \\
[\min(1, 0.665 + 0.15), \min(1, 0.665 + 0.15)] & = \\
[\min(1, 0.815), \min(1, 0.815)] = [0.815, 0.815] &
\end{array}$$

which represents the expected solution of the problem.

The upward iteration operator might require an infinite number of iterations to produce the least fixpoint. Consider the following example adapted from [8].

Example 7. Let $P = \langle R, \tau \rangle$ be an h-program where R is

$$\begin{array}{ll}
r : [0.5, 0.5] & \leftarrow \\
q : [0.6, 0.6] & \leftarrow \\
q : [V_1 \times V_2, V_1 \times V_2] & \leftarrow r : [V_1, V_1], q : [V_2, V_2]
\end{array}$$

and $\tau(q) = ind$ and $\tau(q) = \pi$. In the first iteration $T_P \uparrow 1$ assigns $[0.5, 0.5]$ to r, $[0.6, 0.6]$ to q. $T_P \uparrow 2$ assigns $[0.72, 0.72]$ to q while $T_P \uparrow 2(r)$ is the same. After the third iteration, $T_P \uparrow 3(q) = [0.744, 0.744]$. This process continues by providing a better approximation of the interval for q at each iteration. However, the least fixpoint of P can associate a probability interval to q by following the reasoning in [8]. Let $[V_{n-1}, V_{n-1}]$ and $[V_n, V_n]$ be the probability intervals of q after the n-1th and nth fixpoint iterations. These two intervals are related by the relation $[V_n, V_n] = [0.6 + 0.5 \times V_{n-1} - 0.6 \times 0.5 \times V_{n-1}, 0.6 + 0.5 \times V_{n-1} - 0.6 \times 0.5 \times V_{n-1}]$. This simplifies to $[V_n, V_n] = [0.6 + 0.2 \times V_{n-1}, 0.6 + 0.2 \times V_{n-1}]$. In the limit we obtain $[V, V] = [0.6 + 0.2 \times V, 0.6 + 0.2 \times V]$. Solving this equation yields $[V = 0.6 + 0.2 \times V, V = 0.6 + 0.2 \times V] = [0.75, 0.75]$.

Similarly to what reported in [8], this situation arises from the existence of a cyclic inference of the same hybrid basic formula with a higher probability interval. This situation does not arise in h-programs that satisfy *the finite termination property* [8, 18, 1].

Definition 20. *Let P be an h-program. P is said to satisfy the the* finite termination property *iff* $(\forall F \in bf_S(B_L))(\exists\, n < \omega)(lfp(T_P)(F) = T_P \uparrow n(F))$.

3.4 Discussion

In this section we show that the proposed semantics is a natural extension of the traditional semantics of definite logic programs.

First of all, any definite program P can be represented as an h-program program $P' = \langle R, \tau \rangle$, where each definite rule $a \leftarrow b_1, \ldots, b_n \in P$ can be represented as an h-rule of the form $a : c_\rho([V_1, V_1], \ldots, [V_n, V_n]) \leftarrow b_1 : [V_1, V_1], \ldots, b_n : [V_n, V_n] \in R$, where $a, b_1, \ldots, b_n$ are atomic hybrid basic formulas, every $[V_i, V_i]$ is a variable annotation, ρ is any conjunctive p-strategy, c_ρ is the conjunctive composition function corresponding to ρ, and τ is any arbitrary assignment of disjunctive p-strategies. Each fact $a \leftarrow$ is converted to $a : [1,1] \leftarrow$. Observe that $[1,1]$ represents the truth value *true* and $[0,0]$ represents the truth value *false*. In addition, we use any arbitrary disjunctive p-strategy ρ' to disjoin the probability intervals associated the same atomic formula derived from different h-rules. This is because for any disjunctive p-strategy $\rho' \in S$, the probability intervals associated to a is a set of intervals from $\{[0,0],[1,1]\}$, then by the *annihilator axiom*, $c'_\rho\{[1,1], \ldots, [1,1], [0,0], \ldots, [0,0]\} = [1,1]$. Moreover, by the annihilator axiom, for any conjunctive p-strategy ρ, $c_\rho\{[1,1], \ldots, [1,1], [0,0], \ldots, [0,0]\} = [0,0]$.

An interpretation of a definite program P can be viewed as a mapping $I : B_L \rightarrow \{[0,0],[1,1]\}$. Observe also that the program P' contains only atomic formulae; thus, for the sake of simplicity, we will refer to a p-model h simply as a function of the type $h : B_L \rightarrow C[0,1]$.

Proposition 3. *Let P be a definite logic program and let M_P be its least Herbrand model. Let h_P be the least p-model of P'. Then, for each $F \in B_L$ we have that $h_P(F) = [1,1] \Leftrightarrow F \in M_P$.*

3.5 Computation of the Least Fixpoint of HPP_1

In this subsection we provide an algorithm to compute the least fixpoint for a large class of hybrid probabilistic programs, denoted by HPP_1. The proposed algorithm is an adaptation of the Dowling-Gallier algorithm for determining the satisfiability of Horn formulae [3]. Let P be an HPP program. Then P is an HPP_1 program iff each h-rule in P has an annotated *atomic* head. This definition of HPP_1 class simplifies the definition of the T_P operator for any program $P \in HPP_1$ as follows.

Definition 21. *Let $P = \langle R, \tau \rangle$ be a ground h-program in HPP_1 and h a hybrid formula function. The fixpoint operator is a mapping $T_P : HFF \rightarrow HFF$ defined as follows:*

1. $T_P(h)(A) = c_{\tau(A)}\ M_A$ *where*
   $M_A = \{\mu | A : \mu \leftarrow Body \in R$ *such that* $h \models Body\}$.
   If $M_A = \emptyset$, *then* $T_P(h)(A) = [0,0]$
2. $T_P(h)(G_1 \wedge_\rho G_2) = c_\rho(T_P(h)(G_1), T_P(h)(G_2))$ *where* $(G_1 \wedge_\rho G_2) \in bf_S(B_P)$
3. $T_P(h)(G_1 \vee_\rho G_2) = c_\rho(T_P(h)(G_1), T_P(h)(G_2))$ *where* $(G_1 \vee_\rho G_2) \in bf_S(B_P)$.

The different data structures used in the algorithm are described as follows. Assume that $P \in HPP_1$ is a ground hybrid probabilistic program. Let $A.annotation$ denote the annotation assigned to the hybrid basic formula A by the fixpoint operator T_P. This means that, when the algorithm terminates, $A.annotation$

corresponds to $lfp(T_P)(A)$. $A.\mu_r$ denotes the annotation associated to the hybrid basic formula A in the body of the h-rule r. In other words, $A.\mu_r$ refers to the annotation associated to the annotated hybrid basic formula $A : \mu_r$ that appears in the body of the h-rule r. We also maintain, for each atomic hybrid basic formula A, a list called $A.formulalist$, which stores every non-atomic hybrid basic formula that appears in P and contains A. For example, if $(A \wedge_{ig} B)$, $(A \vee_{pc} C)$, and $(B \wedge_{ig} C)$ appear in P, then $A.formulalist$ contains $(A \wedge_{ig} B)$ and $(A \vee_{pc} C)$. For each hybrid basic formula A, atomic or non-atomic, we initialize the list *A.rulelist* with all the h-rules whose bodies contain A. In addition, every h-rule r in P is associated with a counter (*r.counter*) which is initialized to the number of annotated hybrid basic formulas that appear in the body of r.

The algorithm LFP (described below) works by first initializing the queue with every annotation $A.\mu$ which is associated to an annotated fact $A : \mu$ in P. The loop in lines 2-4 assigns $[0, 0]$ to each formula in the lexicographical enumeration, $F_1, \ldots, F_M$, of all hybrid basic formulas appear in P. The algorithm LFP incrementally handles point (1) of Definition 21 in line 8 whereas the points (2) and (3) are handled by the for loop in lines 9-12. The loop in lines 14-22 determines which h-rules in *A.rulelist* are satisfied by the *A.annotation* computed so far and it decrements the counter *r.counter* if the condition $A.\mu_r \leq_t A.annotation$ holds. In particular, whenever the counter *r.counter* reaches zero (line 18), the h-rule r fires and the annotation associated to its head is entered into the queue. The complete description of the algorithm is given below.

```
 1: Algorithm LFP
 2: for each $i$ such that $1 \leq i \leq M$ do
 3:    $F_i.annotation := [0, 0]$
 4: end for
 5: while queue is not empty do
 6:    $(A.\mu) := pop(queue)$
 7:    if A is atomic then
 8:       $A.annotation := c_{\tau(A)}(A.annotation, A.\mu)$
 9:       for each $F = A_1 *_\rho \ldots *_\rho A_r (r > 1) \in A.formulalist$ do
10:          $F.annotation := c_\rho(A_1.annotation, ..., A_r.annotation)$
11:          enqueue $F.annotation$
12:       end for
13:    end if
14:    for each rule $r \in A.rulelist$ do
15:       if $A.\mu_r \leq_t A.annotation$ then
16:          $r.counter := r.counter - 1$
17:          delete $r$ from $A.rulelist$
18:          if $r.counter = 0$ then
19:             enqueue $(H.\beta)$ where $H : \beta$ is the head of $r$
20:          end if
21:       end if
22:    end for
23: end while
```

Theorem 2. *Let P be a program in HPP_1 and let N be the number of all hybrid basic formulae in P. The algorithm LFP computes the least fixpoint of T_P in time $O(N^2)$.*

4 Implication-Based Approaches

In this section we show that the probabilistic *Implication Based (IB)* framework [12] is subsumed by a straightforward extension of the hybrid probabilistic framework proposed here. Let us start by performing the following simple extensions:

- Probabilities in h-programs are now expressed as *pairs of intervals*;
- The p-strategies are easily extended to deal with pairs of intervals, in a way similar to the one described in [12].

Let L_C be the set $C[0,1] \times C[0,1]$. A probabilistic rule (p-rule) in the IB framework is an expression of the form $(r; \rho, \rho')$, where r is of the form $(H \overset{c}{\leftarrow} A_1, \ldots, A_n)$, $H, A_1, \ldots, A_n$ are atoms, $c \in L_C$, ρ is the conjunctive p-strategy to be used in the body of r, and ρ' is the disjunctive p-strategy associated to the head of r. A *probabilistic program (p-program)* is a finite set of p-rules, such that p-rules with the same head in the p-program make use of the same disjunctive p-strategy ρ'. Let P be a p-program and B_L be the Herbrand base. The notion of interpretation in the IB framework [12] is defined as a mapping $I : B_L \to L_C$. Associated to each p-program, P, is an operator T_P^{IB} that maps an interpretation to an interpretation and is defined as follows:

$$T_P^{IB}(I)(H) = \bigvee_{\rho'} \left\{ \begin{array}{r} c \wedge_\rho I(A_1) \wedge_\rho \ldots \wedge_\rho I(A_n) | \\ (H \overset{c}{\leftarrow} A_1, \ldots, A_n; \rho, \rho') \text{ is a ground} \\ \text{instance of p-rule in } P \end{array} \right\}$$

The same computation can be expressed in our framework as:
$c_{\rho' d}\{c_{\rho c}\{c, I(A_1), \ldots, I(A_n)\} | (H \overset{c}{\leftarrow} A_1, \ldots, A_n; \rho, \rho') \text{ is a ground p-rule in } P\}$

This is possible because ρ and ρ' correspond to p-strategies in the sense of our semantics, where ρc corresponds to the conjunctive p-strategy of ρ and $\rho' d$ corresponds to the disjunctive p-strategy of ρ'.

Definition 22 (Translation). *Let P be a p-program. Then P can be translated into an h-program, $HP_r(P) =< R, \tau >$, where*

$$R = \left\{ \begin{array}{r} H : c_{\rho c}(c, \mu_1, \ldots, \mu_n) \leftarrow A_1 : \mu_1, \ldots, A_n : \mu_n | \\ (H \overset{c}{\longleftarrow} A_1, \ldots, A_n; \rho, \rho') \in P, \\ \mu_1, \ldots, \mu_n \text{ are variable annotations} \end{array} \right\}$$
$$\tau(H) = \rho', \quad \forall\ (H \overset{c}{\longleftarrow} A_1, \ldots, A_n; \rho, \rho') \in P$$

The following theorem establishes the relation between the probabilistic IB framework and the new hybrid probabilistic programs framework under the assumption that, without loss of generality, an interpretation in the new hybrid probabilistic programs framework is a mapping of the type $h : B_L \to L_C$.

Theorem 3. *Let P be a p-program. Then $T_{HP_r(P)} = T_P^{IB}$.*

5 Final Considerations and Future Work

In this work, we proposed an extension of the hybrid probabilistic programming (HPP) framework, along with a new semantics for the extended language. Having such a new language and semantics is important in order to enable more realistic and practical applications, such as those in probabilistic planning, and to make the framework more practical and easy to use. The new framework of hybrid probabilistic logic programs is built on the work of the two major approaches in probabilistic logic programming: the probabilistic annotation-based approach [18, 19, 1] and the probabilistic implication-based approach [12]. In the framework proposed here, probabilities are expressed as intervals and they are associated to facts and rules in a logic program in a manner similar to the one proposed in the hybrid probabilistic programs framework of [1]. However, the syntax of programs in our framework is different, as it allows the user to explicitly encode the knowledge about how to compose the probability intervals associated to the same hybrid basic formulas derived from different rules in the programs.

The semantics of programs in our framework is quite different from [1]. This is because we employ the truth order while [1] uses the set inclusion order; we believe that the use of truth order in our framework makes it more suitable for reasoning about real-world applications. Furthermore, we believe this work lays the foundations for a more general framework which is the parametric on the choice of probability ordering. We make use of explicit composition functions to compute the overall probability interval of hybrid basic formulas. Our usage of truth order and composition functions of the disjunctive p-strategies is close to [12]. However, HPP strictly subsumes the scheme of [12].

A topic of future research is to extend the new hybrid probabilistic programs framework with non-monotonic negation, through an adaptation of the stable model semantics and well-founded semantics to the framework proposed here.

References

1. A. Dekhtyar and V. S. Subrahmanian. Hybrid Probabilistic Program. *Journal of Logic Programming*, 43(3):187-250, 2000.
2. M. Dekhtyar, A. Dekhtyar, and V. S. Subrahmanian. Hybrid Probabilistic Programs: Algorithms and Complexity. *In Proc. of UAI Conference*, pages 160-169, 1999.
3. W. F. Dowling and J. H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267-284, 1984.
4. D. Dubois et al. Towards Possibilistic Logic Programming. *ICLP*, MIT Press, 1991.
5. M. C. Fitting. Logic Programming on A Topological Bilattice. *Fundamenta Informaticae*, 11:209-218, 1988.
6. M.C. Fitting. Bilattices and The Semantics of Logic Programming. *Journal of Logic Programming*, 11:91-16, 1991.
7. K. Kersting and L. De Raedt. Bayesian Logic Programs. In *Inductive LP*, 2000.

8. M. Kifer and A. Li. On The Semantics of Rule-Based Expert Systems with Uncertainty. In *Intl. Conf. on Database Theory*, 1988. Springer-Verlag.
9. M. Kifer and V. S. Subrahmanian. Theory of Generalized Annotated Logic Programming and Its Applications. *Journal of Logic Programming*, 12:335-367, 1992.
10. N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Planning. *Artificial Intelligence*, 76(1-2):239-286, 1995.
11. V.S.L. Lakshmanan and F. Sadri. Modeling Uncertainty in Deductive Databases. *In Conf. on Database Expert Systems and Applications*, 1994. Springer-Verlag.
12. V.S.L. Lakshmanan and F. Sadri. Probabilistic Deductive Databases. *In Intl. Logic Programming Symposium*, 1994. MIT Press.
13. V.S.L. Lakshmanan and F. Sadri. Uncertain Deductive Databases: A Hybrid Approach. *Information Systems*, 22(8):483-508, December 1997.
14. V.S.L. Lakshmanan and N. Shiri. A Parametric Approach to Deductive Databases with Uncertainty. *IEEE TKDE*, 13(4):554-570, 2001.
15. S.M. Leach and J.J. Lu. Query Processing in Annotated Logic Programming. *Journal of Intelligent Information Systems*, 6(1):33-58, 1996.
16. Y. Loyer and U. Straccia. The Approximate Well-founded Semantics for Logic Programs with Uncertainty. In *MFCS*, 2003, Springer Verlag.
17. J.J. Lu et al. Computing Annotated Logic Programs. *ICLP*, MIT press, 1994.
18. R. T. Ng and V. S. Subrahmanian. Probabilistic Logic Programming. *Information and Computation*, 101(2):150-201, December 1992.
19. R.T. Ng and V.S. Subrahmanian. A Semantical Framework for Supporting Subjective and Conditional Probabilities in Deductive DBs. *J. Automated Reasoning*, 10(2), 1993.
20. R. T. Ng and V. S. Subrahmanian. Stable Semantics for Probabilistic Deductive Databases. *Information and Computation*, 110(1):42-83, 1994.
21. T. Sato, Y. Kameya. PRISM: Language for Symbolic-Statistical Modeling. *IJCAI*, 1997.
22. E. Shapiro. Logic Programs with Uncertainties: A Tool for Implementing Expert Systems. *In Proc. of IJCAI*, pages 529-532, 1983.
23. V. S. Subrahmanian. On The Semantics of Quantitative Logic Programs. *In Symp. on Logic Programming*, pages 173-182, IEEE Computer Society, 1987.
24. M.H. van Emden. Quantitative Deduction and Its Fixpoint Theory. *Journal of Logic Programming*, 4(1):37-53, 1986.
25. J. Vennekens and S. Verbaeten. A General View on Probabilistic Logic Programming. In *Belgian-Dutch Conference on AI*, 2003.

Safe Programming with Pointers Through Stateful Views[⋆]

Dengping Zhu and Hongwei Xi

Computer Science Department
Boston University
{zhudp,hwxi}@cs.bu.edu

Abstract. The need for direct memory manipulation through pointers is essential in many applications. However, it is also commonly understood that the use (or probably misuse) of pointers is often a rich source of program errors. Therefore, approaches that can effectively enforce safe use of pointers in programming are highly sought after. ATS is a programming language with a type system rooted in a recently developed framework *Applied Type System*, and a novel and desirable feature in ATS lies in its support for safe programming with pointers through a novel notion of *stateful views*. In particular, even pointer arithmetic is allowed in ATS and guaranteed to be safe by the type system of ATS. In this paper, we give an overview of this feature in ATS, presenting some interesting examples based on a prototype implementation of ATS to demonstrate the practicality of safe programming with pointer through stateful views.

1 Introduction

The verification of program correctness with respect to specification is a highly significant problem that is ever present in programming. There have been many approaches developed to address this fundamental problem (e.g., Floyd-Hoare logic [Hoa69,AO91], model checking [EGP99]), but they are often too expensive to be put into general software practice. For instance, Floyd-Hoare logic is mostly employed to prove the correctness of some (usually) short but often intricate programs, or to identify some subtle problems in such programs. Though larger programs can be handled with the help of automated theorem proving, it is still as challenging as it was to support Floyd-Hoare logic in a realistic programming languages. On the other hand, the verification of type correctness of programs, that is, type-checking, in languages such as ML and Java scales convincingly in practice. However, we must note that the types in ML and Java are of relatively limited expressive power when compared to Floyd-Hoare logic. Therefore, we are naturally led to form type systems in which more sophisticated properties can be captured and then verified through type-checking.

A heavy-weighted approach is to adopt a type system in which highly sophisticated properties on programs can be captured. For instance, the type system of NuPrl [$C^{+}86$] based on Martin-Löf's constructive type theory is such a case. In such a type system,

[⋆] Partially supported by NSF grant no. CCR-0229480

types are exceedingly expressive but type-checking often involves a great deal of theorem proving and becomes intractable to automate. This is essentially an approach that strongly favors expressiveness over scalability.

We adopt a light-weighted approach, introducing a notion of restricted form of dependent types, where we clearly separate type index expressions from run-time expressions. In functional programming, we have enriched the type system of ML with such a form of dependent types, leading to the design of a functional programming language DML (Dependent ML) [Xi98,XP99]. In imperative programming, we have designed a programming language Xanadu with C-like syntax to support such a form of dependent types. Along a different but closely related line of research, a new notion of types called guarded recursive (g.r.) datatypes is recently introduced [XCC03]. Noting the close resemblance between the restricted form of dependent types (developed in DML) and g.r. datatypes, we immediately initiated an effort to design a unified framework for both forms of types, leading to the formalization of *Applied Type System* ($\mathcal{ATS}$) [Xi03,Xi04]. We are currently in the process of designing and implementing ATS, a programming language with its type system rooted in $\mathcal{ATS}$. A prototype of ATS (with minimal documentation and many examples) is available on-line [Xi03]. Note that we currently use the name *$\mathcal{ATS}$-style dependent types* for the dependent types in $\mathcal{ATS}$ so as to distinguish them from the dependent types in Martin-Löf's constructive type theory.

```
fun arrayAssign {a:type, n:nat} (A:array (a,n), B:array (a,n)): unit =
  let
     fun loop {i:nat | i <= n} (ind: int (i)): unit =
       if ind < length A then
         (set (B, ind, get (A, ind)); loop (ind + 1))
  in
     loop (0)
  end
```

Fig. 1. A simple example in ATS

ATS is a comprehensive programming language designed to support a variety of programming paradigms (e.g., functional programming, object-oriented programming, imperative programming, modular programming, meta-programming), and the core of ATS is a call-by-value functional programming language. In this paper, we are to focus on the issue of programming with pointers in ATS.

As programming with dependent types is currently not a common practice, we use a concrete example to give the reader some feel as to how dependent types can be used to capture program invariants. In Figure 1, we implement a function *arrayAssign* that assigns the content of one array to another array. The header in the definition of the function *arrayAssign* means that *arrayAssign* is assigned the following type:

$$\forall a : \textit{type}.\forall n : \textit{nat}.(\textbf{array}(a, n), \textbf{array}(a, n)) \rightarrow \textbf{1}$$

We use **1** for the unit type, which roughly corresponds to the void type in C. Given a type T and an integer I, we use $\textbf{array}(T, I)$ as the type for arrays of size I in which each element is assigned the type T. Therefore, the type given to *arrayAssign* indicates that *arrayAssign* can only be applied to two arrays of the same size. The quantifications

$\forall a : \textit{type}$ and $\forall n : \textit{nat}$ mean that a and n can be instantiated with any given type and natural number, respectively. The inner function *loop* is assigned the following type: $\forall i : \textit{nat}.i \leq n \supset (\mathbf{int}(i) \rightarrow \mathbf{1})$. Given an integer I, we use $\mathbf{int}(I)$ as the singleton type for I, that is, the only value of type $\mathbf{int}(I)$ equals I. The type given to *loop* means that *loop* can only be applied to a natural number whose value is less than or equal to n, which is the size of the arguments of *arrayAssign*. In ATS, we call $i \leq n$ a guard and $i \leq n \supset (\mathbf{int}(i) \rightarrow \mathbf{1})$ a guarded type. Also we point out that the function *length* is given the following type:

$$\textit{length} \quad : \quad \forall a : \textit{type}.\forall n : \textit{nat}.\mathbf{array}(a, n) \rightarrow \mathbf{int}(n)$$

and the array subscripting function *get* and the array updating function *set* are given the following types:

$$\begin{array}{rcl} \textit{get} & : & \forall a : \textit{type}.\forall n : \textit{nat}.\forall i : \textit{nat}.i < n \supset ((\mathbf{array}(a, n), \mathbf{int}(i)) \rightarrow a) \\ \textit{set} & : & \forall a : \textit{type}.\forall n : \textit{nat}.\forall i : \textit{nat}.i < n \supset ((\mathbf{array}(a, n), \mathbf{int}(i), a) \rightarrow \mathbf{1}) \end{array}$$

which indicate that the index used to access an array must be within the bounds of the array.

To support safe programming with pointers, a notion called *stateful view* is introduced in ATS to model memory layout. Given a type T and an address L, we use $T@L$ for the (stateful) view indicating that a value of type T is stored at address L. This is the only form of a primitive view and all other views are built on top of such primitive views. For instance, we can form a view $(T@L, T'@(L + 1))$ to mean that a value of type T and another value of type T' are stored at addresses L and $L + 1$, respectively, where we use $L + 1$ for the address immediately following L. A stateful view is similar to a type, and it can be assigned to certain terms, which we often refer to as proof terms (or simply proofs) of stateful views. We treat proofs of views as a form of resources, which can be consumed as well as generated. In particular, the type theory on views is based on a form of linear logic [Gir87].

Certain functions may require proofs of stateful views when applied and they may cause stateful views to change when executed. For instance, the functions *getVar* and *setVar* are given the following types:

$$\begin{array}{rcl} \textit{getVar} & : & \forall a : \textit{type}.\forall l : \textit{addr}.(a@l \mid \mathbf{ptr}(l)) \rightarrow (a@l \mid a) \\ \textit{setVar} & : & \forall a_1 : \textit{type}.\forall a_2 : \textit{type}.\forall l : \textit{addr}.(a_1@l \mid a_2, \mathbf{ptr}(l)) \rightarrow (a_2@l \mid \mathbf{1}) \end{array}$$

where we use $\mathbf{ptr}(L)$ as the singleton type for the pointer pointing to a given address L.

The type assigned to *getVar* means that the function takes a proof of view $T@L$ for some type T and address L, and a value of type $\mathbf{ptr}(L)$, and then returns a proof of view $T@L$ and a value of type T. In this case, we say that a proof of view $T@L$ is consumed and another proof of view $T@L$ is generated. We emphasize that proofs are only used at compile-time for performing type-checking and they are neither needed nor available at run-time. We use *getVar* here as the function that reads from a given pointer. Note that the proof argument of *getVar* essentially assures that the pointer passed to *getVar* cannot be a dangling pointer as the proof argument indicates that a value of certain type is stored at the address to which the pointer points.

```
fun swap {t1:type, t2:type, l1:addr, l2:addr}
    (pf1: t1 @ l1, pf2: t2 @ l2 | p1: ptr (l1), p2: ptr (l2))
  : '(t1 @ l2, t2 @ l1 | unit) =
  let
     val '(pf1 | tmp1) = getVar (pf1 | p1)
     val '(pf2 | tmp2) = getVar (pf2 | p2)
     val '( pf1' | _ ) = setVar (pf1 | p1, tmp2)
     val '( pf2' | _ ) = setVar (pf2 | p2, tmp1)
  in
    '(pf2', pf1' | '())
  end
```

Fig. 2. A simple swap function

The type assigned to the function *setVar* can be understood in a similar manner: *setVar* takes a proof of view $T_1@L$ for some type T_1 and address L and a value of type T_2 for some type T_2 and another value of type $\mathbf{ptr}(L)$, and then returns a proof of view $T_2@L$ and the unit (of type $\mathbf{1}$). In this case, we say that a proof of view $T_1@L$ is consumed and another proof of view $T_2@L$ is generated. Since we use *setVar* here as the function that writes to a given address, this change precisely reflects the situations before and after the function *setVar* is called: A value of type T_1 is stored at L before the call and a value of type T_2 is stored at L after the call.

The functions *allocVar* and *freeVar*, which allocates and deallocates a memory unit, respectively, are also of interest, and their types are given as follows:

$$\begin{array}{rcl} allocVar & : & () \rightarrow \exists l : addr.(\mathbf{top}@l \mid \mathbf{ptr}(l)) \\ freeVar & : & \forall a : type.\forall l : addr.(a@l \mid \mathbf{ptr}(l)) \rightarrow \mathbf{1} \end{array}$$

We use **top** for the top type, that is, every type is a subtype of **top**. So when called, *allocVar* returns a proof of view $\mathbf{top}@L$ for some address L and a pointer of type $\mathbf{ptr}(L)$. The proof is needed if a write operation through the pointer is to be done. On the other hand, a call to *freeVar* makes a pointer no longer accessible.

As an example, a function is implemented in Figure 2 that swaps the contents stored at two (distinct) addresses. We use $'(\ldots)$ to form tuples, where the quote symbol ($'$) is solely for the purpose of parsing. For instance, $'()$ stands for the unit (i.e., the tuple of length 0). Also, the bar symbol ($|$) is used as a separator (like the comma symbol (,)).

Note that proofs are manipulated explicitly in the above implementation, and this could be burdensome to a programmer. In ATS we also allow certain proofs be consumed and generated implicitly. For instance, the function in Figure 2 may also be implemented as follows in ATS:

```
fun swap {t1:type, t2:type, l1:addr, l2:addr}
    (pf1: t1 @ l1, pf2: t2 @ l2 | p1: ptr (l1), p2: ptr (l2))
  : '(t1 @ l2, t2 @ l1 | unit) =
  let val tmp := !p1 in p1 := !p2; p2 := tmp end
```

where we use ! for *getVar* and := for *setVar* and deal with proofs in an implicit manner.

The primary goal of the paper is to make ATS accessible to a wider audience who may or may not have adequate formal training in type theory. We are thus intentionally to avoid presenting the (intimidating) theoretical details on ATS as much as possible, striving for a clean and intuitive introduction to the use of stateful views in support of safe programming with pointers. For the reader who is interested in the technical development of ATS, please refer to [Xi03] for further details. Also, there is a prototype implementation of ATS as well as many interesting examples available on-line [Xi03].

We organize the rest of the paper as follows. In Section 2, we give brief explanation on some (uncommon) forms of types in ATS. We then present some examples in Section 3, showing how programming with pointers is made safe in ATS. We mention some related work in Section 4 and conclude in Section 5.

2 ATS/SV in a Nutshell

In this section, we present a brief overview of *ATS/SV*, the type system that supports imperative programming (with pointers) in ATS. As an applied type system, there are two components in *ATS/SV*: static component (statics) and dynamic component (dynamics). Intuitively, the statics and dynamics are each for handling types and programs, respectively, and we are to focus on the statics of *ATS/SV*.

$$\begin{array}{lrcl}
\text{sorts} & \sigma & ::= & \mathit{addr} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{type} \\
\text{static contexts} & \Sigma & ::= & \emptyset \mid \Sigma, a : \sigma \\
\text{static addr.} & L & ::= & a \mid l \mid L + I \\
\text{static int.} & I & ::= & a \mid i \mid c_I(s_1, \ldots, s_n) \\
\text{static prop.} & P & ::= & a \mid b \mid c_P(s_1, \ldots, s_n) \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \supset P_2 \\
\text{types} & T & ::= & a \mid \delta(\boldsymbol{s}) \mid (\overline{V} \mid T) \rightarrow \mathit{CT} \mid P \supset T \mid \forall a : \sigma.T \mid P \wedge T \mid \exists a : \sigma.T \\
\text{computation types} & \mathit{CT} & ::= & \exists \Sigma, \overline{P}.(\overline{V} \mid T) \\
\text{stateful views} & V & ::= & \top \mid T@L \mid \overline{\delta}(\boldsymbol{s}) \mid V_1 \multimap V_2 \mid V_1 \otimes V_2
\end{array}$$

Fig. 3. The syntax for the statics of *ATS/SV*

The syntax for the statics of *ATS/SV* is given in Figure 3. The statics itself is a simply typed language and a type in it is called a *sort*. We assume the existence of the following basic sorts in *ATS/SV*: *addr*, *bool*, *int* and *type*; *addr* is the sort for addresses, and *bool* is the sort for boolean constants, and *int* is the sort for integers, and *type* is the sort for types (which are to be assigned to dynamic terms, i.e., programs). We use a for static variables, l for address constants $\mathbf{l}_0, \mathbf{l}_1, \ldots$, b for boolean values *tt* and *ff*, and i for integers $0, -1, 1, \ldots$. A term s in the statics is called a static term, and we use $\Sigma \vdash s : \sigma$ to mean that s can be assigned the sort σ under Σ. The rules for assigning sorts to static terms are all omitted as they are completely standard. We may also use L, P, I, T for static terms of sorts *addr*, *bool*, *int*, *type*, respectively. We assume some primitive functions c_I when forming static terms of sort *int*; for instance, we can form terms such as $I_1 + I_2$, $I_1 - I_2$, $I_1 * I_2$ and I_1 / I_2. Also we assume certain primitive functions c_P when forming static terms of sort *bool*; for instance, we can form propositions such as $I_1 \leq I_2$ and $I_1 \geq I_2$, and for each sort σ we can form a proposition $s_1 =_\sigma s_2$ if s_1 and s_2 are static terms of sort σ; we may omit the subscript σ in $=_\sigma$ if it can be readily inferred from the context. In addition, given L and I, we can form an address $L + I$, which equals $\mathbf{l}_{n+i}$ if $L = \mathbf{l}_n$ and $I = i$ and $n + i \geq 0$.

We use $\boldsymbol{s}$ for a sequence of static terms, and $\overline{P}$, $\overline{T}$ and $\overline{V}$ for sequences of propositions, types and views, respectively, and $\emptyset$ for the empty sequence.

We use *ST* for a state, which is a finite mapping from addresses to values, and **dom**(*ST*) for the domain of *ST*. We say that a value v is stored at l in *ST* if $ST(l) = v$. Note that we assume that every value takes one memory unit to store, and this, for instance, can be achieved through proper boxing. Given two states ST_1 and ST_2, we write $ST_1 \otimes ST_2$ for the union of ST_1 and ST_2 if $\mathbf{dom}(ST_1) \cap \mathbf{dom}(ST_2) = \emptyset$. We write $ST : V$ to mean that the state *ST* meets the view *V*. We now present some intuitive explanation on certain forms of views and types.

- We use $\top$ for the empty view, which is met by the empty state, that is, the state whose domain is empty.
- We use $\overline{\delta}$ for a view constructor and write $\vdash \overline{\delta}(\sigma_1, \ldots, \sigma_n)$ to mean that applying $\overline{\delta}$ to static terms $s_1, \ldots, s_n$ of sorts $\sigma_1, \ldots, \sigma_n$, respectively, generates a view $\overline{\delta}(s_1, \ldots, s_n)$. There are certain view proof constructors c associated with each $\overline{\delta}$, which are assigned views of the form $\forall \Sigma, \overline{P}.(\overline{V}) \multimap \overline{\delta}(\boldsymbol{s})$. For example, the (recursively defined) view constructor *arrayView* in Figure 6 (in Section 3) forms a view *arrayView*(T, I, L) when applied to a type T, an integer I and an address L; the two proof constructors associated with *arrayView* are *ArrayNone* and *ArraySome*.
- Given L and T, we can form a primitive view $T@L$, which is met by the state that maps L to a value of type T.
- Given V_1 and V_2, a state *ST* meets $V_1 \multimap V_2$ if $ST_1 \otimes ST$ meets V_2 for any state $ST_1 : V_1$ such that $\mathbf{dom}(ST_1) \cap \mathbf{dom}(ST) = \emptyset$.
- Given V_1 and V_2, a state *ST* meets $V_1 \otimes V_2$ if $ST = ST_1 \otimes ST_2$ for some $ST_1 : V_1$ and $ST_2 : V_2$.
- In general, we use $\delta(\boldsymbol{s})$ for primitive types in *ATS/SV*. For instance, **top** is the top type, that is, every type is a subtype of **top**; **1** is the unit type; $\mathbf{ptr}(L)$ is a singleton type containing the only address equal to L, and we may also refer to a value of type $\mathbf{ptr}(L)$ as a pointer (pointing to L); $\mathbf{bool}(P)$ is a singleton type containing the only boolean value equal to P; $\mathbf{int}(I)$ is a singleton type containing the only integer equal to I.
- $(\overline{V} \mid T) \rightarrow CT$ is a type for (dynamic) functions that can be applied to values of type T only if the current state (when the application occurs) meets the views $\overline{V}$, and such an application yields a dynamic term that can be assigned the computation type *CT* of the form $\exists \Sigma', \overline{P}'.(\overline{V}' \mid T')$, which intuitively means that the dynamic term is expected to evaluate to value v at certain state *ST* such that for some static substitution Θ, each proposition in $\overline{P}'[\Theta]$ is true, v is of type $T'[\Theta]$ and *ST* meets $V'[\Theta]$. In the following presentation, we use $T_1 \rightarrow T_2$ as a shorthand for $(\emptyset \mid T_1) \rightarrow \exists \emptyset, \emptyset.(\emptyset \mid T_2)$ and call it a stateless function type.
- $P \supset T$ is called a guarded type and $P \wedge T$ is called an asserting type. As an example, the following type is for a function from natural numbers to negative integers:

$$\forall a : int.a \geq 0 \supset (\mathbf{int}(a) \rightarrow \exists a' : int.(a' < 0) \wedge \mathbf{int}(a'))$$

The guard $a \geq 0$ indicates that the function can only be applied to an integer that is greater than or equal to 0; the assertion $a' < 0$ means that each integer returned by the function is negative.

$$\frac{}{\Sigma; \overline{P} \models T \leq_{tp} \textbf{top}} \qquad \frac{}{\Sigma; \overline{P} \models T \leq_{tp} T} \qquad \frac{\Sigma; \overline{P} \models T_1 \leq_{tp} T_2 \quad \Sigma; \overline{P} \models T_2 \leq_{tp} T_3}{\Sigma; \overline{P} \models T_1 \leq_{tp} T_3}$$

$$\frac{\vdash \delta(\sigma_1, \ldots, \sigma_n) \quad \Sigma; \overline{P} \models s_i \equiv_{\sigma_i} s_i' \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P} \models \delta(s_1, \ldots, s_n) \leq_{tp} \delta(s_1', \ldots, s_n')}$$

$$\frac{\Sigma; \overline{P}; \overline{V'} \models \otimes(\overline{V}) \quad \Sigma; \overline{P} \models T' \leq_{tp} T \quad \Sigma; \overline{P} \models CT \leq_{ct} CT'}{\Sigma; \overline{P} \models (\overline{V} \mid T) \rightarrow CT \leq_{tp} (\overline{V'} \mid T') \rightarrow CT'}$$

$$\frac{}{\Sigma; \overline{P} \vdash (\overline{V} \mid T) \rightarrow CT \leq_{tp} (\overline{V}, V \mid T) \rightarrow CT[V]} \textbf{(ext)}$$

$$\frac{\Sigma; \overline{P}, P' \models P \quad \Sigma; \overline{P}, P' \models T \leq_{tp} T'}{\Sigma; \overline{P} \models P \supset T \leq_{tp} P' \supset T'} \qquad \frac{\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \forall a : \sigma.T \leq_{tp} \forall a : \sigma.T'}$$

$$\frac{\Sigma; \overline{P}, P \models P' \quad \Sigma; \overline{P}, P \models T \leq_{tp} T'}{\Sigma; \overline{P} \models P \wedge T \leq_{tp} P' \wedge T'} \qquad \frac{\Sigma, a : \sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \exists a : \sigma.T \leq_{tp} \exists a : \sigma.T'}$$

$$\frac{\Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models \overline{P}_0' \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0; \overline{V} \models \otimes(\overline{V'}) \quad \Sigma, \Sigma_0; \overline{P}, \overline{P}_0 \models T \leq_{tp} T'}{\Sigma; \overline{P} \models \exists \Sigma_0, \overline{P}_0.(\overline{V} \mid T) \leq_{ct} \exists \Sigma_0, \overline{P}_0'.(\overline{V'} \mid T')}$$

Fig. 4. The subtype rules

There are two forms of constraints in *ATS/SV*: $\Sigma; \overline{P} \models P$ (persistent) and $\Sigma; \overline{P}; \overline{V} \models V$ (ephemeral), which are needed to define type equality. Generally speaking, we use intuitionistic logic and intuitionistic linear logic to reason about persistent and ephemeral constraints, respectively. We may write $\Sigma; \overline{P} \models \overline{P}_0$ to mean that $\Sigma; \overline{P} \models P$ holds for every P in $\overline{P}_0$. Most of the rules for proving persistent constraints are standard and thus omitted. For instance, the following rules are available:

$$\frac{}{\Sigma; \overline{P}, P \models P} \qquad \frac{\Sigma; \overline{P}, \neg P \models \mathit{ff}}{\Sigma; \overline{P} \models P} \qquad \frac{\Sigma; \overline{P}, P_1 \models P_2}{\Sigma; \overline{P} \models P_1 \supset P_2} \qquad \frac{\Sigma; \overline{P} \models P_1 \supset P_2 \quad \Sigma; \overline{P} \models P_1}{\Sigma; \overline{P} \models P_2}$$

We introduce a subtype relation $T_1 \leq_{tp} T_2$ on static terms of sort *type* and define the type equality $T_1 =_{type} T_2$ to be $T_1 \leq_{tp} T_2 \wedge T_2 \leq_{tp} T_1$. A subtype judgment is of the form $\Sigma; \overline{P} \models T_1 \leq_{tp} T_2$, and the rules for deriving such a judgment are given in Figure 4, where the obvious side conditions associated with certain rules are omitted. Note that $\otimes(\overline{V})$ is defined to be $\top$ if $\overline{V}$ is empty or $V_1 \otimes \ldots \otimes V_n$ if $\overline{V} = V_1, \ldots, V_n$ for some $n \geq 1$. In the rule **(ext)**, we write $CT[V]$ for $\exists \Sigma, \overline{P}.(\overline{V}, V \mid T)$, where CT is $\exists \Sigma, \overline{P}.(\overline{V} \mid T)$ and no free variables in V occur in Σ. For those who are familiar with *separation logic* [Rey02], we point out that this rule essentially corresponds to the frame rule there. The rule **(ext)** is essential: For instance, suppose the type of a function is $(\overline{V} \mid T) \rightarrow CT$ and the current state meets the view $\otimes(\overline{V}_0)$ such that $\overline{V}_0 = \overline{V}_1, V$ and $\emptyset; \emptyset; \overline{V}_1 \models \otimes(\overline{V})$ is derivable. In order to apply the function at the current state, we need to assign the type $(\overline{V}, V \mid T) \rightarrow CT[V]$ to the function so that the view V can be "carried over". This can be achieved by an application of the rule **(ext)**.

Some of the rules for proving ephemeral constraints are given in Figure 5, and the rest are associated with primitive view constructors. Given primitive view constructor $\overline{\delta}$ with proof constructors $c_1, \ldots, c_n$, we introduce the following rule for each c_i,

$$\frac{\Sigma; \overline{P} \models T \leq_{tp} T'}{\Sigma; \overline{P}; T@L \models T'@L} \qquad \frac{}{\Sigma; \overline{P}; \emptyset \models \top} \qquad \frac{\Sigma; \overline{P}; \overline{V} \models V}{\Sigma; \overline{P}; \overline{V}, \top \models V}$$

$$\frac{\Sigma; \overline{P}; \overline{V}_1 \models V_1 \quad \Sigma; \overline{P}; \overline{V}_2 \models V_2}{\Sigma; \overline{P}; \overline{V}_1, \overline{V}_2 \models V_1 \otimes V_2} \qquad \frac{\Sigma; \overline{P}; \overline{V}, V_1, V_2 \models V}{\Sigma; \overline{P}; \overline{V}, V_1 \otimes V_2 \models V}$$

$$\frac{\Sigma; \overline{P}; \overline{V}, V_1 \models V_2}{\Sigma; \overline{P}; \overline{V} \models V_1 \multimap V_2} \qquad \frac{\Sigma; \overline{P}; \overline{V}_1 \models V_1 \multimap V_2 \quad \Sigma; \overline{P}; \overline{V}_2 \models V_1}{\Sigma; \overline{P}; \overline{V}_1, \overline{V}_2 \models V_2}$$

$$\frac{\vdash \overline{\delta}(\sigma_1, \ldots, \sigma_n) \quad \Sigma; \overline{P} \models s_i \equiv_{\sigma_i} s'_i \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P}; \overline{\delta}(s_1, \ldots, s_n) \models \overline{\delta}(s'_1, \ldots, s'_n)}$$

$$\frac{\Sigma; \overline{P}[a \mapsto i]; \overline{V}[a \mapsto i] \vdash V[a \mapsto i] \text{ for every integer } i}{\Sigma, a : \mathit{int}; \overline{P}; \overline{V} \vdash V}$$

Fig. 5. Some rules for ephemeral constraints

```
dataview arrayView (type, int, addr) =
  | {a:type, l:addr} ArrayNone (a, 0, l)
  | {a:type, n:nat, l:addr}
      ArraySome (a, n+1, l) of (a @ l, arrayView (a, n, l+1))
```

Fig. 6. An dataview for arrays

$$\frac{\Sigma \vdash \Theta : \Sigma_0 \quad \Sigma \models \overline{P}_0[\Theta] \quad \Sigma; \overline{P}; \overline{V} \models \otimes(\overline{V}_i[\Theta])}{\Sigma; \overline{P}; \overline{V} \models \overline{\delta}(\boldsymbol{s}_i[\Theta])}$$

where we assume that c_i is assigned the following view: $\forall \Sigma_i, \overline{P}_i.(\overline{V}_i) \multimap \overline{\delta}(\boldsymbol{s}_i)$; in addition, we introduce the following rule:

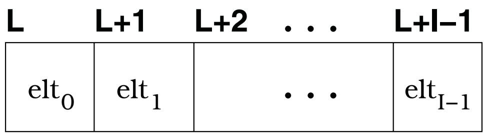
$$\frac{\Sigma, \Sigma_i; \overline{P}, \overline{P}_i, \boldsymbol{s} = \boldsymbol{s}_i; \overline{V}, \overline{V}_i \models V \text{ for } 1 \leq i \leq n}{\Sigma; \overline{P}; \overline{V}, \overline{\delta}(\boldsymbol{s}) \models V}$$

The key point we stress here is that both the persistent and ephemeral constraint relations can be formally defined.

3 Examples

3.1 Arrays

Array is probably the most commonly used data structure in programming. We declare in Figure 6 a dataview for representing arrays. Given a type T, an integer I and an address L, *arrayView*(T, I, L) is a view for an array pictured as follows,



such that (1) each element of the array is of type T, (2) the length of the array is I and (3) the array starts at address L and ends at address $L + I - 1$.There are two view proof constructors *ArrayNone* and *ArraySome* associated with the view *arrayView*, which are assigned the following functional views:

$$\begin{array}{lcl} \mathit{ArrayNone} & : & \forall a : \mathit{type}.\forall l : \mathit{addr}.() \multimap \mathit{arrayView}(a, 0, l) \\ \mathit{ArraySome} & : & \forall a : \mathit{type}.\forall l : \mathit{addr}.\forall n : \mathit{nat}.(a@l, \mathit{arrayView}(a, n, l+1)) \multimap \mathit{arrayView}(a, n+1, l) \end{array}$$

For instance, the view assigned to *ArraySome* means that an array of size $I + 1$ containing elements of type T is stored at address L if an value of type T is stored at L and an array of size I containing values of type T is stored at $L + 1$.

```
fun getFirst {a:type, n:int, l:addr | n > 0}
   (pf: arrayView (a,n,l) | p: ptr(l)): '(arrayView (a,n,l) | a) =
  let
     prval ArraySome (pf1, pf2) = pf
     // pf1: a@l and pf2: arrayView (a,n-1,l+1)
     val '(pf1' | x) = getVar (pf1 | p)
     // pf1': a@l
  in
     '(ArraySome (pf1', pf2) | x)
  end
```

Fig. 7. A simple function on arrays

We now implement a simple function *getFirst* in Figure 7 that takes the first element in a nonempty array. The header of the function *getFirst* indicates that the following type is assigned to it:

$$\forall a : \mathit{type}.\forall n : \mathit{int}.\forall l : \mathit{addr}.n > 0 \supset ((\mathit{arrayView}(a, n, l) \mid \mathbf{ptr}(l)) \to (\mathit{arrayView}(a, n, l) \mid a))$$

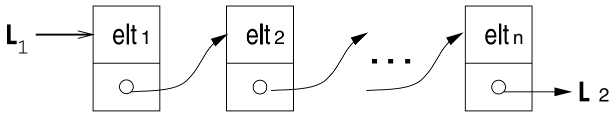
The (unfamiliar) syntax in the body of *getFirst* needs some explanation: *pf* is a proof of the view $\mathit{arrayView}(a, n, l)$, and it must be of the form $\mathit{ArraySome}(\mathit{pf}_1, \mathit{pf}_2)$, where pf_1 and pf_2 are proofs of views $a@l$ and $\mathit{arrayView}(a, n-1, l+1)$, respectively; recall that the function *getVar* is assumed to be of the following type:

$$\forall a : \mathit{type}.\forall l : \mathit{addr}.(a@l \mid \mathbf{ptr}(l)) \to (a@l \mid a)$$

which simply means that applying *getVar* to a pointer of type $\mathbf{ptr}(L)$ requires a proof of $T@L$ for some type T and the application returns a value of type T as well as a proof of $T@L$; thus pf'_1 is also a proof of $a@l$ and $\mathit{ArraySome}(\mathit{pf}'_1, \mathit{pf}_2)$ is a proof of $\mathit{arrayView}(a, n, l)$. In the definition of *getFirst*, we have both code for dynamic computation and code for static manipulation of proofs of views, and the latter is to be erased before dynamic computation starts.

3.2 Singly-Linked Lists

We can declare a dataview for representing singly-linked list segments in Figure 8. Note that we write $(T_0, \cdots, T_n)@L$ for a sequence of views: $T_0@(L + 0), \cdots, T_n@(L + n)$. Given a type T, an integer I and two addresses L_1 and L_2, $slseg(T, I, L_1, L_2)$ is a view for a singly-linked list segment pictured as follows:



where (1) each element in the segment is of type T, (2) the length of the segment is n and (3) the segment starts at L_1 and ends at L_2. A singly-linked list is simply a special kind of singly-linked list segment that ends with a null pointer, and this is clearly reflected in the definition of the view constructor *sllist* presented in Figure 8.

```
dataview slseg (type, int, addr, addr) =
  | {a:type, l:addr} SlsegNone (a, 0, l, l)
  | {a:type, n:nat, first, next, last | first <> null}
    SlsegSome (a, n+1, first, last) of
      ((a, ptr (next)) @ first, slseg (a, n, next, last))

viewdef sllist (a, n, l) = slseg (a, n, l, null)
```

Fig. 8. A dataview for singly-linked list segments

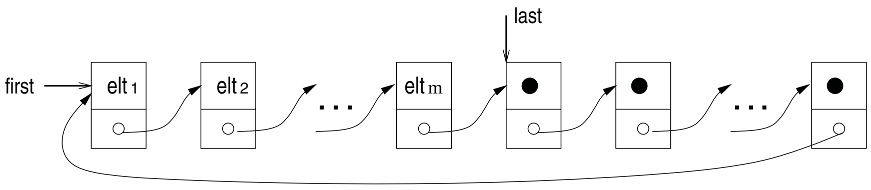
We now present an interesting example in Figure 9. The function *array2sllist* in the upper part of the figure turns an array into a singly-linked list. To facilitate understanding, we also present in the lower part of the figure a corresponding function implemented in C. If we erase the types and proofs in the implementation of *array2sllist* in ATS, then the implementation is tail recursive and tightly corresponds to the loop in the implementation in C. What is remarkable here is that the type system of ATS can guarantee the memory safety of *array2sllist* (even in the presence of pointer arithmetic).

3.3 A Buffer Implementation

We present an implementation of buffers based on linked lists in this section. We first define a view constructor *bufferView* as follows:

```
viewdef bufferView (a:type, m:int, n:int, first: addr, last: addr) =
  '(slseg (a, m, first, last), slseg (top, n-m, last, first))
```

where m and n represent the number of elements stored in a buffer and the maximal buffer size, respectively. For instance, such a buffer can be pictured as follows:



where we use • for uninitialized or discarded content. In the above picture, we see that a buffer of maximal size n consists of two list segments: one with length m, which

```
fun array2sllist {l:addr, n:nat | n >= 1, l <> null}
  (pf: arrayView (top, n+n, l) | p: ptr(l), s: int(n))
  : '(sllist (top, n, l) | unit) =
  if s ieq 1 then
    let
       prval ArraySome (pf0, ArraySome (pf1, ArrayNone)) = pf
       val '(pf1 | _) = setVar (pf1 | p + 1, null)
    in
       '(SlsegSome ('(pf0, pf1), SlsegNone) | '())
    end
  else
    let
       prval ArraySome (pf0, ArraySome (pf1, pf)) = pf
       val '(pf1 | _) = setVar (pf1 | p + 1, p + 2)
       val '(rest | _) = array2sllist (pf | p + 2, s - 1)
    in
       '(SlsegSome ('(pf0, pf1), rest) | '())
    end

//////////////////////////////////////////////////////////////////

/* The following program in C corresponds the above one in ATS */

typedef struct slseg { int val; struct slseg * next; } slseg;

void array2sllist (int* p, int size) {
  int s;

  for (s = size; s > 1; s = s - 1) { *(p+1) = p+2; p = p+2; }

  *(p+1) = 0; /* assign the null pointer */
}
```

Fig. 9. Converting an array into a singly-linked list

contains the values that are currently placed in the buffer, starts at address *first* and ends at *last*, and we call it the *occupied segment*; the other with length $(n - m)$, which contains all free cells in this buffer, starts at *last* and ends at *first*, and we call it *free segment*. The address *first* is often viewed as the head of a buffer.

In Figure 10, we present a function *addIn* that inserts an element into a buffer and another function *takeOut* that removes an element from a buffer. The header of the function *addIn* indicates that the following type is assigned to it,

$$\begin{array}{l}\forall a : \mathit{type}.\forall m : \mathit{nat}.\forall n : \mathit{nat}.\forall l_1 : \mathit{addr}.\forall l_2 : \mathit{addr}.m < n \supset \\ \quad (\mathit{bufferView}(a, m, n, l_1, l_2) \mid a, \mathbf{ptr}(l_2)) \rightarrow \exists l_3 : \mathit{addr}.(\mathit{bufferView}(a, m+1, n, l_1, l_3) \mid \mathbf{ptr}(l_3))\end{array}$$

which simply means that inserting into a buffer requires that the buffer is not full and, if it succeeds, the application increases the length of *occupied segment* by one and returns a new *ending* address for *occupied segment* (a.k.a. the new *starting* address for *free segment*). Similarly, the following type is assigned to the function *takeOut*,

$$\begin{array}{l}\forall a : \mathit{type}.\forall m : \mathit{nat}.\forall n : \mathit{nat}.\forall l_1 : \mathit{addr}.\forall l_2 : \mathit{addr}.m \leq n \wedge m > 0 \supset \\ \quad (\mathit{bufferView}(a, m, n, l_1, l_2) \mid \mathbf{ptr}(l_1)) \rightarrow \exists l_3 : \mathit{addr}.(\mathit{bufferView}(a, m-1, n, l_3, l_2) \mid a, \mathbf{ptr}(l_3))\end{array}$$

which means that removing an element out of a buffer requires that the buffer is not empty and, if it succeeds, the application decreases the length of *occupied segment* by

```
fun addIn {a:type, m: nat, n:nat, first:addr, last:addr | m < n}
    (pf: bufferView (a, m, n, first, last) | x: a, t: ptr(last))
  : [last':addr]
      '(bufferView (a, m+1, n, first, last') | ptr (last')) =
   let
      prval '(pf0, pf1) = pf
      prval SlsegSome ('(pf100, pf101), pf11) = pf1
      val '(pf100 | _) = setVar (pf100 | t, x)
      val '(pf101 | p) = getVar (pf101 | t + 1)
      prval pf0 =
        slsegAppend (pf0, SlsegSome ('(pf100, pf101), SlsegNone))
   in
     '( '(pf0, pf11) | p )
   end

fun takeOut {a:type, m:nat, n:nat, first:addr, last:addr | m>0, n>=m}
    (pf: bufferView (a, m,  n, first, last) | h: ptr(first))
  : [first':addr]
      '(bufferView (a, m-1, n, first', last) | '(a, ptr(first'))) =
   let
      prval '(pf0, pf1) = pf
      prval SlsegSome ('(pf000, pf001), pf01) = pf0
      val '(pf000 | x) = getVar (pf000 | h)
      val '(pf001 | p) = getVar (pf001 | h + 1)
      prval pf1 =
        slsegAppend (pf1, SlsegSome ('(pf000, pf001), SlsegNone))
   in
     '( '(pf01, pf1) | '(x, p) )
   end
```

Fig. 10. Two functions on cyclic buffers

one and returns the element and a new *starting* address for *occupied segment* (a.k.a the new *ending* address for *free segment*). In addition, from the type of function *takeOut*, we can see that there is no need to fix the position of the buffer head and, in fact, the head of a buffer moves along the circular list if we keep taking elements out of that buffer.

The function *slsegAppend* is involved in the implementation of *addIn* and *takeOut*. This is a proof function that combines two list segment views into one list segment view, and it is assigned the following functional view:

$$\begin{array}{l}\forall a : \mathit{type}.\forall n_1 : \mathit{nat}.\forall n_2 : \mathit{nat}.\forall l_1 : \mathit{addr}.\forall l_2 : \mathit{addr}.\forall l_3 : \mathit{addr}.\\ \quad (\mathit{slseg}(a, n_1, l_1, l_2), \mathit{slseg}(a, n_2, l_2, l_3)) \multimap \ \mathit{slseg}(a, n_1 + n_2, l_1, l_3)\end{array}$$

Note that this function is only used for type-checking at compile-time and is neither needed nor available at run-time.

3.4 Other Examples

In addition to arrays and singly-linked lists, we have also handled a variety of other data structures such as doubly-linked lists and doubly-linked binary trees that make (sophisticated) use of pointers. Some of the examples involving such data structures (e.g., a splay tree implementation based on doubly-linked binary trees) can be found on-line [Xi03].

4 Related Work

A fundamental problem in programming is to find approaches that can effectively facilitate the construction of safe and reliable software. In an attempt to address this problem, studies on program verification, that is, verifying whether a given program meets its specification, have been conducted extensively.

Some well-known existing approaches to program verification include model checking (which is the algorithmic exploration of the state spaces of finite state models of systems), program logics (e.g., Floyd-Hoare logic), type theory, etc. However, both model checking and Floyd-Hoare logic are often too expensive to be put into software practice. For instance, although model checking has been used with great success in hardware verification for more than twenty years, its application in software is much less common and the focus is often on verifying programs such as device drivers that are closely related to hardware control. In particular, model checking suffers from problems such as state space explosion and highly non-trivial abstraction and is thus difficult to scale in practice. There are also many cases reported in the literature that make successful use of program logics in program verification. As (a large amount of) theorem proving is often involved, such program verification is often too demanding for general practice.

On the other hand, the use of types in program error detection is ubiquitous. However, the types in programming languages such as ML and Java are often too limited for capturing interesting program invariants. Our work falls naturally in between full program verification, either in type theory or systems such as PVS, and traditional type systems for programming languages. When compared to verification, our system is less expressive but much more automatic. Our work can be viewed as providing a systematic and uniform language interface for a verifier intended to be used as a type system during the program development cycle. Our primary motivation is to all the programmer to express more program properties through types and thus catch more program errors at compile-time.

In Dependent ML (DML), a restricted form of dependent types is proposed that completely separates programs from types. This design makes it rather straightforward to support realistic programming features such as general recursion and effects in the presence of dependent types. Subsequently, this restricted form of dependent types is employed in designing Xanadu [Xi00] and DTAL [XH01] in attempts to reap similar benefits from dependent types in imperative programming. In hindsight, it can be readily noticed that the type systems of Xanadu and DTAL bear a close relation to Floyd-Hoare logic.

Along another line of research, a new notion of types called guarded recursive (g.r.) datatypes is recently introduced [XCC03]. Noting the close resemblance between the restricted form of dependent types (developed in DML) and g.r. datatypes, we immediately initiate an effort to design a unified framework for both forms of types, leading to the design and formalization of the framework *Applied Type System*. To support safe programming with pointers, the framework is further extended with stateful views [Xi03].

Also, the work in [OSSY02] is casually related to this paper as it shares the same goal of ruling out unsafe memory accesses. However, the underlying methodology adopted there is fundamentally different. In contrast to the static approach we take, it essentially relies on run-time checks to prevent dangling pointers from being accessed as well as to detect stray array subscripting.

There have been a great number of research activities on verifying program safety properties by tracking state changes. For instance, Cyclone [JMG$^+$01] allows the programmer to specify safe stack and region memory allocation; both CQual [FTA02] and Vault [FD02] support some form of resource usage protocol verification; ESC [Det96] enables the programmer to state various sorts of program invariants and then employs theorem proving to prove them; CCured [NMW02] uses program analysis to show the safety of mostly unannotated C programs. In [MWH03], we also see an attempt to develop a general theory of type refinements for reasoning about program states.

5 Conclusion

Despite a great deal of research, it is still largely an elusive goal to verify the correctness of programs. Therefore, it is important to identify the properties that can be practically verified for realistic programs. We have shown with concrete examples the use of a restricted form of dependent types combined with stateful views in facilitating program verification in the presence of pointer arithmetic. A large number of automated program verification approaches often focus on verifying sophisticated properties of some particularly chosen programs. We feel that it is at least equally important to study scalable approaches to verifying elementary properties of programs in general programming as we have advocated in this paper.

In general, we are interested in promoting the use of light-weighted formal methods in practical programming, facilitating the construction of safe and reliable software. We have presented some examples in this paper in support of such a promotion, demonstrating a novel approach to safe programming with pointers.

References

[AO91] Krzysztof R. Apt and Olderog, E.-R. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York, 1991. ISBN 0-387-97532-2 (New York) 3-540-97532-2 (Berlin). xvi+441 pp.

[C$^+$86] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986. ISBN 0-13-451832-2. x+299 pp.

[Det96] David Detlefs. An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*, 1996.

[EGP99] E.M.Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[FD02] M. Fahndrich and R. Deline. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 13–24. Berlin, June 2002.

[FTA02] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive Type Qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12. Berlin, June 2002.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[JMG$^+$01] Trevor Jim, Greg Morrisett, Dan Grossman, Mike Hicks, Mathieu Baudet, Matthew Harris, and Yanling Wang. Cyclone, a Safe Dialect of C, 2001. URL `http://www.cs.cornell.edu/Projects/cyclone/`. Available at `http://www.cs.cornell.edu/Projects/cyclone/`.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–226. Uppsala, Sweden, September 2003.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 128–139. London, January 2002.

[OSSY02] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ansi-c compiler: An approach to making c programs secure (progress report). In *International Symposium on Software Security*, volume 2609 of *Lecture Notes in Computer Science*. Springer-Verlag, November 2002.

[Rey02] John Reynolds. Separation Logic: a logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS '02)*, 2002. URL `citeseer.nj.nec.com/reynolds02separation.html`.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235. New Orleans, LA, January 2003.

[XH01] Hongwei Xi and Robert Harper. A Dependently Typed Assembly Language. In *Proceedings of International Conference on Functional Programming*, pages 169–180, September 2001.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. viii+181 pp. pp. viii+189. Available at `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[Xi00] Hongwei Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387. Santo Barbara, CA, June 2000.

[Xi03] Hongwei Xi. Applied Type System, July 2003. Available at: `http://www.cs.bu.edu/~hwxi/ATS`.

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. San Antonio, Texas, January 1999.

Towards Provably Correct Code Generation via Horn Logical Continuation Semantics*

Qian Wang[1], Gopal Gupta[1], and Michael Leuschel[2]

[1] Department of Computer Science
University of Texas at Dallas, Richardson, TX 75083-0688 USA
{qxw015000,gupta}@utdallas.edu
[2] Department of Electronics and Computer Science
University of Southampton, Southampton, UK S017 1BJ
mal@ecs.soton.ac.uk

Abstract. Provably correct compilation is an important aspect in development of high assurance software systems. In this paper we explore approaches to provably correct code generation based on programming language semantics, particularly *Horn logical semantics*, and partial evaluation. We show that the *definite clause grammar (DCG)* notation can be used for specifying both the syntax and semantics of imperative languages. We next show that continuation semantics can also be expressed in the Horn logical framework.

1 Introduction

Ensuring the correctness of the compilation process is an important consideration in construction of reliable software. If the compiler generates code that is not faithful to the original program code of a system, then all our efforts spent in proving the correctness of the system could be futile. Proving that target code is correct w.r.t. the program source is especially important for high assurance systems, as unfaithful target code can lead to loss of life and/or property. Considerable research has been done in this area, starting from the work of McCarthy [18]. Most efforts directed at proving compiler correctness fall into three categories:

– Those that treat the compiler as just another program and use standard verification techniques to manually or semi-automatically establish its correctness (e.g., [3]). However, even with semi-automation this is a very labour intensive and expensive undertaking, which has to be repeated for every new language, or if the compiler is changed.

* The authors have been partially supported by NSF grants CCR 9900320, INT 9904063, and EIA 0130847, by the Department of Education and the Environmental Protection Agency, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project.

– Those that *generate* the compiler automatically from the mathematical semantics of the language. Typically the semantics used is denotational (see for example Chapter 10 of [23]). The automatically generated compilers, however, have not been used in practice due to their slowness and/or inefficiency/poor quality of the code generated.
– Those that use program transformation systems to transform source code into target code [16, 20]. The disadvantage in this approach is that specifying the compiler operationally can be quite a lengthy process. Also, the compilation time can be quite large.

In [6] we developed an approach for generating code for imperative languages in a provably correct manner based on partial evaluation and a type of semantics called *Horn logical semantics*. This approach is similar in spirit to semantics-based approaches, however, its basis is Horn-logical semantics [6] which possesses both an operational as well as a denotational (declarative) flavor. In the Horn logical semantics approach, both the syntax and semantics of a language is specified using Horn logic statements (or pure Prolog).

Taking an operational view, one immediately obtains an interpreter of the language $\mathcal{L}$ from the Horn-logical semantic description of the language $\mathcal{L}$. The semantics can be viewed dually as operational or denotational. Given a program $\mathcal{P}$ written in language $\mathcal{L}$, the interpreter obtained for $\mathcal{L}$ can be used to execute the program. Moreover, given a partial evaluator for pure Prolog, the interpreter can be *partially evaluated* w.r.t. the program $\mathcal{P}$ to obtain compiled code for $\mathcal{P}$. Since the compiled code is obtained automatically via partial evaluation of the interpreter, it is faithful to the source of $\mathcal{P}$, provided the partial evaluator is correct. The correctness of the partial evaluator, however, has to be proven only once. The correctness of the code generation process for *any* language can be certified, provided the compiled code is obtained via partial evaluation. Given that efficient execution engines have been developed for Horn Logic (pure Prolog), partial evaluation is relatively fast. Also, the declarative nature of the Horn logical semantics allows for language semantics to be rapidly obtained.

In this paper, we further develop our approach and show that in Horn logical semantics not only the syntax but also the semantics can be expressed using the definite clause grammar notation. The semantics expressed in the DCG notation allows for the store argument to be naturally (syntactically) hidden. We show that continuation semantics can also be expressed in Horn logic. Continuation semantics model the semantics of imperative constructs such as *goto statements*, *exception handling mechanisms, abort*, and *catch/throw constructs* more naturally. We also show that continuation semantics expressed as DCGs can be partially evaluated w.r.t. a source program to obtain "good quality" target code.

In this work we use partial evaluation to generate target code. Partial evaluation is especially useful when applied to interpreters; in this setting the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialized version of the interpreter, which can be viewed as a compiled version of the object program [5].

In our work we have used the LOGEN system [14]. Much like MIXTUS, LOGEN can handle many non-declarative aspects of Prolog. LOGEN also supports *partially static* data by allowing the user to declare custom "binding types." More details on the LOGEN system can be found elsewhere [14]. Unlike MIXTUS, LOGEN is a so-called *offline* partial evaluator, i.e., specialization is divided into two phases: (i) A *binding-time analysis* (*BTA* for short) phase which, given a program and an approximation of the input available for specialization, approximates all values within the program and generates annotations that steer (or control) the specialization process. (ii) A (simplified) *specialization phase*, which is guided by the result of the *BTA*.

Because of the preliminary BTA, the specialization process itself can be performed very efficiently, with predictable results (which is important for our application). Moreover, due to its simplicity it is much easier to establish correctness of the specialization process.

Finally, while our work is motivated by provably correct code generation, we believe our approach to be useful to develop "ordinary" compilers for domain specific languages in general [8].

2 Horn Logical Semantics

The denotational semantics of a language $\mathcal{L}$ has three components: (i) *syntax specification:* maps sentences of $\mathcal{L}$ to parse trees; it is commonly specified as a grammar in the BNF format; (ii) *semantic algebra:* represents the mathematical objects whose elements are used for expressing the meaning of a program written in the language $\mathcal{L}$; these mathematical objects typically are sets or domains (partially ordered sets, lattices, etc.) along with associated operations to manipulate the elements of the sets; (iii) *valuation functions:* these are functions mapping parse trees to elements of the semantic algebras.

Traditional denotational definitions express syntax as BNF grammars, and the semantic algebras and valuation functions using λ-calculus. In Horn Logical semantics, Horn-clauses (or pure Prolog) and constraints[1] are used instead to specify all the components of the denotational semantics of programming languages [6]. There are three major advantages of using Horn clauses and constraints for coding denotational semantics.

First, the syntax specification *trivially and naturally* yields an executable parser. The BNF specification of a language $\mathcal{L}$ can be quite easily transformed to a *Definite Clause Grammar* (DCG) [24]. The syntax specification[2] written in the DCG notation serves as a parser for $\mathcal{L}$. This parser can be used to parse

[1] Constraints may be used, for example, to specify semantics of languages for real-time systems [7].

[2] A grammar coded as a DCG is syntax specification in the sense that various operational semantics of logic programming (standard Prolog order, tabled execution, etc.) can be used for execution during actual parsing. Different operational semantics will result in different parsing algorithms (e.g., Prolog in recursive descent parsing with backtracking, tabled execution in chart parsing, etc.).

programs written in $\mathcal{L}$ and obtain their parse trees (or syntax trees). Thus, the syntactic BNF specification of a language is easily turned into *executable syntax* (i.e., a parser). Note that the syntax of even context sensitive languages can be specified using DCGs [6].

Second, the semantic algebra and valuation functions of $\mathcal{L}$ can also be coded in Horn-clause Logic. Since Horn-clause Logic or pure Prolog is a declarative programming notation, just like the λ-calculus, the mathematical properties of denotational semantics are preserved. Since both the syntax and semantic part of the denotational specification are expressed as logic programs, they are both executable. These syntax and semantic specifications can be loaded in a logic programming system and executed, given a program written in $\mathcal{L}$. This provides us with an interpreter for the language $\mathcal{L}$. In other words, the *denotation*[3] of a program written in $\mathcal{L}$ is executable. This executable denotation can also be used for many applications, including automated generation of compiled code.

Third, non-deterministic[4] semantics can be given to a language w.r.t. resources (e.g., time, space, battery power) consumed during execution. For example, some operations in the semantic algebra may be specified in multiple ways (say in software or in hardware) with each type of specification resulting in different resource consumption. Given a program and bounds on the resources that can be consumed, only some of the many possible semantics may be viable for that program. Resource bounded partial evaluation [2] can be used to formalize resource conscious compilation (e.g., energy aware compilation) [26] via Horn Logical semantics.

Horn-logical semantics can also be used for automatic verification and consistency checking [6, 7]. We do not elaborate any further since we are not concerned with verification in this paper.

The disadvantage of Horn logical semantics is that it is not denotational in the strict sense of the word because the semantics given for looping constructs is not compositional. The fix operator used to give compositional semantics of looping constructs in λ-calculus cannot be naturally coded in Horn logic due to lack of higher order functions. This, for example, precludes the use of structural induction to prove properties of programs. However, note that even though the semantics is not truly compositional, it is declarative, and thus the fix-point of the logic program representing the semantics can be computed via the standard T_P operator [17]. Structural/fix-point induction can then be performed over this T_P operator to prove properties of programs. Note that even in the traditional λ-calculus approach, the declarative meaning of the fix operator (defined as computing the limit of a series of functions) is given outside the operational framework of the λ-calculus, just as the computation of the $\mathrm{fix}(\mathrm{T}_P)$ in logic programming is outside the operational framework of Horn Clause logic. For partial evaluation, the operational definition of fix, i.e., $\mathrm{fix(F)} = \mathrm{F(fix\ F)}$, is used.

[3] We refer to the denotation of a program under the Horn-logical semantics as its *Horn logical denotation.*

[4] Non-deterministic in the logic programming sense.

In [6] we show how both the syntax and semantics of a simple imperative language (a simple subset of Pascal whose grammar is shown in Figure 1) can be given in Horn Logic. The Horn logical semantics, viewed operationally, automatically yields an interpreter. Given a program P, the interpreter can be partially evaluated w.r.t. P to obtain P's compiled code.

```
Program ::= C.
C ::= C1;C2 |
     loop while B C end while |
     if B then C1 else C2 endif |
     I := E
E ::= N | Identifier | E1 + E2 |
     E1 - E2 | E1 * E2 | (E)
B ::= E1 = E2 | E1 > E2 | E1 < E2
N ::= 0 | 1 | 2 | ... | 9
Identifier ::= w | x | y | z
```

Fig. 1. BNF grammar

A program and its corresponding code generated via partial evaluation using the LOGEN system [14] is shown below. The specialization time is insignificant (i.e., less than 10 ms). Note that the semantics is written under the assumption that the program takes exactly two inputs (found in variables x and y) and produces exactly one output (placed in variable z). *The definitions of the semantic algebra operations are removed, so that unfolding during partial evaluation will stop when a semantic algebra operation is encountered.* The semantic algebra operations are also shown below.

```
 z = 1;               main(A, B, C) :-          while_eval__1(A, B) :-
 w = x;               initialize_store(D),        access(w, A, C),
 loop while w > 0     update(x, A, D, E),         ( C>0 ->
   z = z * y ;        update(y, B, E, F),            access(z, A, D),
   w = w - 1          update(z, 1, F, G),            access(y, A, E),
 end while.           access(x, G, H),               F is D*E,
                       update(w, H, G, I),            update(z, F, A, G),
                       while_eval__1(I, J),           access(w, G, H),
                       K=J,                           I is H-1,
                       access(z, K, C).               update(w, I, G, J),
                                                      while_eval__1(J, B),
                                                    ;   B=A ).
SEMANTIC ALGEBRA:
 initialize_store([(x,0),(y,0),(z,0),(w,0)]).
 access(Id,[(Id,Val)|_ ],Val).     update(Id,NV,[(Id,_)|R],[(Id,NV)|R]).
 access(Id,[_|R],Val) :-           update(Id,NewV,[P|R],[P|R1]) :-
                access(Id,R,Val).                   update(Id,NewV,R,R1).
```

Notice that in the program that results from partial evaluation, only a series of memory `access`, memory `update`, arithmetic and comparison operations are left, that correspond to `load`, `store`, arithmetic, and comparison operations of a machine language. The while-loop, whose meaning was expressed using recursion, will partially evaluate to a *tail-recursive* program. These tail-recursive calls are easily converted to iterative structures using jumps in the target code.

Though the compiled code generated is in Prolog syntax, it looks a lot like machine code. A few simple transformation steps will produce actual machine code. These transformations include replacing variable names by register/memory locations, replacing a Prolog function call by a jump (using a goto) to the code for that function, etc. The code generation process is provably correct, since target code is obtained automatically via partial evaluation. Of course, we need to ensure that the partial evaluator works correctly. However, this needs to be done only once. Note that once we prove the correctness of the partial evaluator, compiled code for programs written in any language can be generated as long as the Horn-logical semantics of the language is given.

It is easy to see that valuation predicate for an iterative structure will always be tail-recursive. This is because the operational meaning of a looping construct can be given by first iterating through the body of the loop once, and then recursively re-processing the loop after the state has been appropriately changed to reflect the new values of the loop control parameters. The valuation predicate for expressing this operational meaning will be inherently tail recursive.

Note also that if a predicate definition is tail recursive, a folding/unfolding based partial evaluation of the predicate will preserve its tail-recursiveness. This allows us to replace a tail recursive call with a simple jump while producing the final assembly code. The fact that tail-recursiveness is preserved follows from the fact that folding/unfolding based partial evaluation can be viewed as algebraic simplification, given the definitions of various predicates. Thus, given a tail recursive definition, the calls in its body will be expanded in-place during partial evaluation. Expanding a tail-recursive call will result in either the tail-recursion being eliminated or being replaced again by its definition. Since the original definition is tail-recursive, the unfolded definition will stay tail recursive. (A formal proof via structural induction can be given [25] but is omitted due to lack of space.)

3 Definite Clause Semantics

Note that in the code generated, the `update` and `access` operations are parameterized on the memory store (i.e., they take an input store and produce an output store). Of course, real machine instructions are not parameterized on store. This store parameter can be (syntactically) eliminated by using the DCG notation for expressing the valuation predicates as well.

All valuation predicates take a store argument as input, modify it per the semantics of the command under consideration and produce the modified store as output [6]. Because the semantic rules are stated declaratively, the store argument "weaves" through the semantic sub-predicates called in the rule. This suggests that we can express the semantic rules in the DCG notation. Thus, we can view the semantic rules as computing the *difference* between the output and the input stores. This difference reflects the effect of the command whose semantics is being given. Expressed in the DCG notation, the store argument is (syntactically) hidden away. For example, in the DCG notation the valuation predicate

```
command(comb(C1, C2), Store, Outstore) :-
          command(C1, Store, Nstore),
          command(C2, Nstore, Outstore).
```

is written as:

```
command(comb(C1, C2)) --> command(C1), command(C2).
```

In terms of difference structures, this rules states that the difference of stores produced by `C1; C2` is the “sum” of differences of stores produced by the command `C1` and `C2`. The rest of the semantic predicates can be rewritten in this DCG notation in a similar way.

```
main(U,V,A) -->
      update(x,U),
      update(y,V),
      update(z,1),
      access(x,F),
      update(w,F),
      while_eval__1,
      access(z,A).
while_eval__1 -->
     (access(w,C),
      {0<C} ->
         access(z,D),
         access(y,E),
         {F is D*E},
         update(z,F),
         access(w,H),
         {I is H-1},
         update(w,I),
         while_eval__1
       ; []).
```

```
main:                while:
 store x U            load w C
 store y V            skipgtz C
 store z 1            jump else
 load x F             load z D
 store w F            load y E
 jump while           mul D E F
end:                  store z F
 load z W             load w H
                      sub1 H I
                      store w I
                      jump while
                     else:
                      noop
                      jump end
```

Fig. 2. Partially evaluated semantics and its assembly code

Expressed in the DCG notation, the semantic rules become more intuitively obvious. In fact, these rules have more natural reading; they can be read as simple rewrite rules. Additionally, now we can partially evaluate this DCG w.r.t. an input program, and obtain compiled code that has the store argument syntactically hidden. The result of partially evaluating this DCG-formatted semantics is shown to the left in Figure 2. Notice that the store argument weaving through the generated code shown in the original partially evaluated code is hidden away. Notice also that the basic operations (such as comparisons, arithmetic, etc.) that appear in the target code are placed in braces in definite clause semantics, so that the two store arguments are not added during expansion to Prolog. The constructs appearing within braces can be regarded as the “terminal” symbols in this semantic evaluation, similar to terminal symbols appearing in square brackets in the syntax specification. In fact, the operations enclosed within braces are the primitive operations left in the residual target code after partial evaluation. Note, however, that these braces can be eliminated by putting wrappers around the primitive operations; these wrappers will have two redundant store arguments that are identical, per the requirements of the DCG notation. Note also that since the LOGEN partial evaluator is oblivious of the DCG notation, the final generated code was cast into the DCG notation manually.

Now that the store argument that was threading through the code has been eliminated, the access/update instructions can be replaced by load/store instructions, tail recursive call can be replaced by a jump, etc., to yield proper assembly code. The assembly code that results in shown to the right in figure 2. We assume that inputs will be found in registers `U` and `V`, and the output will be placed in register `W`. Note that `x, y, z, w` refer to the memory locations allocated for the respective variables. Uppercase letters denote registers. The instruction `load x Y` moves the value of memory location `x` into register `Y`, likewise `store x Y` moves the value of register Y in memory location `x` (on a modern microprocessor, both `load` and `store` will be replaced by the `mov` instruction); the instruction `jump label` performs an unconditional jump, `mul D E F` multiplies the operands `D` and `E` and puts the result in register `F`, `sub1 A B` subtracts 1 from register A and puts the result in register B, while `skipgtz C` instruction realizes a conditional expression (it checks if register `C` is greater than zero, and if so, skips the immediately following instruction).

Note that we have claimed the semantics (e.g., the one given in section 3) to be denotational. However, there are two problems: (i) First, we use the `(p->q;r)` construct of logic programming which has a hidden cut, which means that the semantics predicates are not even declarative. (ii) second, the semantics is not truly compositional, because the semantics of the while command is given in terms of the while command itself. This non-compositionality means that structural induction cannot be applied.

W.r.t. (i) note that the condition in the `->` always involves a relational operator with ground arguments (e.g., `Bval = true`). The negation of such relational expressions can always be computed and the clause expanded to eliminate the cut. Thus, a clause of the form

```
        p(..) :- (Bval = true -> q(...); r(...))
```

can be re-written as

```
        p(..) :- Bval = true, q(...).
        p(..) :- Bval = false, r(...).
```

Note that this does not adversely affect the quality of code produced via partial evaluation.

W.r.t. (ii), as noted earlier, program properties can still be proved via structural induction on the T_P operator, where P represents the Horn logical semantic definition.

Another issue that needs to be addressed is the ease of proving a partial evaluator correct given that a partial evaluator such as LOGEN [14] or Mixtus [22] are complex pieces of software. However, as already mentioned, because of the offline approach the actual specialization phase of LOGEN is quite straightforward and should be much easier to prove correct. Also, because of the predictability of the offline approach, it should also be possible to formally establish that the output of LOGEN corresponds to proper target code[5].

[5] E.g., for looping constructs, the unfolding of the (tail) recursive call has to be done only once through the recursive call to obtain proper target code.

Note that because partial evaluation is done until only the calls to the semantic algebra operation remain, the person defining the semantics can control the type of code generated by suitably defining the semantic algebra. Thus, for example, one can first define the semantics of a language in terms of semantic algebra operations that correspond to operations in an abstract machine. Abstract machine code for a program can be generated by partial evaluation w.r.t. this semantics. This code can be further refined by giving a lower level semantics for abstract machine code programs. Partial evaluation w.r.t. this lower level semantics will yield the lower level (native) code.

4 Continuation Semantics

So far we have modeled only direct semantics [23] using Horn logic. It is well known that direct semantics cannot naturally model exception mechanisms and `goto` statements of imperative programming languages. To express such constructs naturally, one has to resort to continuation semantics. We next show how continuation semantics can be naturally expressed in Horn Clause logics using the DCG notation. In the definite clause continuation semantics, semantics of constructs is given in terms of the *differences of parse trees* (i.e., difference of the input parse tree and the continuation's parse tree) [25]. Each semantic predicate thus relates an individual construct (difference of two parse trees) to a fragment of the store (difference of two stores). Thus, semantic rules are of the form:

```
    command(C1, C2, Program, S1, S2) :- ...
```

where the difference of C1 and C2 (say ΔC) represents the command whose semantics is being given, and the difference of S1 and S2 represents the store which reflects the incremental change (ΔS) brought about to the store by the command ΔC. Note that the `Program` parameter is needed to carry the mapping between labels and the corresponding command. Each semantic rule thus is a stand alone rule relating the difference of command lists, ΔC, to difference of stores, ΔS. *If we view a program as a sequence of difference of command lists then its semantics can simply be obtained by "summing" the difference of stores for each command.* That is, if we view a program P as consisting of sequence of commands:

$P = \Delta C_1 + \Delta C_2 + \ldots + \Delta C_n$

then its semantics S is viewed as a "sum" of the corresponding differences of stores:

$S = \Delta S_1 \oplus \Delta S_2 \oplus \ldots \oplus \Delta S_n$

and the continuation semantics simply maps each ΔC_i to the corresponding ΔS_i. Note that $\oplus$ is a non-commutative operator, and its exact definition depends on how the store is modeled. Additionally, continuation semantics allow for cleaner, more intuitive declarative semantics for imperative constructs such as exceptions, catch/throw, goto, etc. [23].

Finally, note that the above continuation semantics rules can also be written in the DCG notation causing the arguments `S1` and `S2` to become syntactically hidden:

```
    command(C1, C2, Program) --> ...
```

Below, we give the continuation semantics of the subset of Pascal considered earlier after extending it with statement labels and a goto statement. Note that the syntax trees are now represented as a list of commands. Each command is represented in the syntax tree as a pair, whose first element is a label (possibly null) and the second element is the command itself. Only the valuation functions for commands are shown (those for expressions, etc., are similar to the one shown earlier).

```
prog_eval([], _, _, 0) --> []
prog_eval(CommList, Val_x, Val_y, Output) -->
  update(x, Val_x), update(y, Val_y),
  command_eval(CommList,cont([],[]), CommList), access(z, Output).

command_eval([],[],_Program) --> [].
command_eval([],cont(CommList,Cont),Program)-->
  command_eval(CommList,Cont,Program).
command_eval([Comm|CommList],Cont,Program)-->
  comm_eval(Comm,CommList,Cont,NCommList,NCont,Program),
  command_eval(NCommList,NCont,Program).

comm_eval([(_,abort)|_],_Comm,_Cont,[],[],_Program) --> [].
comm_eval((Label,while(B,LoopBody)),OldRest,OldCont,[],[],Program)
   --> bool_while_eval(B,LoopBody,
            cont([(Label,while(B,LoopBody))|OldRest],
                  OldCont), OldRest,OldCont,Program).
comm_eval((_,ce(B,C1,C2)),OldRest,OldCont,[],[],Program) -->
  bool_eval(B,C1,cont(OldRest,OldCont),C2,cont(OldRest,OldCont),Program).
comm_eval((_,ce(B,C1)),OldRest,OldCont,[],[],Program) -->
  bool_eval(B,C1,cont(OldRest,OldCont),OldRest,OldCont,Program).
comm_eval((_,jmp(ID)),_OldRest,_OldCont,JumpList,cont([],[]),Program)-->
  {find_label(ID,Program,JumpList)}.
comm_eval((_,assign(id(I), E)),OldRest,OldCont,OldRest,OldCont,_Program)
   --> expr(E, Val), update(I, Val).

bool_while_eval(Cond,C1,C1Cont,C2,C2Cont,Program) -->
  bool_eval(Cond,C1,C1Cont,C2,C2Cont,Program).
bool_eval(greater(E1, E2),C1,C1Cont,C2,C2Cont,Program)
   --> expr(E1, Eval1), expr(E2, Eval2),
        ({Eval1 > Eval2} -> command_eval(C1,C1Cont,Program) ;
                            command_eval(C2,C2Cont,Program)).
 /*the code for lesser(E1,E2) and equal(E1,E2) is very similar*/
```

The code above is self-explanatory. Semantic predicates pass command continuations as arguments. The code for `find_label/3` predicate is not shown. It looks for the program segment that is a target of a `goto` and changes the current continuation to that part of the code.

Consider the program shown below to the left in Figure 3. In this program segment, control jumps from outside the loop to inside via the goto statement.

The result of partially evaluating the interpreter (after removing the definitions of semantic algebra operations) obtained from the semantics w.r.t. this program (containing a `goto`) is shown in the figure 3 to the right. Figures 4 shows another instance of a program involving goto's and the code generated by the LOGEN partial evaluator by specialization of the definite clause continuation semantics shown above.

```
//source code
z = 1;
w = x;
goto label;
loop while w > 0
    z = z * y ;
    label: w = w - 1
endloop while;
z = 8;
z = 7.
```

```
//generated code
interpreter(A, B, C) -->
    update(x, A),
    update(y, B),
    update(z, 1),
    access(x, D),
    update(w, D),
    access(w, E),
    {F is E-1},
    update(w, F),
    fix1,
    access(z, C).
```

```
fix1 -->
    ( access(w, A),
        {0<A} ->
        access(z, B),
        access(y, C),
        {D is B*C},
        update(z, D),
        access(w, E),
        {F is E-1},
        update(w, F),
        fix1
    ;   update(z, 8),
        update(z, 7)
    ).
```

Fig. 3. Example with a jump from outside to inside a while loop

Note that a Horn logical continuation semantics can be given for any imperative language in such a way that its partial evaluation w.r.t. a program will yield target code in terms of access/update operation. This follows from the fact that programs written in imperative languages consist of a series of commands executed under a control that is explicitly supplied by the programmer. Control is required to be specified to a degree that the continuation of each command can be uniquely determined. Each command (possibly) modifies the store. Continuation semantics of a command is based on modeling the change brought about to the store by the continuation of this command. Looking at the structure of the continuation semantics shown above, one notes that programs are represented as lists of commands. The continuation of each command may be the (syntactically) next command or it might be some other command explicitly specified by a control construct (such as a goto or a loop). The continuation is modeled in the semantics explicitly, and can be explicitly set depending on the control construct. The semantics rule for each individual command computes the changes made to the store as well as the new continuation. Thus, as long as the control of an imperative language is such that *the continuation of each command can be explicitly determined*, its Horn logical continuation semantics can be written in the DCG syntax. Further, since the semantics is executable, given a program written in the imperative language, it can be executed under this semantics. The execution can be viewed as unfolding the top-level call, until all goals are solved. If the definitions of the semantic algebra operations are re-

```
//source code
z = 1;
w = x;
loop while w > 0
    z = z * y ;
    w = w - 1;
    goto label
endloop while;
label: z = 8
z = 7.
```

```
//generated code
interpreter(A, B, C) -->
    update(x, A), update(y, B),
    update(z, 1),
    access(x, D), update(w, D),
    ( access(w, E),
        {0<E} ->
        access(z, F), access(y, G),
        {H is F*G},
        update(z, H),
        access(w, I),
        {J is I-1},
        update(w, J),
        update(z, 8), update(z, 7)
    ;   update(z, 8), update(z, 7)
    ),
    access(z, C).
```

Fig. 4. Example with a jump from inside to outside a while loop

moved, then the top-level call can be simplified via unfolding (partial evaluation) to a resolvent which only contains calls to the semantic algebra operations; this resolvent will correspond to the target code of the program.

It should also be noted that the LOGEN system allows users to control the partial evaluation process via annotations. Annotations are generated by the BTA and then can be modified manually. This feature of the LOGEN system gives considerable control of the partial evaluation process – and hence of the code generation process – to the user. The interpreter has to be annotated only once by the user, to ensure that good quality code will be generated.

5 A Case Study in SCR

We have applied our approach to a number of practical applications. These include generating code for parallelizing compilers in a provably correct manner [6], generating code for controllers specified in Ada [13] and for domain specific languages [8] in a provably correct manner, and most recently generating code in a provably correct manner for the Software Cost Reduction (SCR) framework.

The SCR (Software Cost Reduction) requirements method is a software development methodology introduced in the 80s [9] for engineering reliable software systems. The target domain for SCR is real-time embedded systems. SCR has been applied to a number of practical systems, including avionics system (the A-7 Operational flight Program), a submarine communication system, and safety-critical components of a nuclear power plant [10].

We have developed the Horn logical continuation semantics for the complete SCR language. This Horn logical semantics immediately provides us with an interpreter on which the program above can be executed. Further, the interpreter was partially evaluated and compiled code was obtained. The time taken to obtain compile code using definite clause continuation semantics of SCR was an order of magnitude faster than a program transformation based strategy described in [16] that uses the APTS system [20], and more than 40 times faster than a strategy that associates C code as attributes of parse tree nodes and synthesizes the overall code from it [16].

6 Related Work

Considerable work has been done on manually or semi-mechanically proving compilers correct. Most of these efforts are based on taking a specific compiler and showing its implementation to be correct. A number of tools (e.g., a theorem prover) may be used to semi-mechanize the proof. Example of such efforts range from McCarthy's work in 1967 [18] to more recent ones [3]. As mentioned earlier, these approaches are either manual or semi-mechanical, requiring human intervention, and therefore not completely reliable enough for engineering high-assurance systems. "Verifying Compilers" have also been considered as one of the grand challenge for computing research [11], although the emphasis in [11] is more on developing a compiler that can verify the assertions inserted in programs (of course, such a compiler has to be proven correct first).

Considerable work has also been done on generating compilers automatically from language semantics [23]. However, because the syntax is specified as a (non-executable) BNF and semantics is specified using λ-calculus, the automatic generation process is very cumbersome. The approach outlined in this paper falls in this class, except that it uses Horn logical semantics which, we believe and experience suggests, can be manipulated more efficiently. Also, because Horn logical semantics has more of an operational flavor, code generation via partial evaluation can be done quite efficiently.

Considerable work has also been done in using term rewriting systems for transforming source code to target code. In fact, this approach has been applied by researchers at NRL to automatically generate C code from SCR specification using the APTS [20] program transformation system. As noted earlier, the time taken is considerably more than in our approach. Other approaches that fall in this category include the HATS system [27] that use tree rewriting to accomplish transformations. Other transformation based approaches are mentioned in [16].

Recently, Pnueli et al have taken the approach of verifying a given run of the compiler rather than a compiler itself [21]. This removes the burden of maintaining the compiler's correctness proof; instead each run is proved correct by establishing a refinement relationship. However, this approach is limited to very simple languages. As the authors themselves mention, their approach "seems to work in all cases that the source and target programs each consist of a repeated execution of a single loop body ..," and as such is limited. For such simple languages, we believe that a Horn logical semantics based solution will perform much better and will be far easier to develop. Development of the refinement relation is also not a trivial task. For general programs and general languages, it is unlikely that the approach will work.

Note that considerable work has been done in partially evaluating meta-interpreters for declarative languages, in order to eliminate the interpretation overhead (see, for example, [19, 1]). However, in this paper our goal is to generate assembly-like target code for imperative languages.

7 Conclusions

In this paper we presented an approach based on formal semantics, Horn logic, and partial evaluation for obtaining provably correct compiled code. We showed that not only the syntax specification, but also the semantic specification can be coded in the DCG notation. We also showed that continuation semantics of an imperative language can also be coded in Horn clause logic. We applied our approach to a real world language – the SCR language for specifying real-time embedded system. The complete syntax and semantic specification for SCR was developed and used for automatically generating code for SCR specifications. Our method produces executable code considerably faster than other transformation based methods for automatically generating code for SCR specifications.

Acknowledgments

We are grateful to Constance Heitmeyer and Elizabeth Leonard of the Naval Research Labs for providing us with the BNF grammar of SCR and the safety injection program as well as for discussions, and to the anonymous referees.

References

1. A. Brogi and S. Contiero. Specializing Meta-Level Compositions of Logic Programs. Proceedings LOPSTR'96, J. Gallagher. Springer-Verlag, LNCS 1207.
2. S. Debray. Resource bounded partial evaluation. PEPM 1997. pp. 179-192.
3. A. Dold, T. Gaul, W. Zimmermann Mechanized Verification of Compiler Backends Proc. Software Tools for Technology Transfer, Denmark, 1998.
4. S. R. Faulk. State Determination in Hard-Embedded Systems. Ph.D. Thesis, Univ. of NC, Chapel Hill, NC, 1989.
5. Y. Futamura. Partial Evaluation of Computer Programs: An approach to compiler-compiler. *J. Inst. Electronics and Comm. Engineers, Japan.* 1971.
6. G. Gupta "Horn Logic Denotations and Their Applications," *The Logic Programming Paradigm: A 25 year perspective.* Springer Verlag. 1999:127-160.
7. G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium*, pp. 230-239. Dec. 1997.
8. G. Gupta, E. Pontelli. A Logic Programming Framework for Specification and Implementation of Domain Specific Languages. In *Essays in Honor of Robert Kowalski*, 2003, Springer Verlag LNAI,

9. K. L. Henninger. Specifying software requirements for complex systems: New techniques and their application. IEEE Trans. on Software Engg. 5(1):2-13.
10. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. ACM TOSEM 5(3). 1996.
11. C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. J.ACM, 50(1):63-69. Jan 2003.
12. N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503.
13. L. King, G. Gupta, E. Pontelli. Verification of BART Controller. In *High Integrity Software*, Kluwer Academic, 2001.
14. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
15. M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 20(1):208-258.
16. E. I. Leonard and C. L. Heitmeyer. Program Synthesis from Requirements Specifications Using APTS. Kluwer Academic Publishers, 2002.
17. J. Lloyd. Foundations of Logic Programming (2nd ed). Springer Verlag. 1987.
18. J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. MIT AI Lab Memo, 1967.
19. S. Owen. Issues in the Partial Evaluation of Meta-Interpreters. Proceedings Meta'88. MIT Press. pp. 319–339. 1989.
20. R. Paige. Viewing a Program Transformation System at Work. *Proc. Programming Language Implementation and Logic Programming*, Springer, LNCS 844. 1994.
21. A. Pnueli, M. Siegel, E. Singerman. Translation Validation. *Proc TACAS'98*, Springer Verlag LNCS, 1998.
22. D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. Thesis. 1994. Royal Institute of Tech., Sweden. (available at www.sics.se)
23. D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.
24. L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.
25. Q. Wang, G. Gupta, M. Leuschel. Horn Logical Continuation Semantics. UT Dallas Technical Report. 2004.
26. Q. Wang, G. Gupta. Resource Bounded Compilation via Constrained Partial Evaluation. UTD Technical Report. Forthcoming.
27. V. L. Winter. Program Transformation in HATS. *Software Transformation Systems Workshop*, '99.

A Provably Correct Compiler for Efficient Model Checking of Mobile Processes*

Ping Yang[1], Yifei Dong[2], C.R. Ramakrishnan[1], and Scott A. Smolka[1]

[1] Dept. of Computer Science, Stony Brook Univ., Stony Brook, NY, 11794, USA
{pyang,cram,sas}@cs.sunysb.edu

[2] School of Computer Science, Univ. of Oklahoma, Norman, OK, 73019, USA
dong@cs.ou.edu

Abstract. We present an optimizing compiler for the π-calculus that significantly improves the time and space performance of the MMC model checker. MMC exploits the similarity between the manner in which resolution techniques handle variables in a logic program and the manner in which the operational semantics of the π-calculus handles names by representing π-calculus names in MMC as Prolog *variables*, with distinct names represented by distinct variables. Given a π-calculus process P, our compiler for MMC produces an extremely compact representation of P's symbolic state space as a set of *transition rules*. It also uses AC unification to recognize states that are equivalent due to symmetry.

1 Introduction

The recent literature describes a number of efforts aimed at building practical tools for the verification of concurrent systems using Logic Programming (LP) technology; see e.g. [23, 20, 11, 7, 6]. The basic idea underlying these approaches is to pose the verification problem as one of query evaluation over (constraint) logic programs; once this has been accomplished, the minimal-model computation in LP can be used to compute fixed points over different domains.

Taking this idea one step further, in [26], we developed the MMC model checker for the π-calculus [17], a process algebra in which new channel names can be created dynamically, passed as values along other channels, and then used themselves for communication. This gives rise to great expressive power: many computational formalisms such as the λ-calculus can be smoothly translated into the π-calculus, and the π-calculus provides the semantic foundation for a number of concurrent and distributed programming languages (e.g. [19]). MMC also supports the spi-calculus [1], an extension of the π-calculus for the specification and verification of cryptographic protocols. The treatment of channel names in the π- and spi-calculi poses fundamental problems in the construction of a model checker, which are solved in MMC using techniques based on LP query-evaluation mechanisms as explained below.

* Research supported in part by NSF grants CCR-9876242, CCR-9988155, IIS-0072927, CCR-0205376, CCR-0311512, ARO grants DAAD190110003 and DAAD190110019, and ONR grant N000140110967.

MMC, which stands for the Mobility Model Checker, targets the finite-control subset of the π- and spi-calculi[1] so that model checking, the problem of determining whether a system specification S entails a temporal-logic formula φ, may proceed in a fully automatic, push-button fashion. It uses the alternation-free fragment of the π-μ-calculus, a highly expressive temporal logic, for the property specification language.

MMC is built upon the XSB logic programming system with tabulation [25] and, indeed, tabled resolution played a pivotal role in MMC's development. State-space generation in MMC is performed by querying a logic program, called `trans`, that directly and faithfully encodes π's symbolic operational semantics [15]. The key to this encoding is the similarity between the manner in which resolution techniques (which underlie the query-evaluation mechanism of XSB and other logic-programming systems) handle variables in a logic program and the manner in which the operational semantics of the π-calculus handles names. We exploit this similarity by representing π-calculus names in MMC as Prolog *variables*, with distinct names represented by distinct variables.

Query evaluation in MMC resorts to renaming (alpha conversion) whenever necessary to prevent name capture, and parameter passing is realized via unification. Variables are checked for identity (i.e. whether there is a substitution that can distinguish two variables) whenever names need to be equal, for instance, when processes synchronize. The management of names using resolution's variable-handling mechanisms makes MMC's performance acceptable.

Other than MMC, there have been few attempts to build a model checker for the π-calculus, despite its acceptance as a versatile and expressive modeling formalism. The Mobility Workbench (MWB) [24] is an early model checker and bisimulation checker for π; the implementation of its model checker, however, does not address performance issues. The PIPER system [3] generates CCS processes as "types" for π-calculus processes, and formulates the verification problem in terms of these process types; traditional model checkers can then be applied. This approach requires, however, user input in the form of type signatures and does not appear to be fully automated. In [13], a general type system for the π-calculus is proposed that can be used to verify properties such as deadlock-freedom and race-freedom. A procedure for translating a subset of the π-calculus into Promela, the system modeling language of the SPIN model checker [12], is given in [21]. Spin allows channel passing and new name generation, but may not terminate in some applications that require new name generation where MMC does; e.g. a handover protocol involving two mobile stations.

Problem Addressed and Our Solution: Although MMC's performance is considerably better than that of the MWB, it is still an order of magnitude worse than that of traditional model checkers for non-mobile systems, such as SPIN and the XMC model checker for value-passing CCS [20]. XMC, like MMC, is built on top of the XSB logic-programming engine; despite its high-level implementation in Prolog, benchmarks show that it still exhibits competitive performance [8].

[1] The class of *finite-control* processes are those that do not admit parallel composition within the scope of recursion.

One reason for this is the development of a compiler for XMC that produces compact transition-system representations from CCS specifications [9].

In this paper, we present an optimizing compiler, developed along the lines of [9], for the π- and spi-calculi. Our compiler (Section 3), which uses LP technology and other algorithms developed in the declarative-languages community (such as AC unification), seeks to improve MMC's performance by compiling process expressions into a set of *transition rules*. These rules, which form a logic program, can then be queried by a model checker for generating actual transitions. In contrast to the compiler for value-passing CCS, a number of fundamental compilation issues arise when moving to a formalism where channels can be passed as messages, and communication links can be dynamically created and altered via a technique known as scope extrusion and intrusion. Our approach to dealing with these issues is as follows:

– *Representation of states:* The compiler uses a very compact representation for transition-system states, requiring only a symbol for the control location (i.e. a program counter value) and the valuations of variables that are free and live at that state. It also identifies certain semantically equivalent states that may have different syntactic representations. For instance, process expressions $(\nu x)((\nu y)p(x, y) \mid q(x))$ and $(\nu x)(\nu y)(p(x, y) \mid q(x))$ are considered distinct in [26]. According to the structural congruence rule of the π-calculus, however, these expressions are behaviorally identical and are given the same state representation by the compiler.
– *Determining the scope of names:* Since names can be dynamically created and communicated in the π-calculus, the scope of a name cannot in general be determined at compile time. The compiler therefore generates transition rules containing *tagged* names that allows the scope of a name to be determined at model-checking time, when transitions are generated from the rules.
– *State-space reduction via symmetry:* The compiler exploits the associativity and commutativity (AC) of the parallel composition operator when generating transition rules for the model checker (Section 4). In particular, transition rules may contain AC symbols and the compiler uses AC unification and indexing techniques to realize a form of symmetry reduction, sometimes leading to an exponential reduction in the size of the state space.

Another important aspect of MMC's compiler is that it is *provably correct*: The compiler is described using a syntax-directed notation, which when encoded as a logic program and evaluated using tabled resolution becomes its implementation. Thus the compilation scheme's correctness implies the correctness of the implementation itself. Given the complex nature of the compiler, the ability to obtain a direct, high-level, provably correct implementation is of significant importance, and is a practical illustration of the power of declarative programming.

Our benchmarking results (Section 5) reveal that the compiler significantly improves MMC's performance and scalability. For example, on a handover protocol involving two mobile stations, the original version of MMC runs out of memory while attempting to check for deadlock-freedom, even though 2GB of memory is available in the system. In contrast, MMC with compilation verifies this property in 47.01sec while consuming 276.29MB of memory. In another example, a webserver application, the AC operations supported by the compiler allow MMC to verify a system having more than 20 servers; MMC without compilation could handle only 6 servers. The MMC system with the compiler is available in full source-code form from `http://lmc.cs.sunysb.edu/~mmc`.

2 Preliminaries

MMC: A Model Checker for the π-Calculus. In MMC [26], π-calculus processes are encoded as Prolog terms. Let $\mathcal{A}$ denote the set of prefixes, $\mathcal{P}$ the set of process expressions, and $\mathcal{D}$ the set of process identifiers. Further, let $X, Y, Z \ldots$ range over Prolog variables and $p, q, r, \ldots$ range over process identifiers. The syntax of the MMC encoding of π-calculus processes is as follows:

$$\begin{aligned}
\mathcal{A} &::= \texttt{in}(X,Y) \mid \texttt{out}(X,Y) \mid \texttt{tau} \\
\mathcal{P} &::= \texttt{zero} \mid \texttt{pref}(\mathcal{A},\mathcal{P}) \mid \texttt{nu}(X,\mathcal{P}) \mid \texttt{par}(\mathcal{P},\mathcal{P}) \mid \texttt{choice}(\mathcal{P},\mathcal{P}) \\
&\quad\mid \texttt{match}(X\texttt{=}Y,\mathcal{P}) \mid \texttt{proc}(p(Y_1,Y_2,\ldots,Y_n)) \\
\mathcal{D} &::= \texttt{def}(p(X_1,X_2,\ldots,X_n),\mathcal{P}) \quad \text{where } X_i\text{'s are pairwise distinct}
\end{aligned}$$

Prefixes `in(`X`,`Y`)`, `out(`X`,`Y`)` and `tau` represent input, output and internal actions, respectively. `zero` is the process with no transitions while `pref(`α`,`P`)` is the process that can perform an α action and then behave as process P. `nu(`X`,`P`)` behaves as P and X cannot be used as a channel over which to communicate with the environment. Process `match(`X`=`Y`,`P`)` behaves as P if the names X and Y match, and as `zero` otherwise. The constructors `choice` and `par` represent non-deterministic choice and parallel composition, respectively. The expression `proc(`$p(Y_1,\ldots,Y_n)$`)` denotes a *process invocation* where p is a process name (having a corresponding definition) and $Y_1,\ldots,Y_n$ is a comma-separated list of names that are the actual parameters of the invocation. Each process definition of the form `def(`$p(X_1,\ldots,X_n)$`, `P`)` associates a process name p and a list of formal parameters $X_1,\ldots,X_n$ with process expression P. A formal definition of the correspondence between MMC's input language and the syntax of the π-calculus can be found in [26].

The standard notions of bound and free names (denoted by bn() and fn() respectively) in the π-calculus carry over to the MMC syntax. We use $n(e)$ to denote the set of *all* names in a process expression e. Names bound by a restriction operator in e are called *local* names of e.

In order to simplify the use of resolution procedures to handle names represented by variables, we use the following naming conventions. We say that a process expression is *valid* if all of its bound names are unique and are distinct from its free names. We say that a process definition of the form `def(`N`,`P`)` is valid if P, the process expression on the right-hand side of the definition, is valid. A process definition of the form `def(`N`,`P`)` is said to be *closed* if all free names of P appear in N (i.e. are parameters of the process). In MMC, we require all process definitions to be valid and closed. Note that this does not reduce expressiveness since any process expression can be converted to an equivalent valid expression by suitably renaming the bound names.

The operational semantics of the π-calculus is typically given in terms of a symbolic transition system [17, 15]. The MMC model checker computes symbolic transitions using the relation `trans(`s`,`a`,`c`,`d`)` where s and d represent the source and destination states of a transition, a the action and c a constraint on the names of s under which the transition is enabled. In the original model checker [26], this relation was computed by *interpreting* MMC process expressions: the `trans` relation was a direct encoding of the *constructive* semantics of π-calculus given in [26] which is equivalent to the symbolic semantics of [15].

In MMC, properties are specified using the alternation-free fragment of the π-μ-calculus [5], and the MMC model checker is encoded as the binary predicate `models(`P`,`F`)` which succeeds if and only if a process P satisfies a formula F. The encoding of the model checker is given in [26].

Logic Programs: We assume standard notions of predicate symbols, function symbols, and variables. Terms constructed from these such that predicate symbols appear at (and only at) the root are called *atoms*. The set of variables occurring in a term t is denoted by $vars(t)$; we sometimes use $vars(t_1, t_2, \ldots, t_n)$ to denote the set of all variables in terms t_1, t_2, …, t_n. A *substitution* is a map from variables to terms constructed from function symbols and variables. We use (possibly subscripted) θ, θ' to denote substitutions; the composition of two substitutions θ_1 and θ_2 is denoted by $\theta_1\theta_2$. A term t under substitution θ is denoted by $t\theta$. A *renaming* is a special case of substitution that defines a one-to-one mapping between variables.

Horn clauses are of the form $a_0 :- a_1, \ldots, a_n$ where the a_i are atoms. A *goal* (also called a query) is an atom. *Definite* logic programs are a set of Horn clauses. In this paper, we consider only definite logic programs, and henceforth drop the qualifier “definite”. Top-down evaluation of logic programs based on one of several resolution mechanisms such as SLD, OLD, and SLG [16, 4], determines the substitution(s) under which the goal can be derived from the program clauses. We use $G \overset{\theta}{\Longrightarrow}_P \Box$ to denote the derivation of a goal G over a program P, where θ represents the substitution collected in that derivation.

3 A Compiler for the π-Calculus

In this section, we present our compiler for the MMC model checker. Given a process expression E, it generates a set of *transition rules* from which E's transitions can be easily computed. The number of transition rules generated for an expression E is typically much smaller than the number of transitions in E's state space. More precisely, the number of transition rules generated for an expression E is polynomial in the size of E even in the worst case, while the number of transitions in E's state space may be exponential in the size of E.

The rules generated by the compiler for a given MMC process expression E define E's *symbolic transition system*, and are represented using a Prolog predicate of the form `trans(`s`,`a`,`c`,`d`)` where s and d are the transition's source and destination *states*, a is the *action* taken, and c is a *constraint* on the names of s under which the transition is enabled. Although the clauses of the definition of `trans` resemble facts, the constraints c that appear in them can be evaluated only at run-time, and hence encode rules.

The representation used for states is as follows. If E is a *sequential* process expression (i.e. does not contain a parallel composition operator) then it is represented by a Prolog term of the form $\mathtt{s}_i(\overline{V})$ where $\overline{V}$ are the free variables in E and $\mathtt{s}_i$ represents the control location (analogous to a program counter) of E. For instance, let E_1 be the MMC process expression `pref(in(X,Z),pref(out(Z,Y), zero))`. Names `X` and `Y` are free in E_1 and `Z` is bound in E_1. The symbolic state corresponding to E_1 is then given by a Prolog term of the form `s1(X,Y)`, where `s1` denotes the control state. Observe that E_1 can make an `in(X,Z)` action and become E_1', where E_1' =`pref(out(Z,Y), zero)`. The state corresponding to E_1' is a term of the form `s2(Z,Y)`. The symbolic transition from E_1 to E_1' becomes the clause `trans(s1(X,Y),in(X,Z),true,s2(Z,Y))`.

If E is a *parallel* expression of the form `par(`E_1`,`E_2`)` then it is represented by a term of the form `prod(`s_i`,`s_j`)` where s_i and s_j are the states corresponding to E_1 and E_2, respectively. For example, let E_2 =`pref(out(U,V),pref(in(V,W), zero))`, and let `s3(U,V)` and `s4(V)` be the states corresponding to E_2 and `pref(in(V,W), zero))`, respectively. Letting E_1 be defined as above, then the state corresponding to E is `prod(s1(X,Y),s3(U,V))`.

Observe that E can perform a `tau` action and become process E' = `par(pref(out(V,Y), zero),pref(in(V,W),zero))` whenever the free names `X` of E_1 and `U` of E_2 are the same. Such a transition can then be represented by a Horn clause or *rule* of the form:

```
trans(prod(s1(X,Y),s3(U,V)),tau,X=U,prod(s2(V,Y),s4(V)))
```

where the constraint `X=U` is the condition under which the transition is enabled.

Further observe that in E, subprocess E_1 is capable of an autonomous (non-synchronous) transition, taking E from `par(`E_1`,`E_2`)` to `par(`E_1'`,`E_2`)`. Such a transition can be represented by a rule of the form:

```
trans(prod(s1(X,Y),P),in(X,Z),true,prod(s2(Z,Y),P)).
```

where `P` is a variable that ranges over states; thus transition rules may specify a set of states using *state patterns* rather than simply individual states.

One of the challenges we encountered in developing a compiler for MMC concerned the handling of scope extrusion in the π-calculus. In MMC without compilation [26], local names can be determined when computing transitions and hence scope extrusion was implemented seamlessly using Prolog's unification mechanism. However, at compilation time, it may be impossible to determine whether a name is local. For instance, it is not clear if y is a local name in process $x(y).\overline{x}y$ before the process receives an input name. Intuitively, we can solve this problem by carrying local names explicitly in the states of the `trans` rule. This approach, however, introduces a significant amount of overhead when invoking the `trans` to compute the synchronization between two processes. Further, if we do not know for certain whether a name is local, we must also carry a constraint within the rule.

In order to handle scope extrusion efficiently, we propose the following solution. We present names in MMC in one of two forms: either as plain Prolog variables (as in the above examples), or as terms of the form `name(`Z`)` where Z is a Prolog variable. Names of the latter kind are used to represent *local* names, generated by the restriction operator, whereas names of the former kind represent all others. This distinction is essential since we expand the scope of restricted names using the structural congruence rule $(\nu x)P|Q \equiv (\nu x)(P|Q)$ whenever $x \notin \mathrm{fn}(Q)$; this expansion process lets us consolidate the pairs of rules Open and Prefix and Close and Com in the semantics of the π-calculus into single rules. This distinction also enables us to quickly check whether a name is restricted without explicitly keeping track of the environment.

3.1 Compilation Rules

Definition 1 (State Assignment) A *state-assignment function* σ maps process expressions to *positive* integers such that for any two valid expressions E and E', $\sigma(E) = \sigma(E')$ if and only if E and E' are variants of each other (i.e. identical modulo names of variables).

Definition 2 (State Generation) Given a state-assignment function σ, the *state generation function* $\Psi_\sigma(\ )$ maps a process expression E to a state as follows:

$$\Psi_\sigma(E) = \begin{cases} \texttt{state}_0 & \text{if } E = \texttt{zero} \\ \texttt{prod}(\Psi_\sigma(E_1), \Psi_\sigma(E_2)) & \text{if } E = \texttt{par}(E_1, E_2) \\ \Psi_\sigma(E_1[\texttt{name}(V)/X]) & \text{if } E = \texttt{nu}(X, E_1) \\ \quad \text{where } V \notin \mathrm{n}(E_1) & \\ \texttt{state}_{\sigma(E)}(\mathrm{fn}(E)) & \text{otherwise} \end{cases}$$

For each process expression E, MMC's compiler recursively generates a set of transition rules; i.e., E's transition rules are produced based on the transition rules of its subexpressions. The following operation over sets of transition rules is used in defining the compilation procedure:

Definition 3 (Source State Substitution) Given a set of transition rules R, a pair of states s and s', and a constraint C, the *source-state substitution* of R, denoted by $R_{[s \leftarrow s'; C]}$, is the set of transition rules

$$\{\texttt{trans}(s', a, (c, C), d) | \texttt{trans}(s, a, c, d) \in R\},$$

i.e. the set of rules obtained by first selecting rules whose source states unify with s, replacing s by s' in the source state, and adding constraint C to the condition part of the selected rules. If C is empty (i.e. `true`) then we denote the substitution simply by $R_{[s \leftarrow s']}$.

The transition rules generated for a process can be viewed as an automaton, and source-state substitution can be viewed as an operation that replaces the start state of a given automaton with a new state.

Expression E	**Transition Rules** $[\![E]\!]$
`zero`	$\emptyset$
$\texttt{pref}(\alpha, E_1)$	$[\![E_1]\!] \cup \{\texttt{trans}(\Psi_\sigma(E), \alpha, \texttt{true}, \Psi_\sigma(E_1))\}$
$\texttt{choice}(E_1, E_2)$	$[\![E_1]\!] \cup [\![E_2]\!] \cup [\![E_1]\!]_{[\Psi_\sigma(E_1) \leftarrow \Psi_\sigma(E)]} \cup [\![E_2]\!]_{[\Psi_\sigma(E_2) \leftarrow \Psi_\sigma(E)]}$
$\texttt{nu}(X, E_1)$	$[\![E_1[\texttt{name}(V)/X]]\!] \quad V \notin \mathrm{n}(E_1)$
$\texttt{match}(C, E_1)$	$[\![E_1]\!] \cup ([\![E_1]\!]_{[\Psi_\sigma(E_1) \leftarrow \Psi_\sigma(E); C]})$
$\texttt{par}(E_1, E_2)$	$\{\texttt{trans}(\texttt{prod}(s_1, V_2), a, c, \texttt{prod}(d_1, V_2)) \mid \texttt{trans}(s_1, a, c, d_1) \in [\![E_1]\!]\}$ $\cup \{\texttt{trans}(\texttt{prod}(V_1, s_2), a, c, \texttt{prod}(V_1, d_2)) \mid \texttt{trans}(s_2, a, c, d_2) \in [\![E_2]\!]\}$ $\cup \{\texttt{trans}(\texttt{prod}(s_1, s_2), \texttt{tau}, (c_1, c_2, c), \texttt{prod}(d_1, d_2)\theta) \mid \texttt{trans}(s_1, a_1, c_1, d_1) \in [\![E_1]\!] \wedge \texttt{trans}(s_2, a_2, c_2, d_2) \in [\![E_2]\!] \wedge \mathit{vars}(s_1, a_1, c_1, d_1) \cap \mathit{vars}(s_2, a_2, c_2, d_2) = \emptyset \wedge c = (u_1 == u_2) \wedge \theta = \mathit{mgu}(v_1, v_2)$ where $\{a_1, a_2\} = \{\texttt{in}(u_1, v_1), \texttt{out}(u_2, v_2)\} \wedge (c_1, c_2, c)$ is satisfiable$\}$
$\texttt{proc}(p(\vec{v}))$	$[\![E_1[\vec{v}/\vec{X}]]\!] \cup [\![E_1[\vec{v}/\vec{X}]]\!]_{[\Psi_\sigma(E_1[\vec{v}/\vec{X}]) \leftarrow \Psi_\sigma(E)]}$ where $\texttt{def}(p(\vec{X}), E_1)$ is a variant of a definition s.t. $\mathrm{bn}(E_1) \cap \vec{v} = \emptyset$.

Fig. 1. Compilation rules for MMC.

Definition 4 (Compilation Function) Given a state-assignment function σ, the compilation function $[\![\cdot]\!]$ maps MMC process expressions to sets of transition rules such that for any process expression E, $[\![E]\!]$ is the smallest set that satisfies the equations of Figure 1.

The salient points of the compilation rules are as follows:

- In contrast to the CCS compiler [9], control states `entry` and `exit` are not included in the π-calculus compilation rules. Instead, these states are uniquely determined by the process expressions. This also avoids the generation of internal i-transitions in the compilation rules.
- The rules for `pref`, `choice`, `match`, and `proc` can be seen as direct encodings of the corresponding inference rules in Lin's symbolic semantics [15].
- The compilation rule for `nu` specifies that the transition rules of $\texttt{nu}(X, E)$ are identical to the transition rules of E where free occurrences of X have been replaced with a fresh local name $\texttt{name}(V)$. Note that transitions of $\texttt{nu}(X, E)$ can be computed by discarding all transitions of E whose action is over channel X. This effect is achieved by considering at model-checking time only those transitions that are not over channels with local names. Additionally, a local name becomes global if it is output along a global channel using the `Open` rule. Thus the scope of names can only be completely determined at model-checking time, when transitions are generated, and not at compile time when transition *rules* are generated. Hence transition rules assume that every local name can eventually become global, and we check for locality of names when transitions are generated.

– The compilation rule for `par` precomputes, conservatively, all possible synchronizations between the parallel components. In general, we can determine whether two actions are complementary only when the binding of names is known; hence we generate rules for `tau` transitions guarded by constraints that are evaluated at model-checking time.

3.2 Proof of the Compiler's Correctness

We show that MMC's interpreted transition relation (henceforth called `INT`) given in [26] is sound and complete with respect to the transition relation produced by the compiler.

Definition 5 A transition from state s_1 to s_2 with action a under constraint c is said to be *derivable* from a logic program $\mathcal{P}$ (denoted by $s_1 \xrightarrow{a,c}_{\mathcal{P}} s_2$) if `trans(`$s_1, X, Y, Z$`)` $\overset{\theta}{\Longrightarrow}_{\mathcal{P}} \Box$ (i.e. the query succeeds with answer θ) and there is a renaming ρ such that $X\theta\rho = a$, $Y\theta\rho \equiv c$ and $Z\theta\rho = s_2$.

A transition from state s to state s' where the action does not contain local names is denoted by $s \longmapsto_{\mathcal{P}} s'$; a sequence of zero or more such transitions is denoted by $s \overset{*}{\longmapsto}_{\mathcal{P}} s'$.

The soundness and completeness proofs make use of the following fact.

Lemma 1 (Extensionality) Let p and q be valid process expressions such that $p \longrightarrow_{\mathtt{INT}} q$. Then any transition from q derived using the rules compiled from q can also be derived using the rules compiled from p and vice versa. That is, $\Psi_\sigma(q) \xrightarrow{a,c}_{[\![q]\!]} \Psi_\sigma(q')$ iff $\Psi_\sigma(q) \xrightarrow{a,c}_{[\![p]\!]} \Psi_\sigma(q')$.

The proof is by induction on the number of steps needed to derive a transition in $\Psi_\sigma(q)$.

The following lemma asserts that the transitions from an initial state derivable from the compiled transition relation can also be derived using `INT`.

Lemma 2 Let p be a valid process expression. Then $p \xrightarrow{a,c}_{\mathtt{INT}} q$ (i.e. expression p becomes q after action a according to `INT`) whenever $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![p]\!]} \Psi_\sigma(q)$ and a does not contain local names of p.

The proof is by induction on the number of steps needed to derive the transition from p using $[\![p]\!]$.

Theorem 3 (Soundness) Let e be a valid process expression and $e \overset{*}{\longmapsto}_{\mathtt{INT}} p$. Then $p \xrightarrow{a,c}_{\mathtt{INT}} q$ whenever $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![e]\!]} \Psi_\sigma(q)$ and a does not contain local names of p.

The soundness theorem follows from Lemmas 1 and 2.

The following lemma asserts that the transitions from an initial state derivable from `INT` can be derived using the compiled transition relation.

Lemma 4 Let p be a valid process expression. If $p \xrightarrow{a,c}_{\mathtt{INT}} q$ (i.e. expression p become q after action a according to INT) then $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![p]\!]} \Psi_\sigma(q)$.

The proof is by induction on the number of steps needed to derive the transition from p using INT.

Theorem 5 (Completeness) Let e be a valid process expression and $e \overset{*}{\longmapsto}_{\mathtt{INT}} p$. If $p \xrightarrow{a,c}_{\mathtt{INT}} q$ then $\Psi_\sigma(p) \xrightarrow{a,c}_{[\![e]\!]} \Psi_\sigma(q)$.

The completeness theorem follows from Lemmas 1 and 4.

Implementation: The MMC compiler is implemented as a logic program that directly encodes the compilation rules of Figure 1. The use of *tabled resolution* makes such an implementation feasible, ensuring that each process expression in the program is compiled only once. Furthermore, tabling ensures that recursive process definitions can be compiled without extra control. More importantly, the direct implementation means that the correctness of the implementation follows from the correctness of the compilation rules. The implementation also uses partial evaluation to optimize the set of transition rules generated. The application of this optimization is straightforward, and is clearly reflected in the compiler's source code.

4 State-Space Reduction Using AC Unification/Matching

One source of inefficiency in using the compiled transition relation (also in the interpreter INT) is the treatment of the product operator `prod`. In particular, it does not exploit the fact that `prod` is associative and commutative (AC); i.e., `prod(`s_1`, `s_2`)` is semantically identical to (has the same transitions as) `prod(`s_2`,`s_1`)`, and `prod(`s_1`,prod(`s_2`,`s_3`))` is semantically identical to `prod(prod(`s_1`,`s_2`),`s_3`)`. Thus, treating `prod` as an AC operator and using AC unification [22] during resolution will undoubtedly result in a reduction in the number of states that need to be examined during model checking.

AC matching and unification algorithms are traditionally viewed as prohibitively expensive in programming systems. Since, however, `prod` occurs only at the top-most level in terms representing states, a particularly efficient procedure for AC unification during clause selection can be attained. As is done in extant AC unification procedures (see, e.g. [14]), state terms are kept in a canonical form by defining an order among terms with non-AC symbols, and a term of the form `prod(`s_1`,prod(`s_2`,`s_3`))` is represented as the term `prod([`s_1`,`s_2`,`s_3`])` where [s_1,s_2,s_3] is a list whose component states occur in the order defined over non-AC terms.

State-space reduction is achieved by treating `prod` as an AC symbol and this can be seen as a form of *symmetry reduction* [10]. The state patterns generated at compile time are kept in canonical form. At model-checking time, the states derived from these patterns are rewritten (if necessary) to maintain canonical

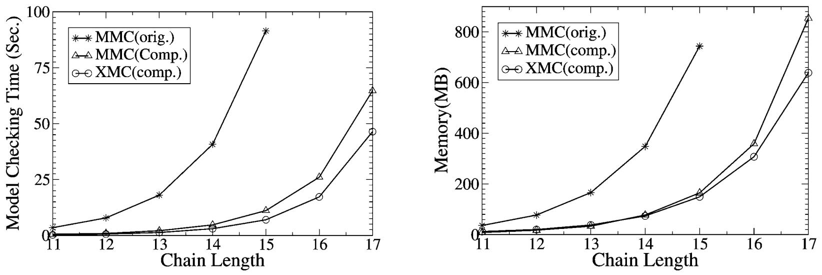


Fig. 2. Effect of compilation on chains of one-place buffers.

forms. Apart from generating fewer states at model-checking time, the compiler generates fewer transition rules when using AC unification. For example, consider the term E =`par(`E_1`,`E_2`)`. For E's autonomous transitions, the non-AC compiler generates rules of the form

$$\begin{array}{ll} & \{\texttt{trans(prod(}s\texttt{,}V\texttt{),}a\texttt{,}c\texttt{,prod(}d\texttt{,}V\texttt{))} \mid \{\texttt{trans(}s\texttt{,}a\texttt{,}c\texttt{,}d\texttt{)} \in [\![E_1]\!]\} \\ \cup & \{\texttt{trans(prod(}V\texttt{,}s\texttt{),}a\texttt{,}c\texttt{,prod(}V\texttt{,}d\texttt{))} \mid \{\texttt{trans(}s\texttt{,}a\texttt{,}c\texttt{,}d\texttt{)} \in [\![E_2]\!]\} \end{array}$$

while the AC compiler generates $[\![E_1]\!] \cup [\![E_2]\!]$. Since rules common to $[\![E_1]\!]$ and $[\![E_2]\!]$ occur only once in $[\![E]\!]$, the number of transition rules for E is reduced.

The use of AC unification can lead to an exponential reduction in the size of the state space. Even in examples that do not display explicit symmetry, state-space reductions by factors of two or more can be seen (Section 5). The number of transition rules generated is also reduced, by more than a factor of two in most examples.

The AC unification operation itself is currently programmed in Prolog and is considerably more expensive than Prolog's in-built unification operation. As will be seen in Section 5, the overhead due to AC unification can be reduced (by a factor of five or more) through the use of AC matching and indexing techniques based on *discrimination nets* [2].

5 Performance Results

We used several model-checking benchmarks to evaluate the performance of the MMC compiler. All reported results were obtained on an Intel Xeon 1.7GHz machine with 2GB RAM running Debian GNU/Linux 2.4.21 and XSB version 2.5 (with slg-wam and local scheduling and without garbage collection).

Benchmark 1: Chains of one-place buffers. This example was chosen for three reasons. (1) We can use it to easily construct large state spaces: a chain of size i has a state space of size $O(2^i)$. (2) The example is structured such that any performance gains due to compilation are due strictly to the compact state representation, thereby allowing us to isolate this optimization's effect. (3) This example does not involve channel passing, thereby enabling us to see how MMC with compilation compares performance-wise to XMC.

The graphs of Figure 2 show the time and space requirements of the original MMC model checker, MMC with compiled transition rules, and the XMC

Table 1. (a)Number of states and transitions for variants of Handover protocol. (b) Performance of MMC for model checking variants of Handover protocol.

(a)

Instance	States			Transitions		
	Orig	Comp	AC	Orig	Comp	AC
1bsp	104	58	29	164	86	43
2bsp	607	408	76	1033	636	130
3bsp	3373	2304	224	5725	3600	416
2ms	N/A	73344	5026	N/A	227712	15461

(b)

Instance	Prop.	Time (Sec.)				Memory (MB)			
		Orig	Comp	AC	AC-net	Orig	Comp	AC	AC-net
1bsp	df	0.04	0.09	0.10	0.09	1.28	1.58	1.37	1.26
	ndl	0.04	0.07	0.15	0.09	1.42	1.60	1.39	1.27
2bsp	df	0.55	0.31	0.44	0.31	7.46	4.02	2.12	1.99
	ndl	0.86	0.31	0.42	0.30	9.69	4.44	2.21	2.06
3bsp	df	4.69	1.04	1.20	0.74	49.30	10.15	2.78	2.60
	ndl	6.90	1.13	1.21	0.76	64.77	16.51	3.06	2.82
2ms	df	N/A	47.01	75.64	33.72	N/A	276.29	24.70	24.47
	ndl	N/A	54.45	80.17	31.93	N/A	340.31	30.69	24.85

system (with compiled transition rules) to verify deadlock freedom in chains of varying length. As expected, MMC with compilation outperforms the original MMC model checker both in terms of time and space. Moreover, MMC with compilation approaches the time and space performance of XMC: the mechanisms needed to handle channel passing in MMC appear to impose an overhead of about 20% in time and 40% in space.

Benchmark 2: Handover procedure. Table 1(a) gives the number of states and transitions generated by three versions of MMC for four variants of the handover procedure of [18]; the four versions differ in the number of passive base stations (bsp) and mobile stations (ms). The column headings "Orig", "Comp", and "AC" refer to the original version of MMC, MMC with compilation, and MMC with compilation and AC reduction, respectively. The results show that MMC with compilation and AC reduction generates the fewest number of states and transitions whereas MMC without compilation generates the most. This is due to the fact that the performance of MMC with compilation is insensitive to the placement of the ν operator.

Table 1(b) presents the time and memory needed to verify the deadlock-freedom (df) and no-data-lost (ndl) properties of the handover protocol. Column heading "AC-net" refers to the version of MMC with AC discrimination nets; the other column headings are as in Table 1(a). Observe that MMC with compilation is more efficient and has superior scalability compared to MMC without compilation. Also observe that the use of AC unification reduces the number of states visited by up to a factor of 20 and space usage by a similar factor, although a concomitant increase in CPU time can be seen. The use of

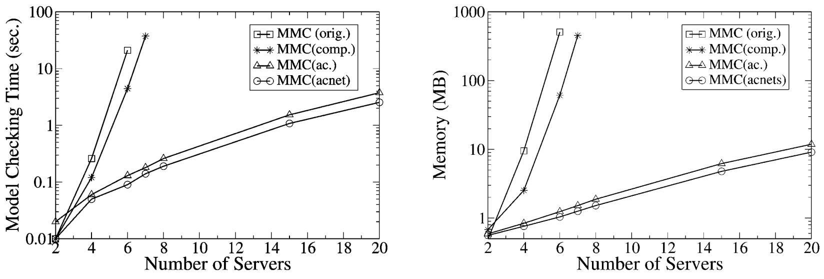


Fig. 3. Effect of AC-based symmetry reduction on chains of web-servers.

AC discrimination nets, however, mitigates this overhead by reducing the number of AC unification operations attempted, resulting in uniformly better time and space performance compared to all other schemes. Note that in the current implementation, both the AC unification and indexing operations are written as Prolog predicates while the non-AC unification and indexing operations use the primitives provided in the Prolog engine. Engine-level support for AC unification and indexing will result in further improvements in performance.

Benchmark 3: Variable-length chains of webservers (from [3]). This example models a file reader of a webserver. The file is divided into several blocks and each block is read and transmitted over the network by a separate process. Blocks can be read in parallel but are required to be transmitted in sequential order. Our AC-unification-based state-space reduction technique applied to this benchmark results in a state space that grows quadratically with the length of the chain, while non-AC techniques (compiled or original) result in a state space that grows exponentially in size. Figure 3 shows the time and memory requirements when the "order-preserved" property is verified on this example. MMC with compilation and AC unification performs best in terms of time, space, and scalability. Note that independent of the number of servers, the AC compiler generates the same number of transition rules (65). The discrimination net-based indexing improves the time and space performance even further.

Benchmark 4: Security protocols specified using the spi-calculus. Table 2(a) gives the number of states and transitions generated by three versions of MMC for three security protocols specified in the spi-calculus: Needham-Schroeder, Needham-Schroeder-Lowe and BAN-Yahalom. Observe that MMC with compilation and AC generates the fewest number (or the same as MMC with compilation) of states and transitions, whereas MMC without compilation generates the most. As mentioned above, this is because MMC without compilation is sensitive to the placement of ν operator. Table 2(b) gives the time (as $x + y$ where x is the compilation time and y is the model-checking time) and memory consumed when model checking these protocols. Compilation (with or without AC discrimination nets) yields an order of magnitude of improvement in time usage and a factor of 10-35% improvement in memory usage. The performance of MMC with AC is similar to that of MMC with AC discrimination nets and is not given in the table.

Table 2. (a) Number of states and transitions for the spi-calculus examples. (b) Performance of MMC for model checking the spi-calculus examples.

(a)

Benchmark	States			Transitions		
	Orig	Comp	AC-net	Orig	Comp	AC-net
Needham-Schroeder	167	164	164	287	282	282
Needham-Schroeder-Lowe	108	105	105	181	176	176
BAN-Yahalom	29133	6674	2011	107652	18106	5322

(b)

Benchmark	Prop.	Time (Sec.)			Memory (MB)		
		Orig	Comp	AC-net	Orig	Comp	AC-net
Needham-Schroeder	attack	0.02	0.07+0.01	0.09+0.02	0.70	1.03	1.12
Needham-Schroeder -Lowe	no attack	0.22	0.08+0.01	0.11+0.02	1.93	1.16	1.23
BAN-Yahalom	interleaving attack	0.11	0.16+0.01	0.18+0.01	2.15	1.89	1.67
	replay attack	0.14	0.16+0.00	0.18+0.01	2.88	1.78	1.65

6 Conclusions

We have shown that an optimizing compiler for the π- and spi-calculi can be constructed using logic-programming technology and other algorithms developed in the declarative-languages community. Extensive benchmarking data demonstrate that the compiler significantly improves the performance and scalability of the MMC model checker. The compiler is equipped with a number of optimizations targeting the issues that arise in a modeling formalism where channels can be passed as messages, and communication links can be dynamically created and altered via scope extrusion and intrusion. Of particular interest is the new symmetry-reduction technique that we have seamlessly integrated into the compiler though the use of AC-unification in resolution.

We are currently investigating techniques that adaptively apply the expensive AC operations only when necessary. We are also in the process of identifying conditions under which channels can be statically named, reducing the overhead incurred when most channels are fixed and only a few are mobile. This would enable us to tightly integrate the XMC system with MMC, paying the price for mobility only when channel names are dynamically created and communicated.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of CCS*, pages 36–47. ACM Press, 1997.
2. L. Bachmair, T. Chen, and I.V. Ramakrishnan. Associative-commutative discrimination nets. In *TAPSOFT*, pages 61–74, 1993.
3. S. Chaki, S.K.Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *Proceedings of POPL*, pages 45 – 57, 2002.
4. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

5. M. Dam. Proof systems for pi-calculus logics. *Logic for Concurrency and Synchronisation*, 2001.
6. G. Delzanno and S. Etalle. Transforming a proof system into Prolog for verifying security protocols. In *LOPSTR*, 2001.
7. G. Delzanno and A. Podelski. Model checking in CLP. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems*, 1999.
8. Y. Dong, X. Du, G. Holzmann, and S. A. Smolka. Fighting livelock in the i-Protocol: A case study in explicit-state model checking. *Software Tools for Technology Transfer*, 4(2), 2003.
9. Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE*, pages 241–256, 1999.
10. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.
11. G. Gupta and E. Pontelli. A constraint based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symposium*, 1997.
12. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
13. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1–3):121–163, 2004.
14. H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories. *J. Functional Prog.*, 11(2):207–251, 2001.
15. H. Lin. Symbolic bisimulation and proof systems for the π-calculus. Technical report, School of Cognitive and Computer Science, U. of Sussex, UK, 1994.
16. J. W. Lloyd. *Foundations of Logic Programming.* Springer, 1984.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.
18. F. Orava and J. Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.
19. B. C. Pierce and D. N. Turner. Pict: a programming language based on the pi-calculus. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
20. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of CAV*. Springer, 1997.
21. H. Song and K. J. Compton. Verifying pi-calculus processes by Promela translation. Technical Report CSE-TR-472-03, Univ. of Michigan, 2003.
22. M. E. Stickel. A unification algorithm for associative-commutative unification. *Journal of the ACM*, 28(3):423–434, 1981.
23. L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP)*, 1996.
24. B. Victor. The Mobility Workbench user's guide. Technical report, Department of Computer Systems, Uppsala University, Sweden, 1995.
25. XSB. The XSB logic programming system v2.5, 2002. Available under GPL from `http://xsb.sourceforge.net`.
26. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the π-calculus: Model checking mobile processes using tabled resolution. In *Proceedings of VMCAI*, 2003. Extended version in *Software Tools for Technology Transfer*, 6(1):38-66,2004.

An Ordered Logic Program Solver

Davy Van Nieuwenborgh*, Stijn Heymans, and Dirk Vermeir**

Dept. of Computer Science
Vrije Universiteit Brussel, VUB
Pleinlaan 2, B1050 Brussels, Belgium
Telephone: +32495939799, Fax: +3226293525
{dvnieuwe,sheymans,dvermeir}@vub.ac.be

Abstract. We describe the design of the OLPS system, an implementation of the preferred answer set semantics for ordered logic programs. The basic algorithm we propose computes the extended answer sets of a simple program using an intuitive 9-valued lattice, called $\mathbf{T}_9$. During the computation, this lattice is employed to keep track of the status of the literals and the rules while evolving to a solution. It turns out that the basic algorithm needs little modification in order to be able to compute the preferred answer sets of an ordered logic program. We illustrate the system using an example from diagnostic reasoning and we present some preliminary benchmark results comparing OLPS with existing answer set solvers such as SMODELS and DLV.

Keywords: Preference, Answer Set Programming, Implementation

1 Introduction

In *answer set programming* (see e.g. [2] and the references therein), one uses a logic program to modularly describe the requirements that must be fulfilled by the solutions to a problem. The solutions then correspond to the models (answer sets) of the program, which are usually defined through (a variant of) the stable model semantics [13]. The technique has been successfully applied in problem areas such as planning [14, 6, 7], configuration and verification [20], diagnosis [5, 17, 24], game theory [25], updates [8] and database repairs [1, 15].

The *extended answer set* semantics for, possibly inconsistent, simple programs (containing only classical negation) is defined by allowing rules to be *defeated* (not satisfied). An *ordered logic program* then consists of a simple program with a partial order on the rules, representing a preference for satisfying certain rules, possibly at the cost of violating less important ones. Such a rule preference relation induces an order on extended answer sets, the minimal elements of which are called *preferred answer sets*. It can be shown [18] that the resulting semantics has a similar expressiveness as disjunctive logic programming, e.g. the membership problem is Σ_2^P-complete. Ordered programs have natural applications in e.g. database repair [15] or diagnosis [17, 24].

* Supported by the FWO.

** This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

This paper describes the design and implementation of the OLPS system that can be used to compute the preferred answer sets of ordered programs. It is organized as follows: after a brief overview of the preferred answer set semantics for ordered programs (Section 2), we present the OLPS system in Section 3. Section 4 discusses an algorithm, based on *partial interpretations*, to compute the extended answer sets of a simple (unordered) program. In Section 5, this algorithm is adapted to take into account the rule order, and compute only preferred answer sets. Finally, Section 6 contains the results of some preliminary experiments and directions for further research.

The OLPS system has been released under the GPL and is available for download from `http://tinf2.vub.ac.be/olp`.

2 Preferred Answer Sets for Ordered Programs

Preliminaries and Notation. A *literal* is an *atom* a or a negated atom $\neg a$. For a literal l we use $\neg l$ to denote its inverse, i.e. $\neg l = \neg a$ iff $l = a$ while $\neg l = a$ iff $l = \neg a$. For a set of literals X, we use $\neg X$ to denote $\{\neg l \mid l \in X\}$. Such a set is *consistent* iff $X \cap \neg X = \emptyset$. In addition, we also consider the special symbol $\perp$ denoting contradiction. Any set $X \cup \{\perp\}$, with X a set of literals, is inconsistent. For a set of atoms A, we use $\mathcal{L}_A$ to denote the set of literals over A and define $\mathcal{L}_A^{\perp} = \mathcal{L}_A \cup \{\perp\}$.

A *rule* r is of the form $h_r \leftarrow b_r$ where b_r, the *body* of the rule, is a set of literals and h_r, the rule's *head*, is a literal or $\perp$. In the latter case, the rule is called a *constraint*[1], in the former case, it is called a h_r-rule.

For a set of rules R we use $R^\star$ to denote the unique smallest Herbrand model, see [22], of the positive logic program obtained from P by considering all literals and $\perp$ as separate atoms.

Simple Logic Programs and Extended Answer Sets. A *simple logic program* (SLP) is a countable set of rules. For a SLP P, we use $\mathcal{B}_P$ to denote its *Herbrand base*, i.e. the set of atoms appearing in the rules of P. An *interpretation* for P is any consistent subset of $\mathcal{L}_{\mathcal{B}_P}$. For an interpretation I and a set of literals X we write $I \models X$ just when $X \subseteq I$.

A rule $r = h_r \leftarrow b_r$ is *satisfied* by I, denoted $I \models r$, iff $h_r \in I$ whenever $I \models b_r$, i.e. whenever r is *applicable* ($I \models b_r$), it must be *applied* ($I \models b_r \cup \{h_r\}$); r is *defeated* by I, denoted $I \models \neg r$ iff there is an applied *competing* rule $r' = \neg h_r \leftarrow b_{r'}$. Note that, consequently, constraint rules cannot be defeated.

The semantics defined below deals with possibly inconsistent programs in a simple, yet intuitive, way: when faced with contradictory applicable rules for l and $\neg l$, one selects one, e.g. the l-rules, for application and ignores (defeats) the contradicting $\neg l$-rules.

Let I be an interpretation for a SLP P. The *reduct* of P w.r.t. I, denoted P_I is the set of rules satisfied by I, i.e. $P_I = \{r \in P \mid I \models r\}$. An interpretation I is called an *extended answer set* of P iff I is *founded*, i.e. $P_I^\star = I$, and each rule r in P is either satisfied or defeated, i.e. $\forall r \in P \cdot I \models r \vee I \models \neg r$.

[1] To simplify the theoretical treatment we use an explicit contradiction symbol $\perp$ in the head of constraint rules. The concrete OLPS syntax employs the usual notation where the head of a constraint is empty.

Example 1. The program P_1 shown below has 2 extended answer sets $\{\neg a, b\}$ and $\{a, \neg b\}$ corresponding to the reducts $\{c, r_{\neg a}, r_a, r_b\}$ and $\{c, r_{\neg b}, r_a, r_b\}$, respectively.

$$\begin{array}{rlrlrl} r_{\neg a}: & \neg a \leftarrow & r_a: & a \leftarrow \neg b & r_{\neg b}: & \neg b \leftarrow \\ r_b: & b \leftarrow \neg a & c: & \bot \leftarrow \neg a, \neg b & & \end{array}$$

Ordered Programs and Preferred Answer Sets. An *ordered logic program* (OLP) is a pair $\langle R, < \rangle$ where R is a simple program and $<$ is a well-founded strict[2] partial order on the rules in R[3].

Intuitively, $r_1 < r_2$ indicates thats r_1 is preferred over r_2. The notation is extended to sets of rules, e.g. $R_1 < R_2$ abbreviates $\bigwedge_{r_1 \in R_1 \wedge r_2 \in R_2} r_1 < r_2$.

The preference $<$ on rules in $\langle R, < \rangle$ will be translated to a preference relation on the extended answer sets of R via an ordering on reducts: a reduct R_1 is preferred over a reduct R_2, denoted $R_1 \sqsubseteq R_2$ iff $\forall r_2 \in R_2 \backslash R_1 \cdot \exists r_1 \in R_1 \backslash R_2 \cdot r_1 < r_2$, i.e. each rule from $R_2 \backslash R_1$ is "countered" by a rule in $R_1 \backslash R_2$. It can be shown (Theorem 6 in [15]) that $\sqsubseteq$ is a partial order on 2^R. Consequently, we write $R_1 \sqsubset R_2$ just when $R_1 \sqsubseteq R_2$ but $R_1 \neq R_2$. The $\sqsubseteq$-order on reducts induces a preference order on the extended answer sets of R: for extended answer sets M_1 and M_2, $M_1 \sqsubseteq M_2$ iff $R_{M_1} \sqsubseteq R_{M_2}$. Minimal (according to $\sqsubset$) extended answer sets of R are called *preferred answer sets* of $\langle R, < \rangle$. An extended answer set is called *proper* iff it satisfies all minimal elements from R.

Example 2. Consider the ordered program below, which is written using the OLPS-syntax: $\neg$ is written as "`-`" and rules are grouped in modules that are partially ordered using statements of the form "`A < B`".

```
Avoid { pass :− study .   study .  }
Prefer { −study . }
ForSure { −pass :− −study .   pass :− −pass .  }
ForSure < Prefer < Avoid
```

The program expresses the dilemma of a person preferring not to study but aware of the fact that not studying leads to not passing (`-pass :- -study`) which is unacceptable (`pass :- -pass`). It is straightforward to verify that the single (proper) preferred answer set is $\{study, pass\}$ which satisfies all rules in `ForSure` and `Avoid`, but not the rules in `Prefer`.

In [15, 16] it is shown that OLP can simulate negation as failure (i.e. adding negation as failure does not increase the expressiveness of the formalism) as well as disjunction (under the minimal possible model semantics) and that e.g. membership is Σ_2^P-complete. This makes OLP as expressive as disjunctive logic programming under its normal semantics. However, as with logic programming with ordered disjunction[4], no effective translation is known in either direction.

2 A strict partial order $<$ on a set X is a binary relation on X that is antisymmetric, anti-reflexive and transitive. The relation $<$ is well-founded if every nonempty subset of X has a $<$-minimal element.

3 Strictly speaking, we should allow R to be a multiset or, equivalently, have labeled rules, so that the same rule can appear in several positions in the order. For the sake of simplicity of notation, we will ignore this issue: all results also hold for the general multiset case.

3 The Ordered Logic Program Solver (OLPS) System

OLPS computes (a selection of) the proper preferred answer sets of a finite ordered program which is described using a sequence of module definitions and order assertions. A module is specified using a module name followed by a set of rules, enclosed in braces while an order assertion is of the form $m_0 < m_1 < \ldots < m_n$, $n > 0$, where each m_i, $0 \leq i \leq n$ is a module name.

Rules are written as usual in datalog, with a few exceptions: variables must start with an uppercase letter and classical negation ($\neg$) is represented by a "-" in front of a literal, e.g. `-p(X,a)`. In addition, some convenient syntactic sugar constructs can be used in non-grounded programs. E.g. rules such as `t({1,2-4,a}).` abbreviate `t(1).t(2).t(3).t(4).t(a).` and variables can be "typed", where a type is a unary predicate: e.g. `p(X:t) :- q(Y:r,Z).` abbreviates `p(X) :- q(Y,Z), t(X), r(Y).`

The example program in Figure 1 describes the operation of a unary adder, as shown in Figure 2 [12]. It illustrates how ordered programs can be used to implement diagnostic systems [17].

```
Error { fault (N:gate , F: fault ). } % May be needed to explain observation.
Default { − fault (N:gate , F: fault ).  % By default, gates are not faulty.
  −adder(X:bit , Y:bit , Z:bit , Sum:bit , Carry: bit ). % Naf for adder/5.
  }
Model { bit ({0, 1}).   gate({xor1 , xor2 , and1 , and2 , or1 }).
   fault ({ stuck_at_0 ,  stuck_at_1 }).
  xor(N:gate ,0,0,1) :−  fault (N, stuck_at_1 ).  xor(N:gate ,1,1,1) :−  fault (N, stuck_at_1 ).
  xor(N:gate ,0,1,0) :−  fault (N, stuck_at_0 ).  xor(N:gate ,1,0,0) :−  fault (N, stuck_at_0 ).
  and(N:gate ,1,1,0) :−  fault (N, stuck_at_0 ).  and(N:gate ,1,0,1) :−  fault (N, stuck_at_1 ).
  and(N:gate ,0,1,1) :−  fault (N, stuck_at_1 ).  and(N:gate ,0,0,1) :−  fault (N, stuck_at_1 ).
  or(N:gate ,1,1,0) :−  fault (N, stuck_at_0 ).  or(N:gate ,1,0,0) :−  fault (N, stuck_at_0 ).
  or(N:gate ,0,1,0) :−  fault (N, stuck_at_0 ).  or(N:gate ,0,0,1) :−  fault (N, stuck_at_1 ).
  % Normal model
  adder(X:bit , Y:bit , Z:bit , Sum:bit , Carry: bit ) :−
    xor(xor1 , X,Y,S ), xor(xor2 , Z,S,Sum), and(and1 , X,Y,C1), and(and2 , Z,S,C2),
    or(or1 , C1,C2,Carry).
  % Normal behaviour of gates.
  xor(N:gate , 1,1,0).   xor(N:gate , 0,1,1).   xor(N:gate , 1,0,1).   xor(N:gate , 0,0,0).
  and(N:gate , 1,1,1).   and(N:gate , 1,0,0).   and(N:gate , 0,1,0).   and(N:gate , 0,0,0).
  or(N:gate , 1,1,1).    or(N:gate , 1,0,1).    or(N:gate , 0,1,1).    or(N:gate , 0,0,0).
}
Observations { :− −adder (0,0,1,0,1).  }
Model < Default < Error
```

Fig. 1. A program for circuit diagnosis.

Intuitively, observations are represented using constraints, and rules describing the normal operation of the system are preferred over "fault rules" that specify possible ab-

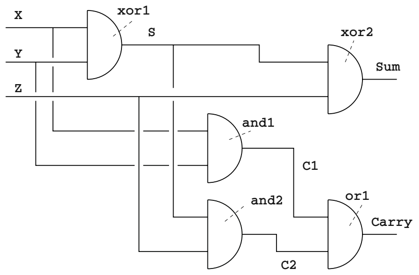


Fig. 2. Unary adder [12] described in the program of Figure 1.

normal behaviors. Here, the `adder`-rule in `Model` describes the normal operation of the circuit where variables correspond to the connections between the gates, which are named in the `gate/1`-predicate. It is assumed that a broken gate may have a fixed output, whatever its inputs. This leads to the introduction of two constants `stuck_at_0` and `stuck_at_1` (defined in the `fault/1` rules) and a specification of the behavior of the various gate types when they are stuck using rules such as `xor(N:gate, 0, 0, 1) :- fault(N, stuck_at_1).` The `Default` module specifies that `fault/2` and `adder/5` are false by default (`Model` < `Default`).

To add diagnostic capabilities, it suffices to add another weaker module `Error` that contains rules that should only be used "as a last resort".

The observation of a malfunctioning circuit is described using a constraint, e.g. `:- -adder(0,0,1,0,1)` forces OLPS to find an explanation for `adder(0,0,1, 0,1)`. To this end, some rules in `Default` will need to be defeated by applying some weaker rules from `Error`. As shown in [17], each preferred answer set will contain a (subset) minimal set of `fault/2` literals.

Running OLPS on the example using the command[4]

```
olps -p 'fault/2' -n 0 circuit.olp
```

will compute the possible minimal explanations shown below.

```
{ + fault (xor1 ,  stuck_at_1 ) }
{ + fault (or1 ,  stuck_at_1 ) + fault (xor2 ,  stuck_at_0 ) }
{ + fault (and2 ,  stuck_at_1 ) + fault (xor2 ,  stuck_at_0 ) }
{ + fault (and1 ,  stuck_at_1 ) + fault (xor2 ,  stuck_at_0 ) }
```

Like SMODELS[21], OLPS first produces a grounded version of the program that then serves as input to the solver proper. The default grounding[5], *olpg*, produces all (some are, however, optimized away) the instances of rules that are used in the computation of the minimal answer set of the positive program, obtained by considering all literals as separate atoms.

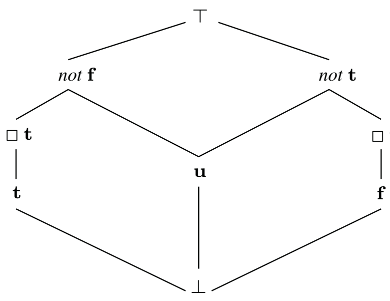
[4] The "`-p`" option is used to print only the `fault/2` predicate, "`-n 0`" will cause the system to compute all proper preferred answer sets.

[5] The grounding program runs as a separate process and can be selected at run time.

4 Computing Extended Answer Sets for Simple Programs

Partial Interpretations

OLPS searches for answer sets by building and extending *partial interpretations* that carry intermediate information regarding the status of literals and rules. To represent such information on literals, we use the lattice $\mathbf{T}_9$ of truth values depicted in Figure 3. Intuitively, $\mathbf{T}_9$ can be considered as an extension of FOUR from [3, 19] with approximations[6] $\Box\,\mathbf{t}$ and $\Box\,\mathbf{f}$ of resp. $\mathbf{t}$ and $\mathbf{f}$, denoting that a literal must eventually become resp. true or false at the end of the computation in order for a partial interpretation to result in an extended answer set. Further, we use *not* $\mathbf{t}$ and *not* $\mathbf{f}$ as explicit representations of the complements of $\mathbf{t}$ and $\mathbf{f}$. Clearly, the order $\sqsubset$ in $\mathbf{T}_9$ corresponds to the "knowledge" ordering[3, 19], i.e. $t_1 \sqsubset t_2$ indicates that t_1 is more determined than t_2.



	Intuition	t	$\neg t$
$\bot$	Contradiction.	$\bot$	$\bot$
$\top$	No information.	$\top$	$\top$
$\mathbf{t}$	True.	$\mathbf{t}$	$\mathbf{f}$
$\mathbf{f}$	False.	$\mathbf{f}$	$\mathbf{t}$
$\Box\,\mathbf{t}$	Eventually true.	$\Box\,\mathbf{t}$	$\Box\,\mathbf{f}$
$\Box\,\mathbf{f}$	Eventually false.	$\Box\,\mathbf{f}$	$\Box\,\mathbf{t}$
not $\mathbf{t}$	Not true.	*not* $\mathbf{t}$	*not* $\mathbf{f}$
not $\mathbf{f}$	Not false.	*not* $\mathbf{f}$	*not* $\mathbf{t}$
$\mathbf{u}$	Neither true nor false.	$\mathbf{u}$	$\mathbf{u}$

Fig. 3. Truth value lattice $\mathbf{T}_9$.

The general idea behind the usage of $\mathbf{T}_9$ is to start with $\top$ ("no information") for each literal and evolve during the computation towards either $\mathbf{t}$, $\mathbf{f}$, $\mathbf{u}$ or $\bot$, taking the knowledge ordering $\sqsubset$ into account. When, at the end of the computation, a partial interpretation assigns either $\mathbf{t}$, $\mathbf{f}$or $\mathbf{u}$ to each literal, we have found an extended answer set.

Definition 1. *A* $\mathbf{T}_9$***-valuation*** *on a set of atoms A is a total function ϕ assigning a truth value $\phi(a)$ to each $a \in A$; it is extended to literals over A by defining $\phi(\neg a) = \neg\phi(a)$, for $a \in A$. A valuation ϕ is* ***consistent*** *iff $\phi^{-1}(\bot) = \emptyset$. It is* ***final*** *iff it assigns only to truth values that cannot be improved without introducing contradiction, i.e. $\forall t \notin \{\mathbf{t}, \mathbf{f}, \mathbf{u}\} \cdot \phi^{-1}(t) = \emptyset$.*

The order in $\mathbf{T}_9$ induces a partial ordering on valuations: ϕ_1 *extends* ϕ_2, denoted $\phi_1 \sqsubseteq \phi_2$, iff $\phi_1(a) \sqsubseteq \phi_2(a)$ for all $a \in A$. Intuitively, $\phi_1 \sqsubset \phi_2$ (i.e. $\phi_1 \sqsubseteq \phi_2$ and $\phi_1 \neq \phi_2$) if ϕ_1 is more determined than ϕ_2.

$\mathbf{T}_9$-valuations will be represented as sets of extended literals where an *extended literal* is a literal or of one of the forms $\Box\, l$ or *not* l, with l an ordinary literal. For an

[6] The notation $\Box\, t$ should not be confused with the "always" modality from modal logic.

extended literal e, we use $\widehat{e}$ to denote the underlying atom, i.e. $\widehat{\neg a} = \widehat{a} = a$, while $\widehat{\Box\, l} = \widehat{\textit{not}\, l} = \widehat{l}$. For a set of extended literals E, $\widehat{E}$ abbreviates $\{\widehat{e} \mid e \in E\}$. The set of all extended literals over a set of atoms A is denoted $\mathcal{E}_A$ while $\mathcal{E}_A^{\perp} = \mathcal{E}_A \cup \{\perp\}$. For a set of literals X, $\Box\, X$ abbreviates $\{\Box\, x \mid x \in X\}$.

We associate a truth value $v(e)$ from $\mathbf{T}_9$ with an extended literal e, where $\widehat{e} = a$, in the obvious way: $v(a) = \mathbf{t}$, $v(\textit{not}\ a) = \textit{not}\,\mathbf{t}$ and $v(\Box\, a) = \Box\,\mathbf{t}$ while $v(\neg a) = \neg(v(a)) = \mathbf{f}$, $v(\textit{not}\,\neg a) = \neg v(\textit{not}\, a) = \textit{not}\,\mathbf{f}$ and $v(\Box\,\neg a) = \neg v(\Box\, a) = \Box\,\mathbf{f}$. Using v, we can interpret a set E of extended literals as a valuation ϕ_E by defining

$$\phi_E(a) = \sqcap\{v(e) \mid e \in E \ \wedge\ \widehat{e} = a\}$$

where, by definition, $\sqcap\emptyset = \top$. E.g., if $E = \{\Box\, a, a, \textit{not}\, b, \textit{not}\,\neg b, \Box\, c, \textit{not}\, c\}$ is a set of extended literals over $\{a, b, c, d\}$ then $\phi_E(a) = \mathbf{t}$, $\phi_E(b) = \mathbf{u}$, $\phi_E(c) = \perp$ and $\phi_E(d) = \top$. A set of extended literals E is consistent and/or final iff ϕ_E is consistent and/or final. Obviously, if $E_1 \subseteq E_2$, then $\phi_{E_2} \sqsubseteq \phi_{E_1}$.

A set of extended literals E_1 *extends* a set E_2, denoted $E_1 \sqsubseteq E_2$ iff $\phi_{E_1} \sqsubseteq \phi_{E_2}$. A *conservative extension* of a set of extended literals E is any superset $E' \supseteq E$ that preserves the associated valuation, i.e. $\phi_{E'} = \phi_E$. Since the set of conservative extensions of a set of extended literals is closed under union, we can define the *closure* $\overline{E}$ of a set of extended literals E as the unique maximal conservative extension of E. E.g., the closure of $E = \{\Box\, a, a, \textit{not}\, b, \textit{not}\,\neg b, \Box\, c, \textit{not}\, c\}$ is $\overline{E} = \{\Box\, a, a, \textit{not}\,\neg a, \textit{not}\, b, \textit{not}\,\neg b, \Box\, c, \Box\,\neg c, c, \neg c, \textit{not}\, c, \textit{not}\,\neg c\}$. It can be shown that, for sets of extended literals E_1 and E_2, $E_1 \sqsubseteq E_2$ iff $\overline{E_2} \subseteq \overline{E_1}$.

For a set of extended literals E we write that $E \models F$, with F a set of extended literals, iff $F \subseteq \overline{E}$.

In the sequel, we will often abuse notation by considering a set of rules R also as a set of atoms (disjoint from $\mathcal{B}_R$), one for each rule $r \in R$, thus defining e.g. $\mathcal{L}_R$.

Definition 2. *A* ***partial interpretation*** *of a simple program R is a set $I \subseteq \mathcal{L}_R \cup \mathcal{E}_{\mathcal{B}_R}^{\perp}$. Intuitively, the rule literals $I_R = I \cap \mathcal{L}_R$ represent the desired status of the rules from R: if $r \in I_R$ then r should be satisfied while $\neg r \in I_R$ indicates that r should be defeated. $I_L = I \cap \mathcal{E}_{\mathcal{B}_R}$ represents a valuation of $\mathcal{B}_R$. The* ***reduct*** *of R w.r.t. I, denoted R_I is defined by $R_I = \{r \mid r \in I_R\}$. A partial interpretation I is*

- ***complete*** *iff $\widehat{I_R} = R$, i.e. each rule has a desired status;*
- ***consistent*** *iff $\perp \notin I$, both I_R and ϕ_{I_L} are consistent and, moreover, there exists a final consistent extension $F \sqsubseteq I_L$ such that $\forall l \in I_R \cdot F \cap \mathcal{L}_{\mathcal{B}_R} \models l$, i.e. I_R is consistent with I_L;*
- ***final*** *iff ϕ_{I_L} is final; and*
- ***founded*** *iff $(R_I)^{\star} = I_L \cap \mathcal{L}_{\mathcal{B}_R}^{\perp}$.*

A partial interpretation J ***extends*** *a partial interpretation I, denoted $I \sqsubseteq J$ iff $I_R \cup \overline{I_L} \subseteq J_R \cup \overline{J_L}$.*

Note that a partial interpretation need not be consistent. It is easily seen that $\sqsubseteq$ defines a partial order on partial interpretations and that all extensions of an inconsistent partial interpretation are themselves inconsistent.

Extended answer sets correspond to partial interpretations that are complete, consistent, final and founded.

Proposition 1. *Let R be a simple program. If M is an extended answer set of R then*

$$\Pi_R(M) = R_M \cup \neg(R \backslash R_M) \cup M \cup \bigcup_{a \in \mathcal{B}_R \backslash \widehat{M}} \{\mathit{not}\, a, \mathit{not}\, \neg a\}$$

is a partial interpretation that is complete, consistent, final and founded. Conversely, $I \cap \mathcal{L}_{\mathcal{B}_R}$ is an extended answer set for any partial interpretation I that is complete, consistent, final and founded.

Note that the last component of $\Pi_R(M)$ corresponds to a version of the closed world assumption: any literal l for which no information is available is assumed to be "necessarily unknown", i.e. $\phi_{\Pi_R(M)}(l) = \mathbf{u}$.

A rule r is *blocked* w.r.t. a set of extended literals E iff $\exists l \in b_r \cdot E \models \mathit{not}\, l$. If r is not blocked w.r.t. E, it is said to be *open*. We use $R_h(E)$ to denote the sets of h-rules from R that are open w.r.t. E. An open rule r is *applicable* w.r.t. E iff $E \models \Box\, b_r$, it is *applied* iff it is applicable, $h_r \neq \bot$, and, moreover, $E \models \Box\, h_r$.

For a given partial interpretation I, we need an operator $\Phi^\star_R(I)$ to compute the maximal deterministic extension of I. This operator is based on the primitive notion of forcing, that, for a partial interpretation I and a single rule r, defines which new information can be deterministically derived from I and r.

Definition 3. *A partial interpretation I for an SLP R* **forces** *a set of (extended or rule) literals J, denoted $I \Vdash J$ iff $X \Vdash J$ (and I fulfills the extra condition, if any) for some $X \subseteq I_R \cup \overline{I_L}$ where $\Vdash$ is defined below.*

$$\Vdash \{r\} \qquad \text{if } h_r = \bot \tag{1}$$
$$\{\neg r\} \Vdash \Box\, b_r \cup \{\Box\, \neg h_r\} \qquad \text{if } h_r \neq \bot \tag{2}$$
$$\{\mathit{not}\, \neg h_r\} \Vdash \{r\} \tag{3}$$
$$\{r\} \cup \Box\, b_r \Vdash \{\Box\, h_r\} \qquad \text{if } h_r \neq \bot \tag{4}$$
$$\{r\} \cup \Box\, b_r \Vdash \{\bot\} \qquad \text{if } h_r = \bot \tag{5}$$
$$\{r\} \cup b_r \Vdash \{h_r\} \tag{6}$$
$$\Box\, b_r \cup \{\mathit{not}\, h_r\} \Vdash \{\neg r\} \tag{7}$$
$$\{\Box\, h_r\} \Vdash \{r\} \cup \Box\, b_r \qquad \text{if } R_{h_r}(I_L = I \cap \mathcal{E}_{\mathcal{B}_R}) = \{r\} \tag{8}$$
$$\Box\, (b_r \backslash \{b\}) \cup \{r, \mathit{not}\, h_r\} \Vdash \{\mathit{not}\, b\} \qquad \text{if } b \in b_r \tag{9}$$
$$\Box\, (b_r \backslash \{b\}) \cup \{r\} \Vdash \{\mathit{not}\, b\} \qquad \text{if } b \in b_r \text{ and } h_r = \bot \tag{10}$$
$$\Vdash \{\mathit{not}\, h_r\} \qquad \text{if } h_r \neq \bot \text{ and } R_{h_r}(I_L) = \emptyset \tag{11}$$
$$\{\mathit{not}\, b\} \Vdash \{r\} \qquad \text{if } b \in b_r \tag{12}$$

Intuitively, (1) asserts that constraints cannot be defeated while (2) encodes the definition of defeat: $\neg r$, i.e. r is defeated, iff r is applicable but $\neg h_r$ is implied by some defeating rule. Consequently, if $\neg h_r$ cannot be true, the rule r must be satisfied (3). Definitions (4,5) and (6) encode (satisfied) rule application while (7) expresses that an applicable rule that cannot be applied must be defeated. Definition (8) indicates that if only a single rule is available to motivate a needed literal, it must eventually become applied. On the other hand, an almost applicable satisfied rule with a conclusion that is inconsistent with the interpretation must be blocked (9,10). If there are no open rules for a literal a, then *not* a must hold (11). Finally, a blocked rule must be satisfied (12).

Definition 4. *Let R be a finite simple program. The operator Φ_R is defined by $\Phi_R(I) = I \cup \bigcup_{I \Vdash X} X$, for any partial interpretation I. The closure $\Phi^\star_R$ of Φ_R is defined by $\Phi^\star_R(I) = \bigcup_{n>0} \Phi^n_R(I)$.*

It can be shown that $\Phi^\star_R(I)$ is unique and extends I, i.e. $I \sqsubseteq \Phi^\star_R(I)$.

Clearly, $\Phi^\star_R(I)$ computes the maximal deterministic extension of a partial interpretation I. It encompasses the Fitting operator[11] and plays a similar role as does the function *det_cons* in DLV[9], or *expand* in SMODELS[21].

Example 3. Reconsider program P_1 from Example 1 and the interpretation $I = \{r_{\neg a}\}$. The table below illustrates a possible computation of $\Phi^\star_{P_1}(I)$.

$\{r_{\neg a}\} \Vdash \{\neg a\}$	(6)	$\{\neg r_{\neg b}\} \Vdash \{\Box\, b\}$	(2)
$\Vdash \{c\}$	(1)	$\{\Box\, b\} \Vdash \{r_b, \Box\, \neg a\}$	(8)
$\{c, \neg a\} \Vdash \{\mathit{not}\, \neg b\}$	(10)	$\{\mathit{not}\, \neg b\} \Vdash \{r_a\}$	(12)
$\{\mathit{not}\, \neg b\} \Vdash \{\neg r_{\neg b}\}$	(7) on $r_{\neg b}$	$\{r_b, \neg a\} \Vdash \{b\}$	(6) on r_b

Thus, $\Phi^\star_{P_1}(I) = \{r_{\neg a}, \neg r_{\neg b}, r_a, r_b, c, \neg a, b\} = \Pi_{P_1}(\{\neg a, b\})$.

Consistency is easy to check for fixpoints of Φ_R.

Proposition 2. *Let I be a partial interpretation of a simple program R such that $\Phi_R(I) = I$. Then I is consistent iff $\bot \notin I$ and both I_R and I_L are consistent.*

The following is an easy consequence of (6) in Definition 3.

Proposition 3. *Let I be a partial interpretation of a simple program R. If I is founded then so is $\Phi^\star_R(I)$.*

Complete founded fixpoints of Φ_R have no consistent founded proper extensions.

Proposition 4. *Let I be a consistent complete founded partial interpretation of a simple program R such that $\Phi_R(I) = I$. Then $J = I$ for all $I \sqsubseteq J$ such that J is consistent and founded.*

Replacing an interpretation I by $\Phi^\star_R(I)$ does not loose any answer sets.

Proposition 5. *Let I be an interpretation of a simple program R. Any extended answer set M of R that extends I, i.e. $I \sqsubseteq \Pi_R(M)$, also extends $\Phi^\star_R(I)$, i.e. $\Phi^\star_R(I) \sqsubseteq \Pi_R(M)$.*

The following example shows that consistent maximal (and thus complete) founded extensions are not necessarily final.

Example 4. Consider the simple program P_2 shown below and the empty partial interpretation.

$$\begin{array}{lll} r_0 : \neg a \leftarrow & r_1 : \ b \leftarrow a & r_2 : c \leftarrow b \\ r_3 : \ \ a \leftarrow c & r_4 : \bot \leftarrow \neg a & \end{array}$$

It is straightforward to verify that $\Phi^\star_{P_2}(\emptyset) = \{r_4, \neg r_0, r_1, r_2, r_3, \mathit{not}\, \neg a, \Box\, a, \Box\, b, \Box\, c\}$ does not correspond to an extended answer set (in fact, P_2 does not have any extended answer sets). Intuitively, r_0 cannot be defeated because the only possible motivation for a is based on a circularity.

A set such as $\{\Box\, a, \Box\, b, \Box\, c\}$ in Example 4 is called *unfounded*[7]. Formally, a set X, $X \subseteq \Box\, \mathcal{L}_{\mathcal{B}_P}$, is unfounded w.r.t. a partial interpretation I iff, for any $\Box\, l \in X$, each non-blocked (w.r.t. I) l-rule r contains a literal $d \in b_r$ such that $\Box\, d \in X$. It can be shown that if I contains an unfounded set, then there are no extended answer sets among its extensions. This result is used in the *prune* function from Figure 4.

The *aset* Procedure

The main procedure for enumerating extended answer sets that are extensions of a given partial interpretation is shown in Figure 4. Note that the $select$ function returns an arbitrary rule from its argument set.

```
 PartialInterpretation
prune(const Program& R, PartialInterpretation $I$) {
J = $\Phi^{\star}_{R}(I)$;
if (J contains an unfounded set)
  J = J $\cup \{\bot\}$;
return J;
}

 set< Interpretation >
 aset (const Program& R, PartialInterpretation $I$) {
// Precondition: I is founded and $\Phi_R(I) = I$.
if (! $I$ is consistent ) // Easy to check because of Proposition 2.
  return $\emptyset$; // There are no answer sets extending $I$.
if ( $I$ is complete) {
  if ( $I$ is final ) // $I$ corresponds to an answer set by Proposition 1.
    return $\{I \cap (\mathcal{B}_R \cup \neg\mathcal{B}_R)\}$;
  else return $\emptyset$; // By Proposition 4, there are no answer sets extending $I$.
  }
else {
  Rule r = select ($R \setminus \widehat{I_R}$);
  // The preconditions for the calls are assured by Proposition 3..
  return aset(R, prune($I \cup \{r\}$) $\cup$ aset(R, prune($I \cup \{\neg r\}$);
  }
}
```

Fig. 4. The *aset* function for simple programs.

Proposition 6. *Let R be a simple program and let I be a founded fixpoint of Φ_R. Then $aset(R, I)$ will return all extended answer sets of R that extend I.*

From Proposition 6, it follows that all extended answer sets of R can be obtained using the call $aset(R, \Phi^{\star}_{R}(\emptyset))$.

[7] We use the term "unfounded" in this context since the intuition behind it is similar to unfounded sets in the well-founded semantics[23].

The implementation of $\Phi^\star_R$ uses a queue of pattern occurrences, each pattern corresponding to the left hand side of one of the rules in Definition 3. The queue is processed by adding the right hand side of the pattern to the partial interpretation, thus possibly generating further patterns for the queue. The computation finishes when an inconsistency is detected or the queue becomes empty. This design is sound because a pattern remains applicable in any consistent extension of the partial interpretation where it was first detected. Detection is facilitated by keeping some derived information such as the number of "open" literals in rule bodies, the number of open rules for a given literal etc.

5 Computing Preferred Answer Sets

A naive way to compute preferred answer sets would be to compute all extended answer sets and then retrieve the minimal (according to $\sqsubset$) elements.

OLPS tries instead to detect (and prune) partial interpretations that cannot lead to preferred answer sets as soon as possible. This is done by (a) always extending a partial interpretation I using a minimal rule (among the "open" rules), and (b) checking, for each previously found preferred answer set M, whether it is still possible to find a set of rules $N \subseteq R$ such that $\{r \in R \mid r \in I_R\} \subseteq N$ and $R_M \not\subseteq N$.

In this context, a module[8] $X \subseteq R$ is said to be *decided* by a partial interpretation I when, by abuse of notation, $X \subseteq \widehat{I_R}$, i.e. each rule $r \in X$ has a status in I. Further, two partial interpretations I and J are *equal* w.r.t. a module X iff $I_X = J_X$, i.e. they have the same status for the rules in X.

For a complete partial interpretation I and an arbitrary partial interpretation J, we say that I is *incomparable* w.r.t. J iff there exist a module $X \subseteq R$ such that

- $(J_X \cap X)\backslash(I_X \cap X) \neq \emptyset$, i.e. J has at least one satisfied rule in X that is defeated by I; and
- every module $Y \subseteq R$ with $Y < X$ is decided by J and, moreover, I and J are equal w.r.t. Y.

On the other hand, I is *stronger* than J iff for each module X which is such that I and J are equal w.r.t. all more preferred modules $Y < X$[9], it holds that

- $(X \cap J_X) \subsetneq (X \cap I_X)$ i.e. I satisfies strictly more rules in X than does J; and
- $(X \backslash \hat{J}) \subseteq I$, i.e. all rules in X that have not yet a status in J are satisfied w.r.t. I.

The following are easy consequences of the above definitions.

Proposition 7. *Let I be a complete partial interpretation and let J be a partial interpretation. I incomparable w.r.t. J implies that $R_{I_L} \not\subseteq R_{K_L}$ for every extension K of J.*

Proposition 8. *Let I be a complete partial interpretation and let J be a partial interpretation. I stronger than J implies that $R_{I_L} \sqsubset R_{K_L}$ for every extension K of J.*

[8] We use the term *module*, just as in the syntax of OLPS, to denote a maximal set of rules $X \subseteq R$ that are all at the same position in the well-founded strict partial order on R. Clearly, this order on rules induces an equivalent order on the modules.

[9] This implies that Y is decided by J.

Clearly, checking incomparability or being stronger can be performed, even in the absence of optimization, in linear time and space (w.r.t. the size of the program).

Importing these checks into an adapted version of the prune function, as shown in Figure 5 ensures an early detection of a situation where no extended answer sets that extend I can be minimal.

```
⟨PartialInterpretation , $set < CompletePartialInterpretation >$⟩) {
 preferred_prune (const Program& R,
                 ⟨PartialInterpretation $I$, $set < CompletePartialInterpretation > P$⟩) {
J = $\Phi^{\star}_{R}(I)$;
if (J contains an unfounded set ) {
  J = J $\cup \{\bot\}$;
  return $\langle J, P\rangle$;
}
for each $T \in P$ {
  if $T$ incomparable w.r.t . $J$
        P = $P \setminus \{T\}$; // Due to Proposition 7.
  else if $T$ stronger than $J$ {
        J = J $\cup \{\bot\}$; // Due to Proposition 8.
        return $\langle J, P\rangle$;
  }
}
return $\langle J, P\rangle$;
}
```

Fig. 5. The *preferred_prune* function for ordered programs.

The procedure for finding preferred answer sets is shown in Figure 6. It can be shown that, if I is founded and $\Phi_R(I) = I$, then *preferred_aset(R,*$\langle I, P\rangle$*)* will return the set of all minimal (according to $\sqsubset$) extended answer sets M of R that extend I and such that no $T \in P$ exists for which $T \sqsubset M$ holds. It follows that *preferred_aset(R,*$\langle \Phi^{\star}_{R}(\emptyset), \emptyset\rangle$*)* computes all preferred answer sets of $\langle R, <\rangle$[10].

6 Conclusions and Directions for Further Research

Some preliminary tests of the current implementation have been conducted on a 2GHz Linux PC. The results are shown in Table 1: *circuit* refers to the program of Figure 1 while *ham-N* and *ham-dN* refer to programs that solve the Hamiltonian circuit problem on a randomly generated graph with N nodes and $N^2/10$, resp. $N^2/2$, edges. Note that the latter problem is Σ^P_1-complete and thus directly solvable by both SMODELS, DLV and OLPS. In [15] a transformation is presented of non-disjunctive seminegative programs into ordered programs where the preferred answer sets of the latter coincide with the classical subset minimal answer sets of the former. We have used this transformation to conduct our experiments with OLPS. It is clear from the table that the current naive

[10] Proper preferred answer sets are obtained by *preferred_aset(R,*$\langle \Phi^{\star}_{R}(R_{min}), \emptyset\rangle$*)*, with R_{min} contains the (atoms corresponding to the) $<$-minimal elements of R.

```
set< Interpretation >
 preferred_aset ( const Program& $R$,
                  $\langle$PartialInterpretation $I, set < CompletePartialInterpretation > P\rangle$) {
// Precondition: $I$ is founded and $\Phi_R(I) = I$.
if (! $I$ is consistent ) // Easy to check because of Proposition 2.
  return $\emptyset$; // There are no answer sets extending $I$.
if ( $I$ is complete) {
  if ( $I$ is final ) // $I$ corresponds to an answer set by Proposition 1.
    return $\{I \cap (\mathcal{B}_R \cup \neg\mathcal{B}_R)\}$;
  else return $\emptyset$; // By Proposition 4, there are no answer sets extending $I$.
  }
else {
  Rule r = select_min ($R \backslash \widehat{I_R}$);
  // Note that $n \not\sqsubseteq m$ for any $n \sqsupseteq (R \cup \{\neg r\})$, $m \sqsupseteq (R \cup \{r\})$.
  set< Interpretation > M = preferred_aset (R, preferred_prune ($(\langle I \cup \{r\}, P\rangle)$);
  return M $\cup$ preferred_aset(R, preferred_prune($(\langle I \cup \{\neg r\}, P \cup M\rangle)$);
  }
}
```

Fig. 6. The *preferred_aset* function for ordered programs.

Table 1. Preliminary performance tests.

input	olpg	OLPS	lparse	SMODELS	DLV
circuit	0m02.536s	0m00.154s	NA	NA	NA
ham-50	0m00.176s	0m00.060s	0m00.072s	0m00.084s	0m00.084s
ham-d50	0m04.465s	0m01.537s	0m00.118s	0m00.670s	0m06.613s
ham-60	0m00.245s	0m00.081s	0m00.086s	0m00.135s	34m14.371s
ham-d60	0m07.807s	0m03.553s	0m00.202s	0m02.103s	0m23.051s
ham-70	0m00.368s	0m00.124s	0m00.109s	0m00.216s	0m00.815s
ham-d70	0m12.882s	0m11.030s	0m00.265s	0m04.513s	0m57.121s
ham-80	0m00.533s	0m00.162s	0m00.145s	0m00.313s	0m00.276s
ham-d80	0m20.078s	0m35.501s	0m00.418s	0m11.050s	1m55.984s
ham-90	0m00.788s	0m00.248s	0m00.175s	0m00.468s	0m00.512s
ham-d90	0m29.781s	1m36.511s	0m00.429s	0m17.944s	3m57.191s
ham-100	0m01.326s	0m00.395s	0m00.228s	0m00.764s	0m01.164s
ham-d100	2m20.249s	54m04.504s	0m00.881s	2m30.887s	47m08.002s
ham-200	0m01.190s	0m00.459s	0m00.684s	0m02.937s	570m12.123s

grounder program *olpg* should be improved considerably: it performs much worse than *lparse*[11]. On the other hand, on the sparser graphs, *olps* performs similarly or slightly better than *smodels*, while on the dense graphs OLPS performs worse. The reason for the latter is subject to further research. For *dlv* only total (grounding and solving) figures are shown. Clearly, these tests are anecdotal and only a wider comparison on a range of applications can lead to firm conclusions. Nevertheless, we believe that the prelim-

[11] The fact that *olpg* outputs source code while *lparse* uses an efficient binary format does not help.

inary results are encouraging. One could argue that a similar approach can be used to compare OLPS with DLV on Σ_2^P-complete problems, however, at the moment there is no known transformation between ordered programs and disjunctive programs in either direction, as is also the case with e.g. logic programming with ordered disjunction[4].

Future versions should investigate the use of heuristics[10]. Currently, *select_min* (Figure 6), simply picks a minimal "open" rule that has a minimal number of undecided body literals. The use of more sophisticated heuristics by *select_min* and the detection and exploitation of certain special cases in other parts of the system could improve performance considerably. Finally, adding support for negation as failure (directly or through the construction used in [16]) would make it easy to add new front-ends for e.g. LPOD[4].

References

1. M. Arenas, L. Bertossi, and J.Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Procs. of the 4th International Conference on Flexible Query Answering Systems*, pages 27–41. Springer-Verlag, 2000.
2. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
3. N. D. Belnap. A useful four-valued logic. In *Modern uses of multi-valued logic*, pages 8–37. D. Reidel Publ. Co., 1975.
4. G. Brewka. Logic programming with ordered disjunction. In *Proc. of the National Conference on Artificial Intelligence*, pages 100–105. AAAI Press, 2002.
5. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.
6. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In *Procs. of the International Conference on Computational Logic (CL2000)*, volume 1861 of *LNCS*, pages 807–821. Springer, 2000.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. The DLV^k planning system. In *Logic in Artificial Intelligence*, volume 2424 of *LNAI*, pages 541–544. Springer Verlag, 2002.
8. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *Logic in Artificial Intelligence*, volume 1919 of *LNAI*, pages 2–20. Springer Verlag, 2000.
9. W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in DLP computations. In *Logic Programming and Non-Monotonic Reasoning*, volume 1730 of *LNAI*, pages 177–191. Springer Verslag, 1999.
10. W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 635–640. Morgan Kaufmann, 2001.
11. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of logic programming*, 4:295–312, 1985.
12. P. Flach. *Simply Logical - Intelligent Reasoning by Example*. Wiley, 1994.
13. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Procs. of the Intl. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
14. V. Lifschitz. Answer set programming and plan generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.
15. D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. In *Logic in Artificial Intelligence*, volume 2424 of *LNAI*, pages 432–443. Springer Verlag, 2002.

16. D. Van Nieuwenborgh and D. Vermeir. Order and negation as failure. In *Procs. of the Intl. Conference on Logic Programming*, volume 2916 of *LNCS*, pages 194–208. Springer Verlag, 2003.
17. D. Van Nieuwenborgh and D. Vermeir. Ordered diagnosis. In *Procs. of the Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *LNAI*, pages 244–258. Springer Verlag, 2003.
18. Davy Van Nieuwenborgh and Dirk Vermeir. Preferred answer sets for ordered logic programs. *Theory and Practice of Logic Programming (TPLP)*, page Accepted for publication, 2004.
19. T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, 13:445–463, 1990.
20. T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Procs. of the Intl. Workshop on Practical Aspects of Declarative Languages*, volume 1551 of *LNCS*, pages 305–319. Springer Verslag, 1999.
21. T. Syrjänen and I. Niemelä. The smodels system. In *Procs. of the Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, volume 2173 of *LNCS*, pages 434–438. Springer-Verlag, 2001.
22. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–742, 1976.
23. Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, 1991.
24. Davy Van Nieuwenborgh and Dirk Vermeir. Ordered programs as abductive systems. In *Proceedings of the APPIA-GULP-PRODE Conference on Declarative Programming (AGP2003)*, pages 374–385, Regio di Calabria, Italy, 2003.
25. M. De Vos and D. Vermeir. Choice Logic Programs and Nash Equilibria in Strategic Games. In *Computer Science Logic*, volume 1683 of *LNCS*, pages 266–276. Springer Verslag, 1999.

Improving Memory Usage in the BEAM

Ricardo Lopes[1] and Vítor Santos Costa[2]

[1] DCC-FC & LIACC, University of Porto
Rua do Campo Alegre, 823, 4150-180 Porto, Portugal
Tel. +351 226078830, Fax. +351 226003654
rslopes@dcc.fc.up.pt

[2] COPPE/Sistemas, Universidade Federal do Rio de Janeiro, Brasil
vitor@cos.ufrj.br

Abstract. A critical issue in the design of logic programming systems is their memory performance, both in terms of total memory usage and locality in memory accesses. BEAM, as most modern Prolog systems, requires both good emulator design and good memory performance for best performance. We report on a detailed study of the memory management techniques used on our sequential implementation of the EAM. We address questions like how effective are the techniques the BEAM uses to recover and reuse memory space, how garbage collection affects performance and how to classify and unify variables in a EAM environment. We also propose a finer variable allocation scheme to reduce memory overheads that is quite effective at reducing memory pressure, with only a small overhead.

Keywords: Logic Programming, Extended Andorra Model, Memory Management, Language Implementation.

1 Introduction

The BEAM or Basic Implementation of the Extended Andorra Model, is a first sequential implementation of the core rewriting rules from David H. D. Warren's Extended Andorra Model with Implicit Control [19]. We have designed the BEAM to be an extension of a traditional Prolog system, YAP [16], so that users can take advantage of the more flexible control strategy in the BEAM as a complement to a traditional Prolog environment.

Our first goal in designing the BEAM was to study the feasibility of Warren's design for the EAM. Initial results [10] do indeed show that our prototype is between 3 to 10 times slower than YAP on applications with the same search space (YAP is recognised as one of the fastest currently available Prolog systems [16]). This is reasonable considering the extra overheads of the EAM and the fact that implementation technology for Prolog is by now rather mature. On the other hand our results also show that considerable improvements can be achieved in the search-space. Compounded with the potential for parallelism, we believe our initial results agree with Warren's intuition for the EAM.

The BEAM requires an explicit representation of the search-tree and cannot therefore reuse space after backtracking, as traditional Prolog does. Memory management is thus a very fundamental issue in the BEAM and decisions on the subject permeate the whole system. In this study we discuss the memory management scheme used in the BEAM, addressing three main questions:

1. How to classify and unify variables in a EAM environment?
2. How effective are the techniques BEAM uses to recover and reuse space?
3. How to optimize variable allocation in the BEAM?

We study the performance of the reuse and garbage collector algorithms on a set of benchmarks. Our study raises interesting questions on the interplay between the garbage collector and the system caches. Our experience also allows us to propose new optimisation for variable allocation in the BEAM.

The paper is organised as follows. First, we present the basic concepts of the BEAM. Next, we focus on the memory management of the system, we explain the methods used to recover memory space and we present some performance results of BEAM running with different memory configurations. Finally, we propose a new variable allocation scheme that optimises memory usage and we evaluate the effectiveness of the proposed mechanism.

2 BEAM Concepts

We briefly present the BEAM, an implementation of the main concepts in Warren's Extended Andorra Model with Implicit Control, with several refinements [10, 13]. The BEAM model has been implemented for the Herbrand domain, although the EAM does support other constraint domains [17, 8].

A BEAM *computation* is a series of rewriting operations, performed on And-Or Trees. And-Or Trees contain two kinds of nodes: *and-boxes* represent a conjunction of positive literals, and store goals $G_1, \ldots, G_n$, new variables $X_1, \ldots, X_m$, and a set of constraints σ; or-boxes represent alternative clauses. Figure 1 shows an example of BEAM And-Or Tree. The top node is an and-box: it includes three local variables X, Y and Z. We say that a variable is *local* to an and-box Δ when first defined in Δ, and *external* to Δ otherwise. The and-box also includes two goals to execute, $parent(X, Y)$ and $parent(X, Z)$, and an empty set of constraints σ on external variables.

The two circles below corresponds to or-boxes. Or-boxes are created by the *reduction* rule, that given a sub-goal in an and-box generates a set of alternatives for that goal. Notice that or-boxes require relatively little information, except for the number of alternatives.

Last, the four and-boxes below are said to be suspended. In this case, we show only the constraints they are trying to impose on external variables. In general, an and-box Δ *suspends* if the computation on Δ cannot progress deterministically (when there is more than one successful candidate for the goal), and if Δ is trying to impose bindings on external variables. The BEAM thus follows the Andorra principle: it delays non-deterministic execution as long as possible.

Execution in the EAM thus proceeds as a sequence of rewrite operations on a And-Or Tree. The most important rules are based on Warren's original rules: goal

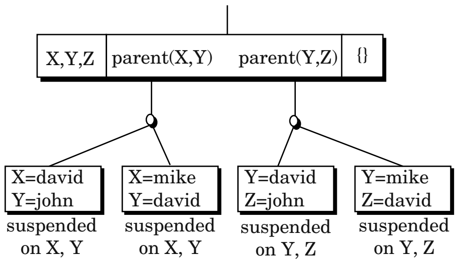


Fig. 1. An axample of BEAM and-or tree

reduction, upward propagation of deterministic bindings, downward propagation of new bindings on local variables, and non-deterministic reduction. These rules are designed to be correct and complete. Towards efficiency the BEAM also implements simplification rules that allow one to recognise failed or successful computation, and optimises common cases of computations (see [13]).

2.1 BEAM Architecture

Figure 2 illustrates the architecture organization for the BEAM execution model. The BEAM was built on top of the YAP Prolog system [10]. It reuses most of the YAP *compiler* and its *builtin library*. The shadowed boxes show where the EAM stores data. The *Code Space* holds the database, including the compiled logic program, information on predicates, and the symbol-table. The BEAM uses this data-area in much the same way as traditional Prolog Systems. This area is largely static during execution.

The *Global Memory* stores the And-Or Tree that is manipulated during the execution of logic programs. This area further subdivides into the *Heap* and the *Box Memory*. The *Box Memory* stores dynamic data structures including boxes and variables. The *Heap* uses term copying to store compound terms and is thus very similar to the WAM's *Heap* [18]. The major difference is that on the BEAM, *Heap* memory cannot be recovered after backtracking. A *garbage collector* is thus necessary to recover space in this area.

The *And-Or Tree Manager* handles most of the complexity in the EAM. It uses the *Code Space* area to determine how many alternatives a goal has and how many goals a clause calls. With this information the *And-Or Tree Manager* constructs the tree and uses the EAM rewriting rules to manipulate it. The Manager requests memory for the boxes from the *Global Memory Areas*. The *Emulator* is called by the *And-Or Tree Manager* in order to execute and unify the arguments of the goals and clauses. As an example consider the clause: `p(X,Y):- g(X), f(Y)`. When running this clause the *And-Or Tree Manager* transforms the `p(X,Y)` into one and-box, and calls the *Emulator* to create the subgoals and or-boxes for `g(X)` and `f(Y)`. Control returns to the *And-Or Tree Manager* if the boxes need to suspend.

More details on how BEAM stores the And-Or Tree, the design of the *Emulator* and of the *And-or Tree manager* can be found in [12].

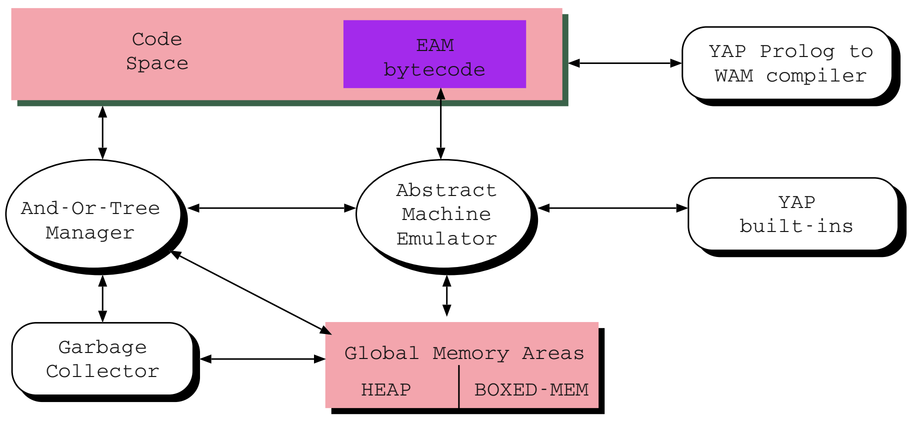


Fig. 2. Execution model

3 Memory Management

Standard WAM [5] implementations follow a stack discipline, where space can be recovered during backtracking. The EAM control strategy is much more flexible. The BEAM must carefully detect the points where to recover space. As we show next, we have two techniques to recover space: we can reuse space for pruned boxes and we can garbage collect useless data.

3.1 Reusing Space in the And-Or Tree

The *Box Memory* must satisfy intensive requests for the creation of and-boxes, or-boxes, local variables, external references, and suspension lists. Objects are small and most, but not all, will have short lifetimes. Objects are created very frequently and minimizing allocation and deallocation overheads is crucial.

In order to address this problem, Prolog systems traditionally use a stack based discipline and rely on backtracking and garbage collection to recover space [2, 4]. Unfortunately, we cannot recover space through backtracking. Instead we explicit maintain liveness of data structures, and rely on a hybrid memory allocation algorithm to allocate space:

1. We keep a top of stack and an array of n buckets, one per sizes i, $1 \leq i \leq n$, all initially set to empty.
2. To allocate a block we check whether bucket i has a free block. Otherwise we allocate the block from the top of the stack.
3. To release a block of size i we add it to the front of the list for bucket i.

The BEAM is therefore able to recover all memory from boxes whenever they fail or succeed. Memory from failed boxes can obviously be recovered since they do not add any knowledge to the computation. Memory from succeeding boxes can also be recovered because the BEAM unification rules, guarantee that and-box variables cannot reference variables within the box subtree, that is, younger box variables can reference variables in upper boxes, but not the other way around.

We have chosen this scheme because it has a low overhead and most requests tend to vary between a relative small number of sizes. To prove so, we have studied our algorithm with a small group of known benchmarks detailed on table 1. The first four benchmarks are deterministic, and the rest are non-deterministic.

Table 1. The benchmarks

Name	**Description**
nreverse	naive reverse of a 30-element list.
qsort	quick-sort of a 50-element list using difference lists.
kkqueens	smart finder of the solutions for the n-queens problem.
tak	heavily recursive with lots of simple integer arithmetic.
houses	logical puzzle based on constraints.
query	finds countries with approximately equal population density.
zebra	logical puzzle based on constraints.
scanner	a program to reveal the content of a box.
queens-9	finds all safe placements of 9-queens on $9 * 9$ chessboard.

In the `query` and `zebra` benchmarks, we also consider the version of the BEAM with eager splitting, *ES*. With eager splitting, producer goals reduce immediately, even if non-determinate, instead of delaying until all determinate operations have been implemented [7, 10].

Table 2 shows how our algorithm performs for the benchmark set. The *Memory requests* column shows how much memory was requested. The *Local Vars* column shows how much memory was spent on storing variables. This column shows that in most cases variable allocation takes a large percentage of total Box memory usage. The *Reuse* column shows the percentage of memory that was served from the free lists, and the *Memory used* shows the real size of memory used by each benchmark. The algorithm has almost no memory reuse for the deterministic `nreverse` benchmark, where the abstract machine itself reuses and-boxes through our implementation of the nested and-boxes simplification rule [13]. On the other hand, we achieve reuse rates greater than 90% in the `zebra`, and `queens-9` benchmarks. Hence, reuse is very effective in applications where the BEAM often prunes branches, such as in search applications, thus providing the advantages of backtracking to same extent.

To emphasize our point, note that in the `queens-9` benchmark, boxed memory had a total of 176Mb of memory requests, and about 98% of those requests were served from previously used memory. Thus we only required 3Mb of *Box Memory* throughout our computation.

3.2 Variable and Boxes

As we have seen, each and-box maintains a collection of local variables and constraints. Variables are represented as slots, and constraints are represented

Table 2. Box memory reuse

Benchmark	Requested	Local Vars	Reuses	Memory used
nreverse	33Kb	84%	4.13%	32Kb
qsort	82Kb	47%	59.51%	33Kb
kkqueens	41005Kb	46&	49.27%	20800Kb
tak	24599Kb	45%	72.47%	6773Kb
houses	558Kb	49%	84.02%	89Kb
query	2108Kb	7%	85.40%	308Kb
query-ES	435Kb	32%	90.14%	43Kb
zebra	5806Kb	38%	94.41%	325Kb
zebra-ES	2296Kb	38%	91.82%	188Kb
scanner	2687Kb	75%	64.86%	944Kb
queens-9	180140Kb	70%	98.34%	2982Kb

as handles to a memory-value pair. Note that the EAM deterministic promotion rule allows constraints to move upwards in the tree. It is interesting to explore how Box memory is used for variable storage.

Figure 3 details the BEAM's memory usage for the queens benchmark. The graphic shows the longevity (`X` axis) of boxes and of variables. We define longevity as the amount of time a box or a variable survives. We measure time in ticks, where each tick is a call to the memory allocator. We thus assume the number of calls to the memory allocator is constant throughout execution.

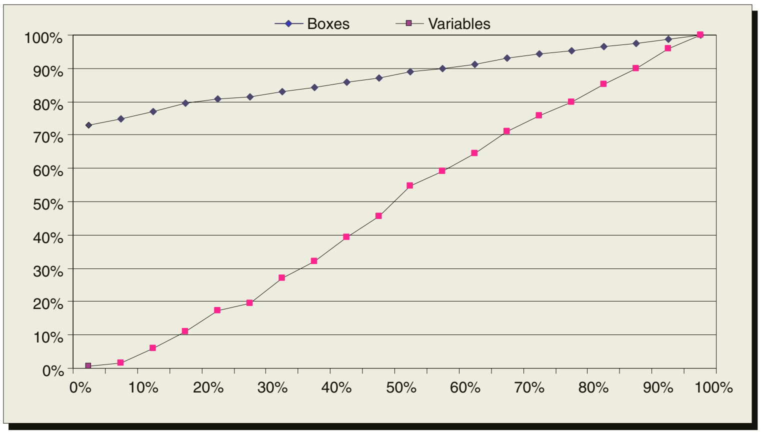


Fig. 3. Boxes and variables longevity

This graphic clearly shows that more than 70% of boxes live less than 5% of the runtime. In contrast, local variables are close to equally distributed: half of these variables live less than 50% of the runtime, and the other half lives longer.

We can conclude that, at least for this non-determinate benchmark, the liveness of boxes is usually very short, and much shorter than the duration of variables. Indeed, at the end, only a mere 2% of boxes survive more than 95% of the total execution time. On the other hand, local variables do seem to live longer.

In contrast to the memory usage for boxes, there are very few local variables to live less than 5% of the execution time. Moreover, most local variables seem to live for about half of the computation.

This graphic explains clearly why recovering box space is indeed fundamental to a EAM implementation. In this search application, most boxes live for a short time. Although local variables do live longer, a substantial percentage of space should thus be recoverable during the computation, as demonstrated in practice.

In practice, memory requests can still grow through the computation and exhaust the *Box Memory*. If the originally available *Box Memory* ends, the system can expand the Box Space by requesting more memory from the OS.

3.3 Recovering Heap Space

The algorithm used to reuse memory space in the *Box Memory* will not work for the *Heap* because the BEAM releases memory eagerly, and he released objects in the *Heap* tend to be very small, causing fragmentation and leaving only small blocks available. We could coalesce blocks to increase available block size [6], but the price would be an increase in overheads. Instead, we have chosen to rely on a garbage collector to compact the *Heap Memory*.

We implemented a *copying* garbage collector [9, 3] for the BEAM: living data structures are copied to a new memory area and the old memory area is released. The *Heap* memory is divided into two equal halves, growing in the same direction. The two halves could not grow in the opposite direction because the BEAM uses YAP builtins, and they expect the Heap to always grow upwards. Therefore we have a pre-defined limit-zone that, when reached, will activate the garbage collection mechanism by setting the garbage collector flag.

The garbage collection flag is periodically checked by the And-Or Tree manager to activate garbage collection.

Thus, the garbage collector starts by replicating the living data in the root of the And-Or Tree and then follows a top-down-leftmost approach.

The advantage of a *stop-and-copy* garbage collector implementation is that the execution time for the garbage collection is proportional to the amount of data in use by the system. In contrast, the mark-and-sweep garbage collection algorithm has execution time proportional to the original stacks. The main disadvantage of a copying algorithm is that the system can only use part of the total memory available. We do not use generational garbage collection [1, 15] because the percentage of garbage cells is very high, as we shall discuss next.

Table 3 shows how effective garbage collection is on average for `queens-9`. Essentially, the working size for this benchmark is around 40KB for *Heap*. Increasing stack space allows delaying garbage collection and results in very good effectiveness for the *Heap*. In general this behavior is similar on other programs. We can conclude that garbage collection is hus mostly useful at recovering *Heap* memory as we do not reuse *Heap* otherwise.

3.4 Performance Analysis

We have chosen the `queens-9` benchmark for a deeper analysis of BEAM memory management. We have experimented with different memory configurations:

Table 3. Average effect of garbage collection for `queens-9`

MEM Config	Heap before	Heap after	Heap recovered
1Mb	526,903	34,203	93.51%
2Mb	1,051,371	33,925	96.77%
4Mb	2,099,569	34,051	98.38%
8Mb	4,197,191	32,225	99.23%
16Mb	8,390,564	32,401	99.61%
32Mb	16,778,753	31,728	99.81%
64Mb	33,555,428	31,994	99.90%
128Mb	67,112,556	41,588	99.94%

– `BEAM-OnlyGC`: uses the garbage collector to recover memory in the Box Memory and Heap.
– `BEAM-HybGC`: use our hybrid algorithm (see section 3.1) to reuse memory in the Box Memory and the garbage collector to recover memory in the Heap.
– `BEAM-GCHybGC`: use our hybrid algorithm to reuse memory in the Box Memory and the garbage collector to recover memory on Heap. Moreover, in this version, whenever the garbage collector is recovering Heap memory, it also compacts the data on the Box Memory.

Table 4 shows the three different BEAM versions running the `queens-9` benchmark with several memory configurations. The time is presented in milliseconds, and the number of invocations to the garbage collector is presented in brackets `(X+Y)`. `X` being the number of garbage collections activated by *Box Memory* overflows and `Y` the number of garbage collections activated by *Heap* overflows. The timings were measured running the benchmark on a Intel Pentium-M Banias 1.6Ghz with 1024Kb *on chip* cache and on a Intel Pentium-4 Willamette 1.7Ghz with 256Kb *on chip* cache. Both systems have a 4*100Mhz FSB, were equipped with 1Gb RAM and running Mandrake Linux 10. Note that, although `queens-9` is an example where the BEAM can outperform Prolog by limiting the search space [12], the BEAM still depends on garbage collection to run this benchmark on most of the memory configurations used.

First, some considerations about the memory configurations. All three systems used the same amount of Heap memory on each run. Note that 128MB for the heap means that the system is using two halves of 64Mb alternately, switching during garbage collection. `BEAM-OnlyGC` reserves for the box memory the same amount available for the Heap. `BEAM-HybGC` runs with only 3Mb for the Box Memory, while the `BEAM-GCHybGC` runs with 6Mb for the Box memory (again using only 3Mb alternately between garbage collections). Finally, we should remark that garbage collections on `BEAM-OnlyGC` were activated by *Box Memory* overflows, and that on `BEAM-HybGC` and `BEAM-GCHybGC` the garbage collections were activated by *Heap* overflows.

The table entry with `400Mb` stack size is specially important because all systems can run without garbage collections, allowing us to observe the overhead

Table 4. BEAM running queens-9 with different memory configurations

queeens9		**BEAM-OnlyGC**		**BEAM-HybGC**		**BEAM-GCHybGC**	
	Heap Mem	Time	GC	Time	GC	Time	GC
	1Mb	1,572	(382+0)	**1,320**	(0+253)	1,379	(0+253)
	2Mb	1,492	(182+0)	**1,290**	(0+122)	1,342	(0+122)
1	4Mb	1,429	(89+0)	**1,293**	(0+60)	1,314	(0+60)
0	8Mb	1,396	(44+0)	1,289	(0+30)	**1,284**	(0+30)
2	16Mb	1,373	(22+0)	1,288	(0+14)	**1,271**	(0+14)
4	32Mb	1,349	(11+0)	1,282	(0+7)	**1,262**	(0+7)
Kb	64Mb	1,310	(5+0)	1,282	(0+3)	**1,272**	(0+3)
L2	128Mb	1,271	(2+0)	1,272	(0+1)	**1,266**	(0+1)
	256Mb	**1,240**	(1+0)	1,265	(0)	1,262	(0)
	400Mb	**1,228**	(0)	1,265	(0)	1,262	(0)
	1Mb	2,050	(382+0)	**1,909**	(0+253)	1,960	(0+253)
	2Mb	1,854	(182+0)	1,862	(0+122)	**1,842**	(0+122)
2	4Mb	**1,761**	(89+0)	1,849	(0+60)	1,783	(0+60)
5	8Mb	**1,720**	(44+0)	1,825	(0+30)	1,770	(0+30)
6	16Mb	**1,705**	(22+0)	1,833	(0+14)	1,771	(0+14)
Kb	**32Mb**	**1,695**	(11+0)	1,843	(0+7)	1,783	(0+7)
	64Mb	**1,699**	(5+0)	1,840	(0+3)	1,783	(0+3)
L2	128Mb	**1,728**	(2+0)	1,840	(0+1)	1,827	(0+1)
	256Mb	**1,698**	(1+0)	1,820	(0)	1,819	(0)
	400Mb	**1,720**	(0)	1,820	(0)	1,819	(0)

that our Hybrid algorithm for reusing space in the box memory has on the system. Indeed, from these results is it possible to observe that the algorithm implies a minor 3% to 5% slowdown in the system. Not a big price to pay attending that it allow us to run the `queens-9` benchmark with only 3Mb of memory whereas without it, the system requires 400Mb for the box memory.

It is interesting to compare the results obtained with the two machines with similar architectures, but different caches. Note that in both machines, the system will run the garbage collector the same number of times. It is interesting to analyze the two extreme data-points: *256Kb L2* versus *1024Kb L2*. The numbers show that, although the *Pentium-M* has a slower CPU clock cycle, its L2 cache makes the day. A second interesting remark is that the `BEAM-OnlyGC` when running in the smaller cache CPU, beats the other two versions in almost every run. This result shows that having the garbage collector compacting more often the living data structures in memory can have advantages for lower specs CPU's with less cache. Moreover, on the *1024Kb L2* CPU, the throne is divided. On the higher stack sizes, the `BEAM-OnlyGC` is faster, which results, as we have already stated, from the overhead the other two systems have in using the algorithm to reuse space in the Box memory. On the lower stack sizes (4Mb and less), the `BEAM-HybGC`, behaves better, meaning that the cache size still compensates for the extra work the other systems do in compacting the Box memory. Finally, on the other stack sizes, `BEAM-GCHybGC` behaves better. These results are interesting specially when comparing BEAM-GCHybGC and BEAM-HybGC. These two versions differ only in the fact that the BEAM-GCHybGC, compacts the living structures in the box memory during garbage collection. Having better performance on the BEAM-GCHybGC shows that, despite doing more code, having the data structures compacted can benefit in the long run. Moreover, this raises the question of how much the L2 caches affect general system performance.

Our main conclusion is that the impact of garbage collection on execution time is not very significant on this example. Indeed, garbage collection can be advantageous due to the fact that it compacts the living data structures, allowing the cache to behave better (confirming results previously observed on [11]). Moreover, even with more code to execute, BEAM-GCHybGC performed in most cases better than the BEAM-HybGC version. This results also confirm our approach of using a simple memory allocator leaving compression for the garbage collector.

4 Optimizing Variable Allocation

Variables are a major source of memory pressure. In the initial implementation of the BEAM, all variables were processed the same way. Every and-box maintains a list of its local variables, and every variable would be in some and-box. Lets refer to these variables as *permanent* variables.

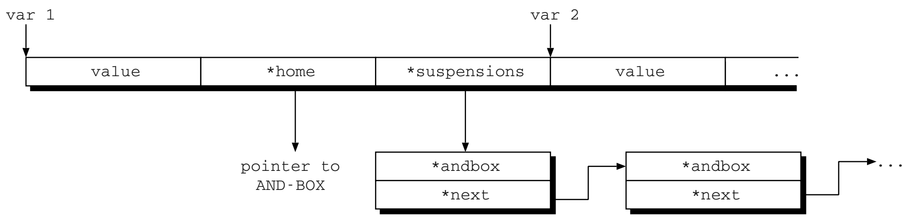


Fig. 4. Local variables representation

The actual implementation is illustrated in Figure 4. A variable either belongs to a single subgoal in an and-box, or it is shared between subgoals. The value field stores the current working value for a variable. Variables also maintain a list of and-boxes suspended on them, and points back to their home and-box. The suspensions field is important because, whenever a local variable is constrained, binding propagation is performed by sending a signal to all and-boxes that are suspended on that variable. All and-boxes that received the signal become aware that they must perform a consistency check of the constraints recently generated with their constraints composing their environment. The home field determines whether a variable is local or external to an and-box.

4.1 Compile-Time Variable Classification

Processing all variables the same way has major drawbacks. Namely, during the execution of a program there is a large portion of memory that can only be released when the and-boxes fail or succeed. This problem was evidenced in Figure 3. Boxes have a very short lifetime, while variables live longer.

The complexity of variable implementation can also harm system performance. Consider one of the main rules of the EAM, *P*romotion, used to promote the variables and constraints from an and-box Δ to the nearest and-box Δ' above. Δ must be a single alternative to the parent or-box, as shown in Fig. 5.

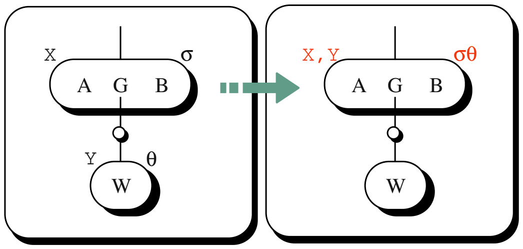


Fig. 5. BEAM promotion rule

As in the original EAM promotion rule, promotion propagates results from a local computation to the level above. However, promotion in the BEAM does not merge the two and-boxes because the structure of the computation may be required towards pruning [14].

During the promotion of *permanent* variables, the field *home* field of the variable structure needs to be updated so that it points to the new and-box Δ'. There is an overhead in this operation since one must go through the list of all *permanent* variables of Δ. Moreover if Δ' is promoted later, the system will have go through Δ' variables including all that it has inherited during promotions. Thus, is obvious that the list of *permanent* variables can grow very fast when promoting boxes, slowing down the BEAM.

Therefore, we would like to classify some variables as *temporary*, meaning that they would be used to create an and-box, and that they do no need support for suspension. Doing so would save memory and gain in performance since managing *temporary* variables is simpler than dealing with *permanent* variables.

4.2 Classification of Variables at Compile and Run-Time

Unfortunately, in general we do not know beforehand if we will need to suspend on a variable. Next, we propose a WAM-inspired scheme, the **BEAM-Lazy**. Like in the WAM, variables that appear only in the bodies of clauses or in queries are classified at compile time as *permanent* variables, meaning that all data-structures required for suspension are created for them. All others variables are classified on compile time as *temporary*. Further, variables that are needed to create compound terms in write mode need not to be *permanent*.

As an example, consider the second clause of the nreverse:

```
nreverse([X|L0],L) :- nreverse(L0,L1), concatenate(L1,[X],L).
```

For this clause, `L1` is the only variable that is classified as *permanent* at compilation time. The other variables are classified as *temporary*. Thus, whenever BEAM creates a and-box for this clause, it will only need to create one *permanent* variable and three *temporary* variables. One, it may need to create two more *permanent* variables, `X` and `L0`, when the clause is called with the first argument as variable (`unify_var` in write mode).

The advantage of this classification, is that classifying a group of variables as *temporary* requires less memory and also improves performance since we avoid managing the more complex structure of the *permanent* variables. Another advantage gained when using *temporary* variables is that they can be immediately released after executing all the `put` instructions in the clause body, unlike *permanent* variables that can only be released when the and-box succeeds or fails. Most frequently, *temporary* variables will be released immediately before the call for the last subgoal, acting similarly to the `deallocate` of environments in WAM implementations.

One disadvantage of this classification of variables, is that the implementation of the instructions in the abstract machine is more complex, with instructions requiring extra tests to determine if the system needs to create new *permanent* variables. Also, when deciding if the memory used by the *temporary* variables can be released, there is an extra overhead for the system. Note that while WAM Prolog systems follow a static goal order, the BEAM may choose to execute the goals in a different order. Thus, the last goal may execute before previous goals, so the rule of releasing the environment before calling the last goal is not sufficient and cannot be used. The system must perform a test within every call, to determine if the goal is the last one being executed within the clause.

4.3 Variable Unification Rules

The main consideration in implementing a unification algorithm that supports both types of variables is that an and-box only suspends when trying to bind one or more *permanent* variables external to the and-box. Some of these bindings can be generated when unifying two variables. Although the EAM proposal did not present details on variable to variable unification, the BEAM, as the WAM, has an optimal form to unify two different variables in a EAM environment.

There are three possible cases of variable to variable binding:

1. *temporary variable* to *permanent variable*: in this case the unification should makes the *temporary* variable refer to the *permanent* variable. An immediate advantage is that the computation may not suspend. Moreover, unifying in the opposite direction may lead to an incorrect state, since future dereferencing of the *permanent* variable would reference a *temporary* variable that can unify without suspending the computation.
2. *temporary variable* to *temporary variable*: this case may never occur as whenever these variables are created they should, immediately unify with a *permanent* variable. Thus, using the previous rule, a *temporary* variable is always guaranteed to reference a *permanent* variable.

3. *permanent variable* to *permanent variable*: the *permanent* variable that has its home box at a lower level of the tree should always reference the *permanent* variable that has its home box closest to the root of the tree.

By following these unification rules one can often delay the suspension of an and-box and thus delay application of the splitting rule. Note that Splitting is the most expensive operation and in most cases we want to avoid it.

4.4 Results

This section compares the two schemes presented to classify variables in a EAM environment. Our initial goal is to evaluate the memory usage of the new proposed algorithm for classifying variables and to determine if there is an overhead in the system due to the added complexity.

Since the analyses performed in section 3.4 have shown that the BEAM is a memory intensive-system and its performance may largely depend on the target machine's cache/memory subsystem, we have chosen to use the Pentium-M system with 1Mb of cache to run the tests. We have used the same set of benchmarks presented in section Table 1. We recall that this set of benchmarks is divided into deterministic programs: `nreverse`, `qsort`, `kkqueens` and `tak`), and non-deterministic programs: `houses`, `query`, `zebra`, `scanner` and `queens`.

Table 5. BEAM-All vs BEAM-Lazy

	BEAM-All			**BEAM-Lazy**			
		Box Memory			**Box Memory**		
Bench.	**Heap**	**Requests**	**Reuses**	**Heap**	**Requests**	**Reuses**	**Perf.**
nreverse	3.87Kb	33Kb	4.13%	3.87Kb	18Kb	50.43%	-3.13%
nreverse-1000	3918Kb	33314Kb	0.14%	3918Kb	17650Kb	55.40%	-4.60%
qsort	4.3Kb	82Kb	59.51%	4.3Kb	61Kb	61.72%	0%
qsort-1000	7820Kb	179742Kb	69.51%	7820Kb	140648Kb	66.62%	-1.83%
kkqueens	3323Kb	41005Kb	49.27%	3323Kb	28730Kb	73.46%	2.50%
tak	559Kb	24599Kb	72.47%	559Kb	17642Kb	74.99%	0%
houses	153Kb	558Kb	84.02%	23Kb	278Kb	77.01%	133%
query	84Kb	2108Kb	85.40%	104Kb	2081Kb	71.86%	0%
query-ES	42Kb	435Kb	90.14%	61Kb	409Kb	83.93%	-18%
zebra	4211Kb	5806Kb	94.41%	158Kb	4029Kb	60.65%	123%
zebra-ES	527Kb	2296Kb	91.82%	90Kb	1713Kb	79.47%	31%
scanner	5861Kb	2687Kb	64.86%	2510Kb	1121Kb	51.31%	121%
queens-9	122143Kb	180140Kb	98.34%	66113Kb	86834Kb	67.24%	76%

Results are shown in table 5. We consider the two versions of BEAM: `BEAM-All` that classifies all the variables on compile-time as *permanent* and `BEAM-Lazy` that classifies the variables on compile and run-time. For each version we present three columns: the Heap memory used, the Box memory requested for boxes and variables and the percentage of those requests served with reused memory. The last column evaluates the performance of the two systems. A positive value represents that `BEAM-Lazy` is faster and a negative value that is slower.

Results show that the memory requests on Box Memory is significatively less for `BEAM-Lazy`. Moreover, the memory used on the Heap is also favorably influenced. One may wonder how creating less *permanent* variables reduces Heap usage. When splitting, a subset of the and-or-tree is copied to another location of the and-or-tree replicating living structures. Reducing the number of *permanent* variables, that as we have shown live longer than the *temporary* variables, reduces of the number of compound terms that need to be replicated in the Heap during the split operation. On deterministic programs, where no splitting is executed, the heap memory used is equal for both versions.

There are few cases where the memory requirements have increased. This increase happens when there is a high number of variables being classified as *permanent* on run-time. Note that initially `BEAM-Lazy` reserves only a pointer for variables that later are classified as *permanent*. Later, when classifying the vars as *permanent*, the abstract machine requests a new structure that uses more 3 machine words. Thus it requests a total of $1 + 3$ machine words for these variables. Contrarily, `BEAM-All` requests immediately the *permanent* variable structure when creating the and-box for the goal, thus only requiring 3 machine words for each variable.

The performance numbers show that on deterministic benchmarks (`nreverse`, `qsort`, `kkqueens` and `tak`), `BEAM-Lazy` has a small slow down, which is due to the extra complexity of the abstract machine. On the other hand, on non-deterministic programs, except for `query`, `BEAM-Lazy` has huge improvements. Indeed, on `houses`, `zebra` and `scanner`, the system is more than twice faster.

5 Conclusions

We discuss several memory allocation issues in the BEAM, a first prototype implementation of the Extended Andorra Model with Implicit Control. We rely on two major data areas, the Box Memory and the Heap. Our results show that box memory can be reused quite effectively by using a bucket-based memory allocator, and our garbage collector is very effective at managing Heap space. We observed that a significant amount of Box space is used to store variables. We propose a finer variable allocation scheme to reduce memory overheads. Our results show that the new scheme can be quite effective at reducing memory pressure, with only a small overhead.

Costs might be reduced further if we could know at variable creation time that a variable is always going to be bound deterministically, or if we would know that we are going to bound the variable with an older variable. In some cases we can do so from by looking at the indexing code. In general, to do so would require global analysis of the logic program, and we are studying how global analysis tools can be applied to this problem.

We are also working in integrating the BEAM with the current version of the YAP Prolog system so that it can be made more widely available.

Acknowledgments

The work presented in this paper has been partially supported by project APRIL (Project POSI/SRI/40749/2001) and funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*.

References

1. K. A. M. Ali. A Simple Generational Real-Time Garbage Collection Scheme. *New Generation Computing*, 16(2):201–221, 1998.
2. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):171–183, 1989.
3. J. Bevemyr and T. Lindgren. A simple and efficient copying garbage collector for Prolog. *Lecture Notes in Computer Science*, 844:88–101, 1994.
4. L. F. Castro and V. Santos Costa. Understanding Memory Management in Prolog Systems. In *Proceedings of ICLP'01*, November 2001.
5. B. Demoen and P. Nguyen. So Many WAM Variations, So Little Time. In *LNAI 1861, Proceedings Computational Logic*, pages 1240–1254. Springer, July 2000.
6. D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software, Practice and Experience*, 24(6), 1994.
7. G. Gupta and D. H. D. Warren. An Interpreter for the Extended Andorra Model. Technical report, Dep. of Computer Science, University of Bristol, November 1991.
8. J. Jaffar and M. Maher. Constraint Logic Programming: a Survey. *The Journal of Logic Programming*, 19/20, May/July 1994.
9. R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, July 1996. Reprinted February 1997.
10. R. Lopes. *An Implementation of the Extended Andorra Model.* PhD thesis, Universidade do Porto, December 2001.
11. R. Lopes, L. Castro, and V. Costa. From Simulation to Pratice: Cache Performance Study of a Prolog Systems. *ACM SIGPLAN Notices*, 38(2):56–64, Feb. 2003.
12. R. Lopes, V. S. Costa, and F. Silva. A novel implementation of the extended andorra model. In *Pratical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 199–213. Springer-Verlag, March 2001.
13. R. Lopes, V. S. Costa, and F. Silva. On deterministic computations in the extended andorra model. In *19th International Conference on Logic Programming, ICLP 2003*, volume 2916 of *LNCS*, pages 407–421. Springer-Verlag, Dec. 2003.
14. R. Lopes, V. Santos Costa, and F. Silva. Prunning in the extended andorra model. In *Pratical Aspects of Declarative Languages:6th International Symposium*, volume 3057 of *LNCS*, pages 120–134. Springer-Verlag, June 2004.
15. T. Ozawa, A. Hosoi, and A. Hattori. Generation Type Garbage Collection for Parallel Logic Languages. In *Proceedings of the North American Conference on Logic Programming*, pages 291–305. MIT Press, October 1990.
16. V. Santos Costa. Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.
17. E. Shapiro. The family of Concurrent Logic Programming Languages. *ACM computing surveys*, 21(3):412–510, 1989.
18. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.
19. D. H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

Solving Constraints on Sets of Spatial Objects*

Jesús M. Almendros-Jiménez and Antonio Corral

Dpto. de Lenguajes y Computación, Universidad de Almería
{jalmen,acorral}@ual.es

Abstract. In this paper, we present a *constraint solver* for *constraints on sets of spatial objects*. With this aim, we define a constraint system for handling *spatial data types* (points, lines, polygons and regions) and constraints on them (equalities and inequalities, memberships, metric, topological and structural constraints), and provide a suitable theory for this constraint system. The constraint solver is presented in the form of *transformation rules*. These transformation rules handle a special kind of constraints used for *consistency checking*, enabling an *optimized and efficient* solving of spatial constraints.

1 Introduction

In the field of *Constraint Programming* [2], a wide research has been done in order to define specialized constraint systems for particular problems. For instance, *linear equations and inequations over reals* [12] for spatial objects handling, *boolean constraints* [5] for circuit structure reasoning, *linear constraints on integer intervals* [10] for combinatorial discrete problems, and *temporal constraints* [19] for time and scheduling reasoning. For each one of these constraint systems both generic and particular constraint solver techniques have been studied in order to improve the *searching of solutions*. They typically are based on *heuristics*, *backtracking* and *branch and bound algorithms*. Several techniques like *constraint propagation* achieving *local*, *arc* and *path* consistency, among others, ensure a good performance of *constraint solvers*.

The framework of *constraint databases* [11] is based on the simple idea that a constraint can be seen as an extension of a relational tuple, called *generalized tuple*, or, vice versa, that a relational tuple can be interpreted as a conjunction of equality constraints. Each generalized tuple finitely represents a possibly infinite set of relational tuples, called *extension* of the generalized tuple, one for each solution of the constraint. As far as we know spatial constraint databases have been focused on *linear constraints* handling [16, 15, 4] in which spatial reasoning is based on linear equations and inequations. The basic idea is to see spatial objects as infinite sets of points and provide a finite representation of these infinite sets, with constraints over a dense domain. Most known spatial data types like point, line and region can be modeled with these frameworks, and distance-based operations can be included on it [3].

* This work has been partially supported by the Spanish project of the Ministry of Science and Technology, "INDALOG" TIC2002-03968 under FEDER funds.

However, in our opinion, an interesting direction could be the study of other kinds of constraint systems, *specialized for any class of spatial data types*, but suitable for practical applications. In particular, we are interested in modeling problems on *sets of spatial objects*, where each spatial object can be a *point*, *line*, *polygon*, or a *region*. In addition, the *query language* could handle *metric* and *topological queries* on these object sets.

In this paper, we present a *constraint solver* for *constraints on sets of spatial objects* (points, lines, polygons and regions). With this aim, we will present a constraint system for handling these *spatial data types* and constraints on them (equalities and inequalities, memberships, metric, topological and structural constraints). We will provide a suitable theory for this constraint system. The constraint solver will be presented in the form of *transformation rules*. These transformation rules handle a special kind of constraints used for *consistency checking*, enabling an *optimized and efficient* solving of spatial constraints. This consistency checking is based on solving of *constraints on real intervals*. The basis of the optimization of the constraint solver is the use of the so-called *Minimum Bounded Rectangles (MBR's)* for approximating objects, and sets of objects organized in data structures called *R-trees* [8]. An MBR is a rectangle with faces parallel to the coordinate axis, which approximates the exact geometrical representation of the spatial object in two points. The constraint solver uses this representation for using *branch and bound techniques* in order to solve queries on sets of spatial objects. Our framework is close to the research on spatial databases focused on sets of spatial objects [7, 17, 6, 18, 14], which is a widely investigated area. Some attempt to compare this framework with constraint programming has been studied in [13] (for topological relations) however a more general approach is still an open topic of research.

The structure of the paper is as follows. Section 2 presents a set of examples of constraint satisfaction problems using spatial data. Section 3 formally defines the set of spatial data types together with the set of operations defined on them. Section 4 formally defines a spatial constraint satisfaction problem and its solved form. Section 5 presents the constraint solver and finally, section 6 concludes and presents future work.

2 Examples of Spatial CSP's

Basically, we handle sets of spatial objects where a spatial object can be a point, line, polygon or region. Points will be represented by means of pairs of real and variable coordinates: $(\mathtt{1},\mathtt{2})$ and $(\mathtt{x},\mathtt{1})$; lines by means of an ordered sequence of points: $\mathtt{line}[(\mathtt{1},\mathtt{2}),(\mathtt{4},\mathtt{5}),(\mathtt{9},\mathtt{10})]$; and the same can be said for polygons and regions, where the points define a closed path (i.e. last point is connected to the first one). Lines, polygons and regions are not intended to contain more vertices.

With this representation we can describe sets of *closed points-based geometric figures*: $\{(\mathtt{8},\mathtt{1}),\mathtt{line}[(\mathtt{1},\mathtt{2}),(\mathtt{4},\mathtt{5}),(\mathtt{9},\mathtt{10})\}$. On these spatial data types, a set of binary spatial operations is defined. We can classify them in five groups: equality and membership relations, set operations, metric operations, topological relations, and finally, structural operations. We are not interested in a complete set of operations, but enough for most usual spatial queries.

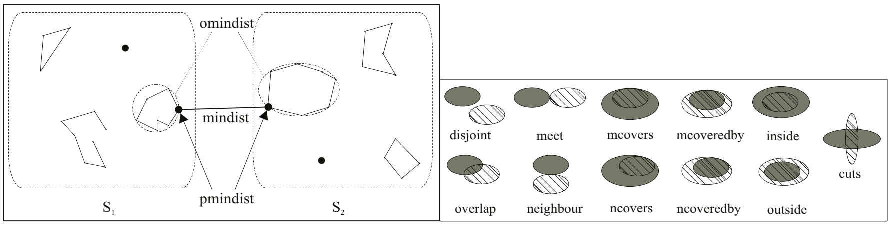


Fig. 1. Metric and Topological Operations

With respect to equality relations, spatial objects can be compared, for instance $\texttt{(1,2)} = \texttt{(2,3)}$, $\texttt{(1,x)} = \texttt{(1,9)}$, $\texttt{line[(1,2),(10,y)]} = \texttt{line[(z,2),(10,12)]}$. The first equality relation is false, but the other two equality relations define a relation on x, y and z, which can be seen as a constraint, that is, a requirement that states which combinations of values from the variable domains are admitted. In this case, x equal to 9, y equal to 12 and z equal to 1. This can be expressed, at the same time, as constraints of the form $\texttt{x} = \texttt{9}, \texttt{y} = \texttt{12}, \texttt{z} = \texttt{1}$.

With respect to the membership relations, we can express membership of points to a spatial object, for instance $\texttt{(1,6)} \in \texttt{line[(1,2),(1,9)]}$, which is trivially true, but we can also add variables. However, we can also use the membership relation for expressing that a spatial object belongs to a set of spatial objects. For instance, $\texttt{A} \in \{\texttt{line[(1,2),(1,9)]}, \texttt{(9,10)}\}$. In this case, A ranges on two spatial objects, line[(1,2), (1,9)] and (9,10). Combining membership relations we can build our first CSP using spatial data. For instance, $\texttt{(1,x)} \in \texttt{A}, \texttt{A} \in \{\texttt{line[(1,2),(1,9)]}, \texttt{(1,10)}\}$. The allowed set of values for x is then [2,9] and 10. This can be expressed by means of two constraints $\texttt{x} \in [\texttt{2},\texttt{9}]$ and $\texttt{x} = \texttt{10}$. Usually, in constraint programming, a disjunction can be used for expressing both cases in a sole constraint: $\texttt{x} \in [\texttt{2},\texttt{9}], \texttt{A} = \texttt{line[(1,2),(1,9)]} \vee \texttt{x} = \texttt{10}, \texttt{A} = \texttt{(1,10)}$. This written form can be seen as a *solved form* or *answer* to the constraint satisfaction problem.

With respect to the set operations, we consider *union*, *intersection*, and *difference*. We have two cases, for spatial objects and sets of spatial objects. For instance, $\texttt{(1,2)} \cup \texttt{line[(1,2),(9,10)]}$ represents the set of points belonging to both spatial objects. Therefore, the constraint $\texttt{(1,x)} \in \texttt{(1,2)} \cup \texttt{line[(1,2),(9,10)]}$ has a solved form $\texttt{x} = \texttt{2}$. An accurate treatment needs the *difference*. When the difference involves two spatial objects, the result of performing such operation can be a spatial object with holes. For this reason, we allow solved forms in our framework, including geometric figures like $\texttt{line[(1,2), (9,10)]} - \texttt{(1,2)}$ representing a *closed points-based geometric figure with holes*. These holes are an arbitrary union of closed points-based geometric figures.

The third kind of operations are the metric ones (see figure 1). They are based on distances, and also handle spatial objects and sets of spatial objects. We can suppose a fixed distance $d(\_,\_)$ for pairs of points: Euclidean, Manhattan, etc. The fixed distance is not relevant, and we can suppose any of them. In this case we have mindist, maxdist, pmindist, pmaxdist, omindist and omaxdist. The first two represent the minimum and maximum distance between two spatial objects. For instance $\texttt{x} = \texttt{mindist}(\texttt{line}[(1,2),\ (1,10)], \texttt{line}[(3,2),\ (12,15)])$, is a constraint with solved form $\texttt{x} = 3$. Distance-based queries have a particularity. They are *rigid* operations, that is, objects for which a minimum (maximum) distance is computed must be a fully defined object, avoiding the use of ranged variables in their description. For instance, it is forbidden to formulate a constraint satisfaction problem of the form $\texttt{x} = \texttt{mindist}((1,\texttt{y}),(1,1)), \texttt{y} \in [1,3]$. The operations `pmindist` and `pmaxdist` (resp. `omindist` and `omaxdist`) get the set of pairs of points (resp. objects) with the minimum (resp. maximum) distance. For instance, $\texttt{P} \in \texttt{pmindist}(\texttt{line}[(1,2),\ (1,10)],\ \texttt{line}[(3,2),\ (12,15)])$, requires the variable P, a new kind of variable, to be a member of a set of pairs of spatial objects. This set represents the set of pairs of points which are at the minimum distance. The solved form will be $\texttt{P} \in \{\{< (1,2),(3,2) >\}\}$. Therefore, we have to handle in addition to set of objects, set of pairs of objects, using a couple of brackets as notation.

With respect to topological relations (see figure 1), they are usual binary topological relations on spatial objects and set of spatial objects. For instance, $\texttt{A} \in \{(1,4),(4,5)\}, \texttt{B} \in \{\texttt{line}[(1,4),(1,10)],(4,5)\}, \texttt{A inside B}$ has as solved form $\texttt{A} = (1,4), \texttt{B} = \texttt{line}[\ (1,4),(1,10)] \vee \texttt{A} = (4,5), \texttt{B} = (4,5)$. Analogously, the constraint $\texttt{P} \in \{(1,4),(4,5)\}\ \texttt{inside}\ \ \{\texttt{line}[\ (1,4),(1,10)],(4,5)\}$ has as solved form $\texttt{P} \in \{\{< (1,4), \texttt{line}[\ (1,4),\ (1,10)] >, < (4,5),\ (4,5) >\}\}$. A special case is `rangeq`, which is a topological operation for range queries, such as the occurrence of a spatial object or a set of spatial objects that fall on a certain distance from a point or on a window. With this aim, we consider as special case of spatial data types, the circle defined by a point and a given radius (`circle`) and a window given by two points (`window`). For instance, $\texttt{A} \in \{(1,2),(9,10)\}$ $\texttt{rangeq circle}((0,0),3)$ has as solved form $\texttt{A} = (1,2)$.

In addition, we have considered a set of structural operations in order to compute the size of spatial objects, and the set of points and line segments which conform the frontier of a spatial object (and a set of spatial objects). They are `setpoints`, `setsegments`, `area`, `length` and `perimeter`.

Finally, our constraint system can handle *negation*, but with restricted use. Equality and membership relations can be used in negative form $\neq$ and $\notin$, but variables used in them (which can be referred to distances, coordinates of points, spatial objects and pairs of spatial objects) must be ranged in *at least one positive constraint* in order to formulate a constraint satisfaction problem. This restricted version of negation prevents anomalies in the use of negation such as infinite answers for a constraint satisfaction problem. For instance, $\texttt{C} = \texttt{A} \cap \texttt{B}, \texttt{A} \notin \{(1,2),(3,4)\}, \texttt{B} \notin \{(1,2)\}$ cannot be represented in solved form as a finite disjunction of equality and membership relations for C, A and B. The problem is that there is a *infinite and not finitely representable set of solutions.*

Therefore, and in order to ensure that *each constraint satisfaction problem has an equivalent solved form as a disjunction of equality and membership relations*, we require that the constraint satisfaction problem is formulated as follows. Each variable, representing a distance, coordinates of points, spatial objects or pair of spatial object, must be ranged in an interval of real numbers, spatial object, sets of spatial objects or sets of pairs of spatial objects, respectively. This (syntactic) condition of range restriction, called *safety condition*, and required over the constraint satisfaction problems, is usual in constraint databases [15].

Now, we show an example of spatial data to explain the use of our framework for processing spatial queries as Constraint Satisfaction Problems (CSP's). In this example, we have spatial information (non-spatial data can also be managed (e.g. Name, Population, Length, etc.), but we are only interested in its spatial shape) about Cities (set of points), Tourist Resorts (set of points), Roads (set of lines), River (line) and Lake (polygon). The figure 2 illustrates the spatial data, and assuming the work area in the window (0, 0) for lower-left corner and (80, 60) for the upper-right corner, the coordinates of the spatial components (expressed as constraints) are the following: Cities: $\mathtt{C1} = (16, 24)$, $\mathtt{C2} = (27, 41)$, $\mathtt{C3} = (40, 28)$, $\mathtt{C4} = (50, 10)$, $\mathtt{C5} = (60, 32)$, $\mathtt{C6} = (75, 55)$; Tourist Resorts: $\mathtt{T1} = (12, 16)$, $\mathtt{T2} = (23, 37)$, $\mathtt{T3} = (57, 6)$; $\mathtt{Road} \in \{\mathtt{line}[\mathtt{C6}, \mathtt{C5}, \mathtt{C3}], \mathtt{line}[\mathtt{C6}, \mathtt{C3}, \mathtt{C1}, \mathtt{T1}], \mathtt{line}[\mathtt{C5}, \mathtt{C4}, \mathtt{C1}], \mathtt{line}[\mathtt{T3}, \mathtt{C4}, \mathtt{C3}, \mathtt{C2}, \mathtt{T2}]\}$; $\mathtt{River} = \mathtt{line}[(52, 60), (60, 41), (50, 16), (0, 28)]$; and $\mathtt{Lake} = \mathtt{polygon}[(64, 38), (62, 35), (60, 36), (60, 34), (56, 36), (58, 40), (62, 42)]$. Note that the road is expressed as a set of line segments due to we cannot handle graphs in our representation. The following spatial queries (topological, distance-based and directional) are expressed as a CSP as follows, where capitalized names represent variables:

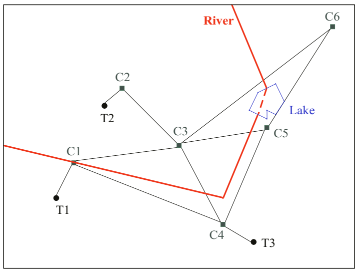


Fig. 2. Spatial Example

	Query	CSP	Answer
(1)	List the part of the river that passes through the lake	$\mathtt{Parts} \in \mathtt{setsegments}(\mathtt{River})$ $\mathtt{Through} = \mathtt{Parts} \cap \mathtt{Lake}$	$\mathtt{Through} = \mathtt{line}[(60, 41), (56, 35)]$ $\mathtt{Parts} = \mathtt{line}[(60, 41), (50, 16)]$
(2)	Report the part of the road that touches the lake and its length	$\mathtt{Parts} \in \mathtt{setsegments}(\mathtt{Road})$ $\mathtt{Touches} = \mathtt{Parts} \cap \mathtt{Lake}$ $\mathtt{Parts\ neighbor\ Lake}$ $\mathtt{L} = \mathtt{length}(\mathtt{Touches}), \mathtt{L} \neq 0$	$\mathtt{Touches} = \mathtt{line}[(64, 36), (62, 35)]$ $\mathtt{Parts} = \mathtt{line}[\mathtt{C6}, \mathtt{C5}], \mathtt{L} = 3.5$
(3)	Report all the cities that can use water from the river (at most within 10 km)	$\mathtt{City} \in \{\mathtt{C1}, \mathtt{C2}, \mathtt{C3}, \mathtt{C4}, \mathtt{C5}, \mathtt{C6}\}$ $\mathtt{D} = \mathtt{mindist}(\mathtt{City}, \mathtt{River})$ $\mathtt{D} \in [0, 10]$	$\mathtt{D} = 0, \mathtt{City} = \mathtt{C1} \vee$ $\mathtt{D} = 9, \mathtt{City} = \mathtt{C3} \vee$ $\mathtt{D} = 6, \mathtt{City} = \mathtt{C4} \vee$ $\mathtt{D} = 3, \mathtt{City} = \mathtt{C5}$
(4)	Find tourist resorts that are within 7.5 km of a city	$\mathtt{Near} \in \{\mathtt{T1}, \mathtt{T2}, \mathtt{T3}\}\ \mathtt{rangeq}$ $\mathtt{circle}(\mathtt{City}, 7.5)$	$\mathtt{Near} = \mathtt{T2}, \mathtt{City} = \mathtt{C2}$
(5)	Which city is the closest to any tourist resort?	$\mathtt{Closest} \in \mathtt{pmindist}(\{\mathtt{C1}, \mathtt{C2}, \mathtt{C3}, \mathtt{C4}, \mathtt{C5},$ $\mathtt{C6}\}, \{\mathtt{T1}, \mathtt{T2}, \mathtt{T3}\})$	$\mathtt{Closest} \in \{\{< \mathtt{C2}, \mathtt{T2} >\}\}$
(6)	List the cities to the north of C3	$(\mathtt{X}, \mathtt{Y}) = \mathtt{City}, \mathtt{C3} = (\mathtt{U}, \mathtt{V}), \mathtt{Y} \in [\mathtt{U}, 60]$	$(\mathtt{X}, \mathtt{Y}) = \mathtt{C2}, (\mathtt{U}, \mathtt{V}) = (40, 28) \vee$ $(\mathtt{X}, \mathtt{Y}) = \mathtt{C5}, (\mathtt{U}, \mathtt{V}) = (40, 28) \vee$ $(\mathtt{X}, \mathtt{Y}) = \mathtt{C6}, (\mathtt{U}, \mathtt{V}) = (40, 28)$
(7)	Report bridges which are not in a city	$\mathtt{Intersection} = \mathtt{Road} \cap \mathtt{River}$ $\mathtt{Intersection} \notin \{\mathtt{C1}, \mathtt{C2}, \mathtt{C3}, \mathtt{C4}, \mathtt{C5}, \mathtt{C6}\}$	$\mathtt{Intersection} = (59, 43) \vee$ $\mathtt{Intersection} = (56, 31) \vee$ $\mathtt{Intersection} = (46, 27)$

3 Constraints on Sets of Spatial Objects

Now we formally present our spatial constraint system. Let `NVar` be a set of numerical variables $x, y, \ldots$, `OVar` a set of variables for objects $A, B, \ldots$, `PVar` a set of variables for pairs of objects $P, Q, \ldots$, and $\mathbb{R}$ the set of real numbers. Now, we have a data constructor for pairs of coordinates $(\_,\_)$: $\texttt{NType} \times \texttt{NType} \rightarrow \texttt{Point}$, where $\texttt{NType} = \texttt{NVar} \cup \mathbb{R}$, a set of data constructors for spatial data types: $\emptyset :\rightarrow \texttt{SDT}$, $\texttt{line} : [\texttt{Point}] \rightarrow \texttt{Line}$, $\texttt{polygon} : [\texttt{Point}] \rightarrow \texttt{Polygon}$, and $\texttt{region} : [\texttt{Point}] \rightarrow \texttt{Region}$ where $\texttt{SDT} = \texttt{OVar} \cup \texttt{Point} \cup \texttt{Line} \cup \texttt{Polygon} \cup \texttt{Region}$. We have also a data constructor for pairs of *spatial data types*: $<\_,\_>: \texttt{SDT} \times \texttt{SDT} \rightarrow \texttt{PairSDT}$, and $\texttt{PSDT} = \texttt{PVar} \cup \texttt{PairSDT}$. With respect to sets of spatial objects and pairs of spatial objects, we have a set of data constructors for sets of (pairs of) spatial objects: $\emptyset :\rightarrow \texttt{SetSDT}$, $\{\_|\_\} : \texttt{SDT} \times \texttt{SetSDT} \rightarrow \texttt{SetSDT}$, $\emptyset :\rightarrow \texttt{SetPSDT}$, and $\{\{\_|\_\}\} : \texttt{PSDT} \times \texttt{SetPSDT} \rightarrow \texttt{SetPSDT}$. Finally, we have data constructors for a special kind of spatial data types, called `Window`, `Circle` such that $\texttt{window} : \texttt{Point} \times \texttt{Point} \rightarrow \texttt{Window}$, $\texttt{circle} : \texttt{Point} \times \texttt{NType} \rightarrow \texttt{Circle}$ and $\texttt{Range} = \texttt{Window} \cup \texttt{Circle}$. In addition, `Interval` builds finite closed real intervals: $[\_,\_] : \texttt{NType} \times \texttt{NType} \rightarrow \texttt{Interval}$.

Now we present a set of *operations* over the defined types. Some operations are *partial*, that is, are not defined for some arguments. Assuming the symbol $\perp$ representing a special value called *undefined*, we can consider the following types. For the type `NType`, we have the binary operations $=, \neq$: $\texttt{NType} \times \texttt{NType} \rightarrow \texttt{Boolean}$, and $\in, \notin$: $\texttt{NType} \times \texttt{Interval} \rightarrow \texttt{Boolean}$. The formal semantics is as follows. Given a valuation μ of numerical variables into $\mathbb{R} \cup \{\perp\}$, then we denote by $\mu(\texttt{n})$, where $\texttt{n} \in \texttt{NType}$, the value of $\texttt{n}$ under the valuation μ as the real number (or undefined), defined as $\mu(n) = \mu(x)$ if $n \equiv x$ is a variable and $\mu(n) = n$, otherwise. Now, given $n_1, n_2, n_3 \in \texttt{NType}$, and a valuation μ, we define $\perp = \perp, \perp \neq \perp$ are both false; $n_1 = n_2$ iff $\mu(n_1) = \mu(n_2)$; $n_1 \neq n_2$ iff $\mu(n_1) \neq \mu(n_2)$; $n_1 \in [n_2, n_3]$ iff $\mu(n_2) \leq \mu(n_1) \leq \mu(n_3)$; and finally, $n_1 \notin [n_2, n_3]$ iff $\mu(n_2) > \mu(n_1)$ or $\mu(n_1) > \mu(n_3)$. $\perp$ represents undefined and therefore an incomparable value. We denote by $Val(\texttt{NType})$ the set of valuations of numerical variables into $\mathbb{R} \cup \{\perp\}$. Now, the value of a spatial object O under a valuation $\mu \in Val(\texttt{NType})$, denoted by $\mu(O)$, is a *closed point-based geometric figure* ($\mathcal{FIG}$), and it is defined as follows:

- $\mu((p_1, p_2)) =_{def} \{(\mu(p_1), \mu(p_2))\}$ if none of the $\mu(p_i)$ are $\perp$; and $\emptyset$, otherwise.
- $\mu(line[(p_1, p_2), (p_3, p_4), \ldots, (p_{n-1}, p_n)]) =_{def} \{(r, s) \mid r = \alpha \times \mu(p_{2k+1}) + (1 - \alpha) \times \mu(p_{2k+3}), s = \alpha \times \mu(p_{2k+2}) + (1 - \alpha) \times \mu(p_{2k+4}), \alpha \in [0, 1], 0 \leq k \leq n - 4/2, k \in \mathbb{N}, \alpha \in \mathbb{R}\}$, if none of the $\mu(p_i)$ is $\perp$; and $\emptyset$, otherwise, where $n \geq 4$, $n \in \mathbb{N}$, and each $p_i \in \texttt{NType}$. Analogously for polygons and regions.

Spatial objects can be grouped in both sets of objects and sets of pairs of objects. The operations on sets of (pairs of) spatial objects are interpreted into *closed point-based geometric figures with holes* ($\mathcal{FH}$).

Definition 1 (Figures with Holes). *Given $\mathcal{FIG}$'s: F, H_i, a figure with holes $\mathcal{G}$ has the form $F - \cup_{k \geq i \geq 1} H_i$ where $k \geq 0$, $H_i \cap H_j = \emptyset$ if $i \neq j$ and $\cup_{k \geq i \geq 1} H_i \subseteq F$. The set $\cup_{k \geq i \geq 1} H_i$ is called the holes of $\mathcal{G}$. The set of figures*

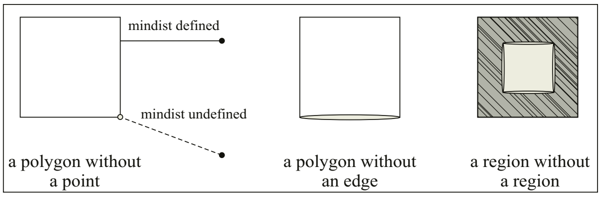

mindist defined
mindist undefined
a polygon without a point
a polygon without an edge
a region without a region

Fig. 3. Closed Point-Based Geometric Figures with Holes

with holes is denoted by $\mathcal{FH}$. We denote by (i) $\bar{\mathcal{G}}$ the figure obtained from $\mathcal{G}$, adding the topological frontier; (ii) $mindist(\mathcal{G}, \mathcal{G}')$ (resp. $maxdist(\mathcal{G}, \mathcal{G}')$) denotes the minimum (resp. maximum) distance from $\mathcal{G}$ to $\mathcal{G}'$, defined as follows: $mindist(\mathcal{G}, \mathcal{G}') =_{def} \perp$ if $min\{d((p,q),(r,s)) \mid (p,q) \in \mathcal{G}, (r,s) \in \mathcal{G}'\} < min\{d((p,q),(r,s)) \mid (p,q) \in \bar{\mathcal{G}}, (r,s) \in \bar{\mathcal{G}'}\}$ and otherwise $mindist(\mathcal{G}, \mathcal{G}') =_{def} min\{d((p,q),(r,s)) \mid (p,q) \in \mathcal{G}, (r,s) \in \mathcal{G}'\}$ and analogously for maxdist; (iii) $\mathcal{L}(\mathcal{G})$ denotes the length of a line $\mathcal{G}$; (iv) $\mathcal{P}(\mathcal{G})$ denotes the perimeter of a polygon or region $\mathcal{G}$; (v) $\mathcal{A}(\mathcal{G})$ denotes the area of a region $\mathcal{G}$. The three are defined as $\perp$ whenever the figure has a hole. (vi) $\mathcal{PS}(\mathcal{G})$ (resp. $\mathcal{LS}(\mathcal{G})$) denotes the set of points (resp. line segments of the frontier) which define $\mathcal{G}$.

An element of $\mathcal{FH}$ can have holes, which are in the topological frontier of the object (see figure 3). When the distance (minimum or maximum) from an element of $\mathcal{FH}$ to another element of $\mathcal{FH}$ is computed, it could be that the nearest points are in the holes. In this case, we consider both distances are undefined (see figure 3). The elements of $\mathcal{FH}$ can be grouped into sets, denoted by $\mathcal{SFH}$, and sets of pairs of elements of $\mathcal{FH}$, denoted by $\mathcal{SPFH}$. Grouping $\mathcal{FH}$ into sets we can handle more complex spatial objects like graphs, among others. Sets of (pairs of) spatial objects can have variables representing spatial objects. Given a valuation Δ of variables of spatial objects into $\mathcal{FH}$ then: $\Delta(\mu(\{O_1, \ldots, O_n\})) =_{def} \{\Delta(\mu(O_1)), \ldots, \Delta(\mu(O_n))\}$, and $\Delta(\mu(\{\{< O_1, O_2 >, \ldots, < O_{n-1}, O_n >\}\})) =_{def} \{(\Delta(\mu(O_1)), \Delta(\mu(O_2))), \ldots, (\Delta(\mu(O_{n-1})), \Delta(\mu(O_n)))\}$ where each $O_i \in$ `SDT`. We can also consider valuations of variables representing pairs of spatial objects into pairs of $\mathcal{FH}$'s. We denote by Val(`SDT`) the set of valuations into $\mathcal{FH}$'s and Val(`PSDT`) the set of valuations into pairs of $\mathcal{FH}$'s.

With respect to set theory operations, we have the set: $\cup, \cap, -$: `SDT` $\times$ `SDT` $\rightarrow$ `SetSDT`, `SetSDT` $\times$`SetSDT` $\rightarrow$ `SetSDT`, the semantics is the usual on set theory, that is, given valuations $\mu \in Val$(`NType`), $\Delta \in Val$ (`SDT`), and $\Omega \in Val$(`PSDT`); $O_i \in$ `SDT` and $S_i \in$ `SetSDT`; the set operations are defined as $[\![O_1 \Theta O_2]\!]_{(\mu,\Delta,\Omega)} = [\![O_1]\!]_{(\mu,\Delta,\Omega)} \Theta [\![O_2]\!]_{(\mu,\Delta,\Omega)}$ and $[\![S_1 \Theta\ S_2]\!]_{(\mu,\Delta,\Omega)} = [\![S_1]\!]_{(\mu,\Delta,\Omega)} \Theta\ [\![S_2]\!]_{(\mu,\Delta,\Omega)}$, where $\Theta \in \{\cup, \cap, -\}$.

With respect to equality and membership operations we have the following set: $=, \neq$: `SDT` $\times$ `SDT` $\rightarrow$ `Boolean`, `PSDT` $\times$ `PSDT` $\rightarrow$ `Boolean`, and $\in, \notin$: `Point` $\times$ `SDT` $\rightarrow$ `Boolean`, `SDT` $\times$ `SetSDT` $\rightarrow$ `Boolean`, `PSDT` $\times$ `SetPSDT` $\rightarrow$ `Boolean`. The above operations are interpreted in terms of the set of points that the (set of) spatial objects define. For instance, $[\![O_1 = O_2]\!]_{(\mu,\Delta,\Omega)} =_{def}$ *true* if $\Delta\mu(O_1) = \Delta\mu(O_2)$ and $[\![(p_1, p_2) \in O]\!]_{(\mu,\Delta,\Omega)} =_{def}$ *true* if $\mu((p_1, p_2)) \in \Delta\mu(O)$. Similarly with the other equality and membership operations.

With respect to metric operations, we have the following set: $\texttt{mindist}, \texttt{maxdist} : \texttt{SDT} \times \texttt{SDT} \rightarrow \mathbb{R}, \texttt{SetSDT} \times \texttt{SetSDT} \rightarrow \mathbb{R}$, $\texttt{pmindist}, \texttt{pmaxdist} : \texttt{SDT} \times \texttt{SDT} \rightarrow \texttt{PSDT}, \texttt{SetSDT} \times \texttt{SetSDT} \rightarrow \texttt{PSDT}$, and $\texttt{omindist}, \texttt{omaxdist} : \texttt{SetSDT} \times \texttt{SetSDT} \rightarrow \texttt{PSDT}$. Operations mindist (resp. maxdist) are undefined whenever the minimum (resp. maximum) distance of the $\mathcal{FH}$'s they represent is undefined. For instance, $[\![mindist(O_1, O_2)]\!]_{(\mu,\Delta,\Omega)} =_{def} min\{d((p_1, p_2), (q_1, q_2)) \mid (p_1, p_2) \in [\![O_1]\!]_{(\mu,\Delta,\Omega)},\ (q_1, q_2) \in [\![O_2]\!]_{(\mu,\Delta,\Omega)}\}$ if $mindist([\![O_1]\!]_{(\mu,\Delta,\Omega)},\ [\![O_2]\!]_{(\mu,\Delta,\Omega)}) \neq \perp$, and $\perp$, otherwise; and $[\![mindist(S_1, S_2)]\!]_{(\mu,\Delta,\Omega)} =_{def} min\{mindist(\mathcal{G}_1, \mathcal{G}_2) \mid \mathcal{G}_1 \in [\![S_1]\!]_{(\mu,\Delta,\Omega)},\ \mathcal{G}_2 \in [\![S_2]\!]_{(\mu,\Delta,\Omega)}, mindist(\mathcal{G}_1, \mathcal{G}_2) \neq \perp\}$, where $O_i \in \texttt{SDT}$ and $S_i \in \texttt{SetSDT}$. Similarly with the other metric operations.

With respect to the topological ones, we have the following set: $\texttt{inside}$, $\texttt{outside}$, $\texttt{overlap}$, $\texttt{disjoint}$, $\texttt{meet}$, $\texttt{cuts}$, $\texttt{neighbor}$, $\texttt{mcovers}$, $\texttt{ncovers}$, $\texttt{mcoveredby}$, $\texttt{ncoveredby} : \texttt{SDT} \times \texttt{SDT} \rightarrow \texttt{Boolean}, \texttt{SetSDT} \times \texttt{SetSDT} \rightarrow \texttt{SetPSDT}$, and $\texttt{rangeq} : \texttt{SetSDT} \times \texttt{Range} \rightarrow \texttt{SetSDT}$. For instance, $[\![O_1\ inside\ O_2]\!]_{(\mu,\Delta,\Omega)} =_{def} true$ if $[\![O_1]\!]_{(\mu,\Delta,\Omega)} \subseteq [\![O_2]\!]_{(\mu,\Delta,\Omega)}$; and $[\![S_1\ inside\ S_2]\!]_{(\mu,\Delta,\Omega)} =_{def} \{(\mathcal{G}_1, \mathcal{G}_2) \mid \mathcal{G}_1 \in [\![S_1]\!]_{(\mu,\Delta,\Omega)},\ \mathcal{G}_2 \in [\![S_2]\!]_{(\mu,\Delta,\Omega)}$ and $\mathcal{G}_1 \subseteq \mathcal{G}_2\}$, where $O_i \in \texttt{SDT}$ and $S_i \in \texttt{SetSDT}$. Similarly with the other topological operations.

Finally, we have structural operations: $\texttt{area} : \texttt{Region} \rightarrow \mathbb{R}, \texttt{length} : \texttt{Line} \rightarrow \mathbb{R}, \texttt{perimeter} : \texttt{Polygon} \cup \texttt{Region} \rightarrow \mathbb{R}, \texttt{area}, \texttt{length}, \texttt{perimeter} : \texttt{SetSDT} \rightarrow \mathbb{R}$, and $\texttt{setpoints}$, $\texttt{setsegments} : \texttt{SDT} \rightarrow \texttt{SetSDT}, \texttt{SetSDT} \rightarrow \texttt{SetSDT}$. For instance, $[\![area(S)]\!]_{(\mu,\Delta,\Omega)} =_{def} \sum_{\mathcal{G} \in [\![S]\!]_{(\mu,\Delta,\Omega)}, \mathcal{G}\ is\ a\ region} \mathcal{A}(\mathcal{G})$. Similarly with the other structural operations. A complete version of the semantics of the operations can be found in [1].

4 Spatial Constraint Satisfaction Problem

In this section we define what is a spatial constraint satisfaction problem, by presenting its general form, a special class which represents the solved forms, and a set of conditions in order to ensure that each spatial constraint satisfaction problem is equivalent to a solved form.

Definition 2 (Spatial CSP).
A spatial constraint satisfaction problem (SCSP) Γ is a conjunction of typed boolean operations over the types NType*,* SDT*,* PSDT*,* SetSDT*,* SetPSDT*,* Range *and* Interval *of the form $\Gamma \equiv \varphi_1, \ldots, \varphi_n$.*

Definition 3 (Solution). *A triple (μ, Δ, Ω), where $\mu \in Val(\texttt{NType})$, $\Delta \in Val(\texttt{SDT})$ and $\Omega \in Val(\texttt{PSDT})$ is a solution of a SCSP $\Gamma \equiv \varphi_1, \ldots, \varphi_n$ if each $[\![\varphi_i]\!]_{(\mu,\Delta,\Omega)}$ is equal to true. A SCSP is called* satisfiable *whenever has at least a solution.*

Definition 4 (Solved SCSP). *A solved SCSP Π is a disjunction of spatial constraint satisfaction problems of the form $\Pi \equiv \bigvee_{i \geq 1} \Gamma_i$ where each Γ_i is of the form: $\Gamma_i \equiv \varphi_1, \ldots, \varphi_n$, and each φ_j is a solved constraint of the form:*

1. *$x = n$ where $x \in \texttt{NVar}$ and $n \in \mathbb{R} \cup Dom(\Gamma_i)$*
2. *$x \in [n_1, n_2] - \cup_{k \geq j \geq 1}[h_j, h_{j+1}]$ where $x \in \texttt{NVar}$, n_1, n_2, $h_j, h_{j+1} \in \mathbb{R} \cup Dom(\Gamma_i)$ and $k \geq 0$; in addition, $\Gamma_i(n_1) \leq \Gamma_i(h_j) \leq \Gamma_i(h_{j+1}) \leq \Gamma_i(n_2)$, for all $k \geq j \geq 1$, and $[\Gamma_i(h_j), \Gamma_i(h_{j+1})] \cap [\Gamma_i(h_l), \Gamma_i(h_{l+1})] = \emptyset$ if $j \neq l$.*

3. $(x, y) \in O$, *where* $x, y \in \mathtt{NVar}$ *and* $O \in \mathcal{FH} \cup Dom(\Gamma_i)$
4. $A = O - \cup_{k \geq j \geq 1} H_j$ *where* $A \in \mathtt{OVar}$, $O, H_j \in \mathcal{FH} \cup Dom(\Gamma_i)$ *and* $k \geq 0$*; in addition,* $\cup_{k \geq j \geq 1} \Gamma_i(H_j) \subseteq \Gamma_i(O)$, *and* $\Gamma_i(H_j) \cap \Gamma_i(H_l) = \emptyset$ *if* $j \neq l$.
5. $A \in S$ *where* $A \in \mathtt{OVar}$ *and* $\Gamma_i(S) \in \mathcal{SFH}$.
6. $P = Q$ *where* $P \in \mathtt{PVar}$, *where* $Q \in Dom(\Gamma_i)$ *or* $Q \equiv < Q_1, Q_2 >$, $Q_1, Q_2 \in \mathcal{FH} \cup Dom(\Gamma_i)$
7. $P \in SP$ *where* $P \in \mathtt{PVar}$ *and* $\Gamma_i(SP) \in \mathcal{SPFH}$.

where there exists at most one solved constraint for each variable x, A *and* P *in each* Γ_i. *Variables* x, A *and* P *in the definition represent the domain of* Γ_i, *denoted by* $Dom(\Gamma_i)$. $\Gamma_i(x)$ *(resp.* $\Gamma_i(A)$, $\Gamma_i(P)$*) denote the set of solutions represented by the solved constraints in which occurs* x *(resp.* A *and* P*), and* $\Gamma_i(S)$, $\Gamma_i(SP)$ *the set of* $\mathcal{SFH}$*'s, respectively* $\mathcal{SPFH}$*'s obtained from them.*

Definition 5 (Solutions of a Solved SCSP).
A solved SCSP $\Pi \equiv \bigvee_{i \geq 1} \Gamma_i$ *defines a set of solutions, denoted by* $Sol(\Pi)$, *and defined as* $Sol(\Pi) = \cup_{\Gamma_i} Sol(\Gamma_i)$, *where each* Γ_i *defines a set of solutions which consists on triples* (μ, Δ, Ω), *recursively defined as follows:*

$$\star \quad \begin{aligned} \mu =_{def} & \cup_{x = n \in \Gamma_i} \{x/\mu(n)\} \\ & \cup_{(x \in [n_1, n_2] - \cup_{k \geq j \geq 1} [h_j, h_{j+1}]) \in \Gamma_i, r \in [\mu(n_1), \mu(n_2)], r \notin \cup_{k \geq j \geq 1} [\mu(h_j), \mu(h_{j+1})]} \{x/r\} \\ & \cup_{((x,y) \in O) \in \Gamma_i, (p,q) \in \Delta\mu(O)} \{x/p, y/q\} \end{aligned}$$

$$\star \quad \begin{aligned} \Delta =_{def} & \cup_{(A = O - \cup_{k \geq j \geq 1} H_j) \in \Gamma_i} \{A/\Delta\mu(O) - \cup_{k \geq j \geq 1} \Delta\mu(H_j)\} \\ & \cup_{(A \in S) \in \Gamma_i, \mathcal{G} \in \Delta\mu(S)} \{A/\mathcal{G}\} \end{aligned}$$

$$\star \quad \Omega =_{def} \cup_{P = Q \in \Gamma_i} \{P/\Omega\Delta\mu(Q)\} \cup_{(P \in SP) \in \Gamma_i, (\mathcal{G}_1, \mathcal{G}_2) \in \Omega\Delta\mu(SP)} \{P/(\mathcal{G}_1, \mathcal{G}_2)\}$$

Finally, $\Gamma_i(x) =_{def} \mu(x)$, $\Gamma_i(A) =_{def} \Delta(A)$ *and* $\Gamma_i(P) =_{def} \Omega(P)$, $\Gamma_i(n) =_{def} n$ *if* $n \in \mathbb{R}$, $\Gamma_i(O) =_{def} O$ *if* $O \in \mathcal{FH}$, $\Gamma_i(P) =_{def} P$ *if* $P = (\mathcal{G}_1, \mathcal{G}_2)$ *and* $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{FH}$ *for each* Γ_i.

In order to ensure that any spatial constraint satisfaction problem is equivalent to a solved SCSP we have to require that the SCSP follows some syntactic conditions. We call *safe SCSP* to SCSP ensuring these conditions.

Definition 6 (Range Restricted Variables). *We say that a variable* $\alpha \in \mathtt{NVar} \cup \mathtt{OVar} \cup \mathtt{PVar}$ *occurring on a SCSP* Γ *is range restricted if there exist (i) an equality constraint* $\alpha = E$ *or (ii) membership constraint* $\alpha \in E$, *such that* E *is ground (without variables) or all the variables of* E *are range restricted.*

We call that a variable is *strongly range restricted* considering only the case (i) in the above definition. Finally, we require that distance-based and structural operations are *rigid.*

Definition 7 (Rigid Constraints). *A spatial constraint is rigid if whenever an operation involving* $\Theta \in \{\mathtt{mindist}, \mathtt{maxdist}, \mathtt{pmindist}, \mathtt{pmaxdist}, \mathtt{omindist}, \mathtt{omaxdist}, \mathtt{area}, \mathtt{length}, \mathtt{perimeter}\}$ *is included in the form* $\Theta(O_1, O_2)$ *(or* $\Theta(O)$*) (resp.* $\Theta(S_1, S_2)$ *(or* $\Theta(S)$*)) then* O_1 *and* O_2 *(or* O*) (resp.* S_1 *and* S_2 *(or* S*)) only contain strongly range restricted variables.*

Definition 8 (Safe SCSP). *A safe SCSP is a SCSP where all the variables are range restricted, and all the spatial constraints are rigid.*

Theorem 1. *Each satisfiable safe SCSP is equivalent to a solved SCSP.*

5 Constraint Solver

In this section we present the basis of the constraint solver. With this aim we propose a set of transformation rules for obtaining a solved CSP from every SCSP.

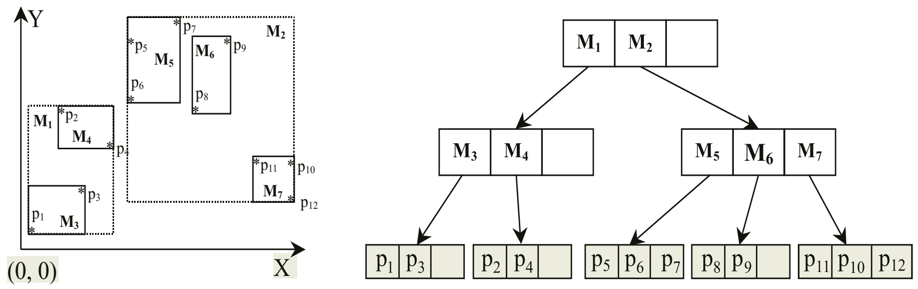


Fig. 4. MBRs and R-trees

5.1 MBRs and R-Trees

The basis of the efficiency of our constraint solver is the use of *Minimum Bounded Rectangles* (MBR's) to enclose objects and set of objects. In the case of set of objects the MBR's are organized in data structures called *R-trees*. These spatial access methods are a similar structure to the well-known data structures *B-trees* used for file indexing. In the case of R-trees, each object is enclosed in its MBR and stored in the *leaves*. Each *internal node* of the tree stores the set of MBR's enclosing its children and has as *searching key* the MBR covering the children.

This structure allows optimizations in the form of branch and bound algorithms which takes into account the coordinates of the MBR's (upper-left corner and lower-right corner of the MBR). As an example of this data structure, the figure 4 shows how to store in an R-tree the points $\mathtt{p}_1, \ldots, \mathtt{p}_{12}$ in which the MBR $\mathtt{M}_1$ encloses the points $\mathtt{p}_1, \ldots, \mathtt{p}_4$ and $\mathtt{M}_2$ the rest of points. At the same time $\mathtt{M}_1$ is subdivided into $\mathtt{M}_3$ and $\mathtt{M}_4$, and so on. It gives us the R-tree described in the right chart of the figure. For more details about R-trees see [8].

We adopt the cited structure but adapting MBR's and R-trees to the context of constrained objects in the following sense. Each spatial object is enclosed into an MBR but the MBR is also constrained given that the spatial object can be described by means of constrained coordinates. That is, a spatial object can be $\mathtt{line}[(\mathtt{x}, \mathtt{8}), (\mathtt{10}, \mathtt{12})]$, and $\mathtt{x}$ can be constrained to belong to an interval (note that due to safe condition, each coordinate must be constrained by an equality or a membership to an interval). It forces to consider constrained MBRs which have the upper-left corner and lower-right corner also constrained. For instance in the case $\mathtt{line}[(\mathtt{x}, \mathtt{8}), (\mathtt{10}, \mathtt{12})]$, and supposing $\mathtt{x} \in [\mathtt{7}, \mathtt{22}]$ it is enclosed by two MBRs with corners $(\mathtt{x}, \mathtt{12})$ and $(\mathtt{10}, \mathtt{8})$ if $\mathtt{x} \in [\mathtt{7}, \mathtt{10}]$, and $(\mathtt{10}, \mathtt{12})$ and $(\mathtt{x}, \mathtt{8})$ whenever $\mathtt{x} \in [\mathtt{10}, \mathtt{22}]$ following the criteria for building MBRs. Now, the building of the R-tree enclosing both MBRs follows the usual criteria.

5.2 Transformation Rules

Our constraint solver will be described by means of a set of transformation rules of the form $\Pi \bigvee CH \odot \Gamma \hookrightarrow \Pi \bigvee CH' \odot \Gamma'$. Given a SCSP Γ is intended to apply these rules on $CH \odot \Gamma \backslash CH$. The set CH is a conjunction of simple constraints, called *consistence constraints*, which should be checked for *consistency* in each step of the transformation process. Initially, each CH contains the set of constraints on `NType` of Γ. The transformation process may generate disjunctions of constraints whenever there are several alternatives. The process ends when no more rules can be applied (i.e. the constraint system only contains *constraints on candidate objects*). In order to obtain a solved SCSP we have to add a *refinement step* which consists on apply the operations defined on the *candidate objects*. The performance of our constraint solver should be tested in the presence of a *massive quantity of constraints*.

5.3 Consistence Constraints

The basis of the optimization of our constraint solver is the consistence checking of a conjunction of simple constraints. The consistence constraints have the following form:

- $\mathtt{n_1} = \mathtt{n_2}$, $\mathtt{n_1} \neq \mathtt{n_2}$, $(\mathtt{p_1}, \mathtt{p_2}) = (\mathtt{p'_1}, \mathtt{p'_2})$, $(\mathtt{p_1}, \mathtt{p_2}) \neq (\mathtt{p'_1}, \mathtt{p'_2})$, $\mathtt{n_1} \in [\mathtt{n_2}, \mathtt{n_3}]$, $\mathtt{n_1} \notin [\mathtt{n_2}, \mathtt{n_3}]$, which are constraints on `NType`;
- $(\mathtt{p_1}, \mathtt{p_2}) \in \mathtt{R}$ and $(\mathtt{p_1}, \mathtt{p_2}) \notin \mathtt{R}$ where $\mathtt{p_1}, \mathtt{p_2} \in \mathtt{NType}$, and `R` is an MBR; which are also equivalent to real interval constraints once $(\mathtt{p_1}, \mathtt{p_2}) \in \mathtt{R}$ is equivalent to $\mathtt{p_1} \in [\mathtt{R.up.x}, \mathtt{R.low.x}] \wedge \mathtt{p_2} \in [\mathtt{R.up.y}, \mathtt{R.low.y}]$, and analogously for $(\mathtt{p_1}, \mathtt{p_2}) \notin \mathtt{R}$; where the suffixes `up` and `low` denote the upper-left corner and lower-right corner of the MBR `R`, respectively, and the suffixes `x` and `y` denote the coordinates of such corners in the axis X and Y, respectively;
- `mbrbound(S, R)` where `S` is from `SetSDT`, and `R` is an MBR; and `distbound(PS,` $\mathtt{n_1}$, $\mathtt{n_2}$`)` where $\mathtt{n_1}$ and $\mathtt{n_2}$ are from `NType` and `PS` is from `SetPSDT`. The first one requires the elements of the set `S` to be included in `R`; and the second one requires the distance for each pair of objects of `PS` is bounded in $[\mathtt{n_1}, \mathtt{n_2}]$.

Assuming a constraint solver (enabled for consistency checking) for real interval constraints, we should provide a consistence checker for the two last kind of constraints. It should follows the next rules of *constraint propagation*:

$$\begin{array}{ll} (\mathbf{P1}) & \Pi \bigvee mbrbound(S, R_1) \wedge mbrbound(S, R_2) \odot \Gamma \hookrightarrow \\ & \Pi \bigvee mbrbound(S, R_3) \wedge R_3 = intermbr(R_1, R_2) \odot \Gamma \\ (\mathbf{P2}) & \Pi \bigvee distbound(PS, n_1, n_2) \wedge distbound(PS, n_3, n_4) \odot \Gamma \hookrightarrow \\ & \Pi \bigvee distbound(PS, n_5, n_6) \wedge n_5 = max(n_1, n_3) \wedge n_6 = min(n_2, n_4) \odot \Gamma \end{array}$$

where $\mathtt{intermbr}(\mathtt{R_1}, \mathtt{R_2})$ denotes the intersection of two MBRs which is trivially an MBR, and $\mathtt{max}(\mathtt{n_1}, \mathtt{n_2})$ (resp. $\mathtt{min}(\mathtt{n_3}, \mathtt{n_4})$) denotes the maximum (respectively the minimum) of two real numbers. In addition, the constraint solver should be able to propagate membership constraints to these special constraints in order to ensure consistence, and it should follow the next rules:

$(\mathbf{P3})\ \Pi \bigvee mbrbound(S,R) \odot O \in S \wedge \Gamma \hookrightarrow \Pi \bigvee mbrbound(S,R) \wedge O \in R \odot O \in S \wedge \Gamma$
$(\mathbf{P4})\ \Pi \bigvee distbound(PS,n_1,n_2) \odot <O_1,O_2> \in PS \wedge \Gamma \hookrightarrow$
$\quad \Pi \bigvee distbound(PS,n_1,n_2) \wedge mindmbr(O_1.mbr,O_2.mbr) = n_3 \wedge$
$\quad n_3 \in [n_1,n_2] \wedge maxdmbr(O_1.mbr,O_2.mbr) = n_4 \wedge n_4 \in [n_1,n_2] \odot <O_1,O_2> \in PS \wedge \Gamma$

where `mindmbr`(R_1,R_2) (resp. `maxdmbr`(R_1,R_2)) denotes the minimum (resp. maximum) distance of two MBR's, and the suffix `O.mbr` denotes the MBR enclosing the object `O`. In summary, the consistence checker of these simple constraints should always check interval constraints. The *failure rule* for the consistence checker is as follows:

$$(\mathbf{FAILURE})\ \Pi \bigvee CH \odot \Gamma \hookrightarrow \Pi \quad if\ CH\ is\ inconsistent$$

5.4 Transformation Rules

In this subsection we will review the transformation rules, showing the main cases. First of all, we will summarize the notation used in the rules:

– For objects, we use the suffixes `mbr`, `up` and `low` with the meaning of the previous section, and using the suffix `obj` to refer to the object itself.
– For sets of objects, the suffix `root` denotes the root of the R-tree storing the set of objects, and we use indices $\mathtt{i}_1,\ldots,\mathtt{i}_n$ for denoting the child of index $\mathtt{i}_j$ in a internal node of an R-tree.
– For set of pairs of objects, the suffixes `first` and `second` refer to the set of objects in the first (resp. second) component of each pair.
– For MBRs, we use functions `unionmbr`, `intermbr` and `diffmbr` for computing the same operations on MBRs.
– We introduce *a new kind of constraints* for each operation obtaining a *set of pairs of objects*, of the form: `mindtree`($\mathtt{m},\mathtt{PS},\mathtt{R}_1,\mathtt{R}_2$), `maxdtree`($\mathtt{m},\mathtt{PS},\mathtt{R}_1,\mathtt{R}_2$), `insidetree`($\mathtt{PS},\mathtt{R}_1,\mathtt{R}_2$), ..., etc, where `PS` is from `SetPSDT`, and $\mathtt{R}_1$ and $\mathtt{R}_2$ are MBRs. The meaning of the new kind of constraints is a bound (in the form of MBRs) for the search space for each pair of `PS`.
– Finally, we introduce *a new kind of constraints* for each operation obtaining a *set of objects*, of the form: `rangetree(S,R,Win)` and `rangetree(S,R,Circle)`, and so on, where `R` is an MBR, `S` is from `SetSDT`. The meaning of such constraints is a bound in the form of an MBR for the search space of `S`.

The transformation rules are shown in tables 1, 2 and 3. Due to the limit of space we have included the main cases of the tranformation rules, the full version can be found in [1]. With respect to the transformation rules of **equality constraints** (table 1), they use the suffixes `up` and `low` to refer to the corners of the MBRs enclosing the object. As an example, the rule **(E1)** introduces *consistence constraints* for comparing the two MBRs of the compared objects. In addition, it uses the suffix `obj` to refer to the object itself (in this case the compared objects are trivially candidates).

With respect to the **membership constraints** (table 1), basically they introduce consistence constraints for handling the MBRs (resp. R-trees) enclosing an object (resp. a set of objects) (rule **(M1)** (resp. **(M2)** to **(M4)**)). The most interesting rules of this block are the rules from **(M2)** to **(M4)**. They handle

Table 1. Equality and Membership Transformation Rules

(E1)	$\Pi \bigvee CH \odot (O_1 = O_2 \wedge \Gamma) \hookrightarrow \Pi \bigvee (O_1.up = O_2.up \wedge O_1.low = O_2.low \wedge CH) \odot (O_1.obj = O_2.obj \wedge \Gamma)$
(M1)	$\Pi \bigvee CH \odot ((p_1, p_2) \in O \wedge \Gamma) \hookrightarrow \Pi \bigvee ((p_1, p_2) \in O.mbr \wedge CH) \odot ((p_1, p_2) \in O.obj \wedge \Gamma)$
(M2)	$\Pi \bigvee CH \odot (O \in S \wedge \Gamma) \hookrightarrow \Pi \bigvee CH \odot (O \in S.root \wedge \Gamma)$
(M3)	$\Pi \bigvee CH \odot (O \in R \wedge \Gamma) \hookrightarrow \Pi \bigvee_{j=s_1,\ldots,s_k} ((O.up \in R \wedge O.low \in R \wedge CH) \odot (O \in R.j \wedge \Gamma))$ $if\ R\ has\ subtrees\ s_1, \ldots, s_k$
(M4)	$\Pi \bigvee CH \odot (O \in R \wedge \Gamma) \hookrightarrow \Pi \bigvee (O.up \in R \wedge O.low \in R \wedge CH) \odot (O.obj \in R.obj \wedge \Gamma)$ $if\ R\ is\ a\ leaf$

the R-tree enclosing a set of objects. For instance, the rule **(M2)** starts the search in the tree root, and the rule **(M3)** discards the children which do not contain the MBR enclosing the searched object. It should be checked by means of the **(FAILURE)** rule. Finally, rule **(M4)** adds constraints for the candidate objects. As an example, we can consider w.r.t the figure 4:

$$\begin{array}{l}
\emptyset \odot p_5 \in \{p_1, \ldots, p_{12}\} \hookrightarrow \emptyset \odot p_5 \in \{p_1, \ldots, p_{12}\}.root \hookrightarrow \\
p_5 \in \{p_1, \ldots, p_{12}\}.root.up \wedge p_5 \in \{p_1, \ldots, p_{12}\}.root.low \odot p_5 \in \{p_1, \ldots, p_{12}\}.root.M_1 \bigvee \\
p_5 \in \{p_1, \ldots, p_{12}\}.root.up \wedge p_5 \in \{p_1, \ldots, p_{12}\}.root.low \odot p_5 \in \{p_1, \ldots, p_{12}\}.root.M_2 \hookrightarrow \ldots \\
\emptyset \odot p_5 \in \{p_1, \ldots, p_{12}\}.root.M_2.M_5 \hookrightarrow \ldots
\end{array}$$

With respect to the transformation rules for **set constraints** (table 2), they use the cited operations for MBRs (as an example see rules **(SE1)** and **(SE2)**).

With respect to **metric constraints** (table 2), they add consistence constraints of the form $\mathtt{m} \in [\mathtt{a}, \mathtt{b}]$ and $\mathtt{distbound}(\mathtt{PS}, \mathtt{a}, \mathtt{b})$, for each minimum and maximum distance to be computed. `m` is a variable used for computing the minimum (resp. maximum) distance of two objects (or set of objects) and `PS` is a variable for storing pairs of objects at the minimum (resp. maximum) distance. In the refinement step the bounds `a` and `b` should be updated for candidate objects. These constraints represent the *lower* and *upper bounds* of the distance of two objects and sets of pairs of points or objects, respectively. In such a way, they are used for the pruning of the search for the minimum distance (and pairs of points or objects at the minimum distance). Similarly for maximum distances. In addition, the use of the new kind of constraints `mindtree`, `pmindtree`, etc, allows the handling of each R-tree, and enables the decomposition of an MBR into its children. For instance, rules **(ME1)** and **(ME2)** compute these bounds for a couple of objects, and the rule **(ME3)** starts the search in the tree root, and the rules from **(ME4)** to **(ME6)** update the lower bounds, for the case of minimum distance of two sets of objects. Finally, **(ME5)** obtains the candidate objects. The rules for pairs of points and objects at the minimum distance are similar. The case of maximum distance is also similar, updating the upper bound.

With respect to the **topological constraints** (table 3), we will show the case of `inside`, and the rest of cases are similar. The technique for solving such constraints is based on the use of the consistence constraint `mbrbound(S, R)`, which keeps the bound in the form of an MBR for the elements of `S`. In this

Table 2. Set and Metric Transformation Rules

(SE1)	$\Pi \bigvee CH \odot \Gamma[O_1 \cup O_2] \hookrightarrow \Pi \bigvee mbrbound(S, unionmbr(O_1.mbr, O_2.mbr)) \wedge CH \odot \Gamma[S] \wedge S = O_1.obj \cup O_2.obj$
(SE2)	$\Pi \bigvee CH \odot \Gamma[S_1 \cup S_2] \hookrightarrow \Pi \bigvee mbrbound(S, unionmbr(S_1.root, S_2.root)) \wedge CH \odot \Gamma[S] \wedge S = S_1.root \cup S_2.root$
(ME1)	$\Pi \bigvee CH \odot \Gamma[mindist(O_1, O_2)] \hookrightarrow \Pi \bigvee m \in [mindmbr(O_1.mbr, O_2.mbr), maxdmbr(O_1.mbr, O_2.mbr)] \wedge CH \odot \Gamma[m] \wedge m = mindist(O_1.obj, O_2.obj)$
(ME2)	$\Pi \bigvee CH \odot \Gamma[pmindist(O_1, O_2)] \hookrightarrow \Pi \bigvee distbound(PS, mindmbr(O_1.mbr, O_2.mbr), maxdmbr(O_1.mbr, O_2.mbr)) \wedge CH \odot \Gamma[PS] \wedge PS = pmindist(O_1.obj, O_2.obj)$
(ME3)	$\Pi \bigvee CH \odot \Gamma[mindist(S_1, S_2)] \hookrightarrow \Pi \bigvee m \in [mindmbr(S_1.root, S_2.root), maxdmbr(S_1.root, S_2.root)] \wedge CH \odot (\Gamma[m] \wedge mindtree(m, PS, S_1.root, S_2.root))$
(ME4)	$\Pi \bigvee m \in [a, b] \wedge CH \odot mindtree(m, PS, R_1, R_2) \wedge \Gamma \hookrightarrow \Pi \bigvee m \in [mindmbr(R_1, R_2), b] \wedge CH \odot \wedge_{j=i_1,\ldots,i_n, k=l_1,\ldots,l_t} mindtree(m, PS, R_1.j, R_2.k) \wedge \Gamma$ $if\ R_1\ has\ subtrees\ i_j\ and\ R_2\ has\ subtrees\ l_k\ and\ a >= mindmbr(R_1, R_2)$
(ME5)	$\Pi \bigvee m \in [a, b] \wedge CH \odot mindtree(m, PS, R_1, R_2) \wedge \Gamma \hookrightarrow \Pi \bigvee m \in [mindmbr(R_1, R_2), b] \wedge CH \odot m = mindist(PS) \wedge PS = \{\{< R_1.obj, R_2.obj >\}\} \cup PS_1 \wedge \Gamma[PS/PS_1]$ $if\ R_1\ and\ R_2\ are\ leaves\ and\ a >= mindmbr(R_1, R_2)\ and\ PS\ occurs\ in\ \Gamma$
(ME6)	$\Pi \bigvee m \in [a, b] \wedge CH \odot mindtree(m, PS, R_1, R_2) \wedge \Gamma \hookrightarrow \Pi \bigvee m \in [a, b] \wedge CH \odot \Gamma$ $if\ a < mindmbr(R_1, R_2)$

case, a new constraint `insidetree` allows the handling of the R-tree (rules **(T1)** to **(T5)**). Finally, the **structural constraints** (table 3) take into account the *lower* and *upper bounds* of area, length and perimeter operations of an object and a set of objects, using also MBRs and R-trees (as an example see rule **(S1)**).

5.5 Refinement Step

The refinement step consists of the solving of the constraints over candidate objects of Γ. For solving these constraints now we should take into account the candidate objects (i.e. figures with holes) stored in each MBR computed with the transformation rules. This solving together the solving of the interval constraints of CH represent the solved CSP of the original one.

5.6 Soundness and Completeness

With regard to soundness and completeness we would like to reason about a crucial point in any set of transformation rules: *the progress in the transformation.* With respect to this point, the transformation rules are defined in such a way that each constraint, occurring in Γ, is handled by at least one rule, it ensures *local progress*, once the applicability conditions are only refereed to the structure of the constraint. In addition, the metric transformation rules have as applicability conditions, conditions of the form $\mathtt{a} < \mathtt{mindmbr}(\mathtt{R}_1, \mathtt{R}_2)$, etc. Although $\mathtt{R}_1, \mathtt{R}_2$ can be defined by means of coordinates which can be variables,

Table 3. Topological and Structural Transformation Rules

(T1) $\mathit{\Pi} \bigvee CH \odot \mathit{\Gamma}[O_1\ inside\ O_2] \hookrightarrow$
$\mathit{\Pi} \bigvee O_1.up \in O_2.mbr \wedge O_1.low \in O_2.mbr \wedge CH \odot \mathit{\Gamma}[O_1.obj\ inside\ O_2.obj]$

(T2) $\mathit{\Pi} \bigvee CH \odot \mathit{\Gamma}[S_1\ inside\ S_2] \hookrightarrow \mathit{\Pi} \bigvee (mbrbound(PS.first, S_1.root) \wedge$
$mbrbound(PS.second, S_2.root) \wedge CH) \odot (\mathit{\Gamma}[PS] \wedge insidetree(PS, S_1.root, S_2.root))$

(T3) $\mathit{\Pi} \bigvee mbrbound(PS.first, R_1) \wedge mbrbound(PS.second, R_2) \wedge CH \odot insidetree(PS, R_1, R_2)$
$\wedge \mathit{\Gamma} \hookrightarrow \mathit{\Pi} \bigvee \wedge_{j=i_1,\ldots,i_n,k=l_1,\ldots,l_t} mrbbound(PS.first, R_1.j) \wedge mbrbound(PS.second, R_2.k)$
$\wedge CH \odot \wedge_{j=i_1,\ldots,i_n,k=l_1,\ldots,l_t} insidetree(PS, R_1.j, R_2.k) \wedge \mathit{\Gamma})$
$if\ R_1\ has\ subtrees\ i_1, \ldots, i_n\ and\ R_2\ has\ subtrees\ l_1, \ldots, l_t, and\ R_1, R_2\ intersect$

(T4) $\mathit{\Pi} \bigvee CH \odot insidetree(PS, R_1, R_2) \wedge \mathit{\Gamma} \hookrightarrow$
$\mathit{\Pi} \bigvee CH \odot PS = R_1.obj\ inside\ R_2.obj \cup PS_1 \wedge \mathit{\Gamma}[PS/PS_1]$
$if\ R_1\ and\ R_2\ are\ leaves\ and\ R_1, R_2\ intersect$

(T5) $\mathit{\Pi} \bigvee CH \odot insidetree(PS, R_1, R_2) \wedge \mathit{\Gamma} \hookrightarrow \mathit{\Pi} \bigvee CH \odot \mathit{\Gamma}$
$if\ R_1, R_2\ do\ not\ intersect\ and\ PS\ occurs\ in\ \mathit{\Gamma}$

(S1) $\mathit{\Pi} \bigvee CH \odot \mathit{\Gamma}[area(O)] \hookrightarrow$
$\mathit{\Pi} \bigvee m \in [0, (O.up.x - O.low.x) * (O.up.y - O.low.y)] \wedge CH \odot \mathit{\Gamma}[m] \wedge m = area(O.obj)$

however the safety condition ensures that `R1` and `R2` are fully defined given that metric operations are rigid. A proof of soundness and completeness has been omitted due to the lack of space.

6 Conclusions and Future Work

In this paper we have developed the basis for a constraint solver based on branch and bound techniques for efficiently solve metric and topological queries on sets of spatial objects. As future work we plan to implement a program library for the handling of R-trees and constraints. In addition, we would like to study how to integrate into the well-known framework of *Constraint Logic Programming (CLP)* [9].

References

1. J. M. Almendros-Jiménez and Antonio Corral. Solving constraints on sets of spatial objects, available in `http://www.ual.es/~jalmen/padl05tr.ps`. Technical report, Dpto. de Lenguajes y Computación, Universidad de Almería, 2004.
2. K. R. Apt. *Principles of Constraint Programming.* Cambridge U. Press, 2003.
3. A. Belussi, E. Bertino, and B. Catania. Manipulating Spatial Data in Constraint Databases. In *SSD'97*, LNCS 1262, pages 115–141. Springer, 1997.
4. A. Belussi, E. Bertino, and B. Catania. An Extended Algebra for Constraint Databases. *TKDE*, 10(5):686–705, 1998.
5. P. Codognet and D. Díaz. A Simple and Efficient Boolean Solver for Constraint Logic Programming. *Journal of Automated Reasoning*, 17(1):97–129, 1996.
6. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *ACM SIGMOD*, pages 189–200, 2000.

7. R. H. Güting. An Introduction to Spatial Database Systems. *VLDB*, 3(4):357–399, 1994.
8. A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, pages 47–57, 1984.
9. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *JLP*, 19,20:503–582, 1994.
10. J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond Finite Domains. In *CP'94*, pages 86–94, 1994.
11. G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.
12. K. Marriot and P. J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
13. D. Papadias, P. Kalnis, and N. Mamoulis. Hierarchical Constraint Satisfaction in Spatial Databases. In *AAAI/IAAI'99*, pages 142–147, 1999.
14. D. Papadias, Y. Theodoridis, T. K. Sellis, and M. J. Egenhofer. Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. In *ACM SIGMOD*, pages 92–103, 1995.
15. P. Z. Revesz. Safe Query Languages for Constraint Databases. *ACM TODS*, 23(1):58–99, 1998.
16. P. Rigaux, M. Scholl, L. Segoufin, and S. Grumbach. Building a Constraint-based Spatial Database System: Model, Languages, and Implementation. *Information Systems*, 28(6):563–595, 2003.
17. P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases: with application to GIS*. Morgan Kaufmann Publishers, 2001.
18. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *ACM SIGMOD*, pages 71–79, 1995.
19. E. Schwalb and L. Vila. Temporal Constraints: A Survey. *Constraints*, 3(2/3):129–149, 1998.

Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization

James Bailey and Peter J. Stuckey

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne 3010, Australia

Abstract. An unsatisfiable set of constraints is minimal if all its (strict) subsets are satisfiable. The task of type error diagnosis requires finding all minimal unsatisfiable subsets of a given set of constraints (representing an error), in order to generate the best explanation of the error. Similarly circuit error diagnosis requires finding all minimal unsatisfiable subsets in order to make minimal diagnoses. In this paper we present a new approach for efficiently determining all minimal unsatisfiable sets for any kind of constraints. Our approach makes use of the duality that exists between minimal unsatisfiable constraint sets and maximal satisfiable constraint sets. We show how to incrementally compute both these sets, using the fact that the complements of the maximal satisfiable constraint sets are the hitting sets of the minimal unsatisfiable constraint sets. We experimentally compare our technique to the best known method on a number of large type problems and show that considerable improvements in running time are obtained.

Keywords: Minimal unsatisfiable sets, constraint solving, hitting sets, hypergraph transversals.

1 Introduction

A set of constraints is unsatisfiable if it has no solution. An unsatisfiable set of constraints is minimal if all its (strict) subsets are satisfiable. A number of forms of error diagnosis, in particular type error diagnosis, require finding all minimal unsatisfiable subsets of a given set of constraints (representing an error), in order to generate the best explanation of the error.

There is a significant amount of work that deals with minimal unsatisfiable sets, particularly in the areas of explanation and intelligent backtracking (e.g. [4]) or nogood creation (e.g. [16]). However, the vast bulk of this work is only interested in finding a single minimal unsatisfiable set. This is usually achieved by relying on some kind of justification recording, and then postprocessing the recorded unsatisfiable set to eliminate unnecessary constraints. In many cases a non-minimal unsatisfiable set is used.

Our motivation for examining the problem of finding all minimal unsatisfiable subsets of a set of constraints arises from type error debugging. In Hindley-Milner type inference and checking, a program is mapped to a system of Herbrand constraints and a type error results when this system of Herbrand constraints is unsatisfiable. An explanation of the type error is given by a minimal unsatisfiable subset of the system of Herbrand constraints

Example 1. Consider the following fragment of Haskell code

```
f [] y = []
f (x:xs) y = if (x < y) then (f xs y) else xs
g xs y = 'z' > (f xs y)
```

that defines a function `f` which returns a list, and then erroneously compares the result of that function to character `'z'`. The Chameleon type debugging system [17, 11] finds a single minimal unsatisfiable set of constraints that causes the type error and underlines the associated the program fragments. If that minimal unsatisfiable set included the constraints posed by the base case in the definition of function `f`, the Chameleon system would show the following

```
f [] y = []
f (x:xs) y = if (x < y) (f xs y) else xs
g xs y = 'z' > (f xs y)
```

while if the minimal unsatisfiable set included the constraints posed by the recursive case in the definition of function `f`, the Chameleon system would instead show the following

```
f [] y = []
f (x : xs) y = if (x < y) (f xs y) else xs
g xs y = 'z' > (f xs y)
```

One explanation may be easier to understand than another, for example if it involves fewer constraints. Hence, deriving all minimal unsatisfiable sets allows us to choose the "simplest" explanation.

Finding all minimal unsatisfiable sets of a system of constraints is a challenging problem because, if it is done naively, it involves examining every possible subset. Indeed in the worst case there may be an exponential number of answers. Most previous work has concentrated on its use in diagnosis of circuit errors. The best method we know of is given by García de la Banda *et al* [5], who presented a series of techniques that significantly improved on earlier approaches of [13] and [12].

This paper presents a completely new method for calculation of all minimal unsatisfiable constraint sets. We show that this problem is closely related to a problem from the area of data mining, concerned with enumerating interesting sets of frequent patterns. In particular, we show how an algorithm known as *Dualize and Advance* [10], which has previously been proposed for discovering collections of maximal frequent patterns in data mining, can be efficiently adapted to the constraint context, to jointly enumerate both all minimal unsatisfiable sets and all maximal satisfiable sets of constraints.

Interestingly, Dualize and Advance, although having good worst case complexity, does not seem to be a practical algorithm for finding maximal frequent patterns in data mining [19, 8], due to the large number of patterns required to be output. However, in the constraint context, the size of the output (i.e. the number of minimal unsatisfiable sets of constraints and the number of maximal satisfiable sets of constraints) is typically far smaller and we demonstrate its efficiency. Furthermore, we show how improvements in the procedure can be made by incorporation of information from the constraint graph. We experimentally compare our method with the best known available technique from [5] on a number of debugging problems containing hundreds of constraints and show that our new approach can result in significant savings in running time. A further advantage of our approach for more traditional circuit diagnosis problems is that a possible diagnosis of an error corresponds to the complement of a maximal satisfiable set of constraints [15]. In our method these are easy to generate from the calculated maximal satisfiable sets.

The outline of the remainder of this paper is as follows. We first give some background definitions in Section 2. Next, we examine the best previous approach to the problem we are aware of, that of García de la Banda *et al* [5] in Section 3. In Section 4, we describe the Dualize and Advance approach and how it can be adapted and optimised for the constraint context. In Section 5 we describe the results of experiments comparing the two approaches. Finally in Section 6 we conclude and discuss future work.

2 Background

Let us start by introducing the notation which will be used herein. A constraint domain $\mathcal{D}$ defines the set of possible values of variables. A *valuation* θ, written $\{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\}$, $d_i \in \mathcal{D}, 1 \leq i \leq m$, maps each variable v_i to a value d_i in the domain.

A *constraint* c is a relation on a tuple of variables $vars(c)$. Let $vars(c) = (v_{i_1}, \ldots, v_{i_n})$ then c defines a subset $vals(c)$ of D^n. A valuation $\theta \equiv \{v_1 \mapsto d_1, \ldots, v_m \mapsto d_m\}$ is a *solution* of constraint c if $(d_{i_1}, \ldots, d_{i_n}) \in vals(c)$.

A set of constraints C is *satisfiable* iff there exists a solution θ of C. Otherwise it is *unsatisfiable*. We assume an algorithm $sat(C)$ which returns $true$ if C is satisfiable and $false$ otherwise.

We will also be interested in *incremental* satisfaction algorithms. Incremental satisfiability checks process each of the constraints one at a time. Hence, to answer the question $sat(\{c_1, \ldots, c_n\})$ we compute the answers to questions $sat(\{c_1\})$, $sat(\{c_1, c_2\})$, ..., $sat(\{c_1, \ldots, c_{n-1}\})$ and finally $sat(\{c_1, \ldots, c_n\})$. We describe incremental satisfiability algorithms as a procedure $isat(c_n, state)$ which takes a new constraint c_n and an internal state representing a set of constraints $\{c_1, \ldots, c_{n-1}\}$ and returns a pair $(result, state')$ where $result = sat(\{c_1, \ldots, c_n\})$ and $state'$ is a new internal state representing constraints $\{c_1, \ldots, c_n\}$.

Since we are focusing on debugging, we will use the Herbrand equation constraint domain $\mathcal{H}$. That is, equations over uninterpreted function symbols, such as the constraint arising in Hindley-Milner typing. The complexity of *sat* for this class of constraints is $O(n)$ where n is the number of symbols in the constraint [14]. The complexity of n calls to *isat*, $(true, s_1) = isat(c_1, true)$, $(true, s_2) = isat(c_2, s_1)$, ..., $(result, s_n) = isat(c_n, s_{n-1})$ is $O(nA^{-1}(n))$ where $A^{-1}(n)$ is the inverse Ackerman's function. The amortized incremental complexity of $isat(c, state)$ is thus effectively constant. We will use calls to *isat* as one measure for the complexity of our algorithms. For this purpose as call $sat(\{c_1, \ldots, c_n\})$ is equivalent to n calls to *isat*.

For a given problem, we define the constraint universe U as the set which contains every possible constraint that can be considered. In a typing problem these are all the type constraints represented by the program to be typed, while in circuit diagnosis it is all the constraints defining the circuit and its inputs and outputs.

A constraint set C is a *minimal unsatisfiable* constraint set if C is unsatisfiable and each $C' \subset C$ is satisfiable. A constraint set C is a *maximal satisfiable* constraint set if C is satisfiable and each $C' \supset C$ (where $C' \subseteq U$) is unsatisfiable. For a set of constraints S, we define its complement to be $\overline{S} = U - S$. Let $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ be a set of constraint sets (i.e. each of S_1, S_2 etc is a set of constraints). We define $\overline{\mathbf{S}}$, the complement of $\mathbf{S}$, to be the set of complements of each of the constraint sets. i.e. $\overline{\mathbf{S}} = \{\overline{S}_1, \overline{S}_2, \ldots, \overline{S}_k\}$.

Let $\mathbf{A} = \{A_1, A_2, \ldots, A_n\}$ be a set of constraint sets. We say a set $P \subseteq U$ is a *hitting set* of $\mathbf{A}$ if $(P \cap A_1 \neq \emptyset) \wedge (P \cap A_2 \neq \emptyset) \wedge \ldots \wedge (P \cap A_n \neq \emptyset)$. We say that P is a *minimal hitting set* of $\mathbf{A}$, if P is a hitting set of $\mathbf{A}$ and each $S \subset P$ is not a hitting set of $\mathbf{A}$. We define $\mathcal{HST}(\mathbf{A})$ to be the set of all the minimal hitting sets of $\mathbf{A}$. The cross product of two sets of set $\mathbf{A} = \{A_1, \ldots A_n\}$ and $\mathbf{B} = \{B_1, \ldots, B_m\}$ is denoted as $\mathbf{A} \otimes \mathbf{B} = \{A_i \cup B_j \mid i \leq n, j \leq m\}$.

3 Best Previous Approach

The best previous approach we are aware of for finding all minimal unsatisfiable subsets of a constraint set is from García de la Banda *et al* [5], who extended approaches by [13] and [12]. Essentially all these approaches rely on enumerating all possible subsets of the constraints and checking which are unsatisfiable but all of whose subsets are satisfiable.

The code for min_unsat shown below gives the core. The call min_unsat($\emptyset$,U,$\emptyset$) it finds all minimal unsatisfiable subsets of U. The first argument is used to avoid repeatedly examining the same subset. The call min_unsat(D, P, $\mathbf{A}$) traverses of all subsets of the set $D \cup P$ which include D, i.e. $\{D \cup P' \mid P' \subseteq P\}$. D refers to definite elements, ones which must appear in all subsequent subsets and P refers to possible elements, ones which may appear in subsequent subsets. The argument $\mathbf{A}$ collects all the minimal unsatisfiable subsets found so far. This call explores all subsets $\{D \cup P' \mid P' \subseteq P\}$ in an order such that all subsets of $D \cup P$ are explored before the **while** loop in the call min_unsat(D,P,$\mathbf{A}$) finishes. When a satisfiable subset is found then the algorithm need not look at its further subsets. If an unsatisfiable set is found then it is added to the collection $\mathbf{A}$ after all its subsets have been examined, unless there is already a subset of it in **A**. The code returns the set of minimal unsatisfiable subsets found.

```
min_unsat($D$, $P$, $\mathbf{A}$)
    if ($sat(D \cup P)$) return $\mathbf{A}$
    while ($\exists c \in P$)
        $P := P - \{c\}$
        $\mathbf{A} :=$ min_unsat($D$, $P$, $\mathbf{A}$)
        $D := D \cup \{c\}$
    endwhile
    if ($\neg \exists A \in \mathbf{A}$ such that $A \subset D$) $\mathbf{A} := \mathbf{A} \cup \{D\}$
    return $\mathbf{A}$
```

This simple approach is improved in [5] by (a) detecting constraints that are present in all minimal unsatisfiable subsets by preprocessing, (b) taking into account constraints that must always be satisfiable once other constraints are not present, (c) using reasoning about independence of constraints to reduce the number of subsets examined, and most importantly (d) using incremental constraint solving to select which elements c to select first in the **while** loop. The last modification is the most important in terms of reducing the amount of satisfaction checking and subsets examined.

Essentially by performing the satisfiability check $sat(D \cup P)$ incrementally we find the first constraint c_i where $D \cup \{c_1, \ldots, c_{i-1}\}$ is satisfiable and $D \cup \{c_1, \ldots, c_i\}$ is not. This guarantees that c_i appears in some minimal unsatisfiable set. By choosing $c = c_i$ in the while loop we (hopefully) quickly find large satisfiable subsets thus reducing the search.

4 Dualization Approach

We now describe our new approach for determining all the minimal unsatisfiable sets of constraints. It is based on a technique that has been proposed in the area of data mining, called Dualize and Advance [10], for discovery of interesting patterns in a database. Other similar algorithms exist from work in hypergraph transversals [3].

The key idea is that for a given constraint universe U, there exists a relationship between the minimal unsatisfiable sets of constraints and the maximal satisfiable sets of constraints. In particular, suppose we have a set $\mathbf{G}$ of satisfiable constraint sets, then $\mathcal{HST}(\overline{\mathbf{G}})$ is the collection of the smallest sets which are not contained in any set from $\mathbf{G}$. If we let $\mathbf{G}$ be the collection of all the maximal satisfiable constraint sets, then $\mathcal{HST}(\overline{\mathbf{G}})$ is the collection of all the smallest sets that are not contained in any maximal satisfiable set (i.e. the minimal unsatisfiable constraint sets). Furthermore, if $\mathbf{G}$ is a collection of some, but not all the maximal satisfiable constraint sets, then $\mathcal{HST}(\overline{\mathbf{G}})$ must contain at least one set which is satisfiable and is not contained in any set in $\mathbf{G}$ (see [10] for proof).

Example 1 *Suppose for universe $U = \{c_1, c_2, c_3, c_4\}$ the maximal satisfiable sets of constraints are $\mathbf{G} = \{\{c_3\}, \{c_4\}, \{c_2\}\}$. Then the complements sets are*

$\overline{\mathbf{G}} = \{\{c_1, c_2, c_3\}, \{c_1, c_3, c_4\}, \{c_1, c_2, c_4\}\}$ *and the minimal unsatisfiable sets are* $\mathcal{HST}(\overline{\mathbf{G}}) = \{\{c_1\}, \{c_2, c_4\}, \{c_2, c_3\}, \{c_3, c_4\}\}$.

We now present the algorithm for jointly generating both the minimal unsatisfiable sets and the maximal satisfiable sets. Although the stated purpose of our work is to find the minimal unsatisfiable sets, it is worth noting that the maximal satisfiable sets are also useful. In particular, if there are several possible error explanations, then the constraints which appear in the most maximal satisfiable sets are least likely to be in error. Similarly for circuit diagnosis, the minimal diagnoses are the complements of the maximal satisfiable sets [15].

The dualize and advance algorithm daa_min_unsat is given in Figure 1. The explanation is as follows. The algorithm maintains a number of variables: X is a satisfiable set, which is grown into a maximal satisfiable set M by the grow procedure which simply adds new constraints that do not cause unsatisfiability; $\mathbf{A}$ is the set of minimal unsatisfiable subsets currently found; $\mathbf{X}$ is the complements of the maximal satisfiable sets currently found; and $\mathbf{N}$ are the hitting sets for $\mathbf{X}$ which are the candidates for minimal unsatisfiable subsets.

Initially all the set of minimal unsatisfiable sets and complements of maximal satisfiable sets are empty. The X variable is set to $\emptyset$. In the **repeat** loop, the algorithm, repeatedly grows M to a maximal satisfiable subset, adds its complement to $\mathbf{X}$ and calculates the new hitting sets for $\mathbf{X}$. This gives the candidates $\mathbf{N}$ for minimal unsatisfiable subsets.

For each of these not already recognised as a minimal unsatisfiable subset we check satisfiability. If the set S is satisfiable then it is the starting point for a new maximal satisfiable set, and we break the **for** loop and continue. Otherwise S is added to the minimal unsatisfiable subsets $\mathbf{A}$. When we find no satisfiable S then we have discovered all minimal unsatisfiable subsets.

Example 2 *We trace the behaviour of the algorithm daa_min_unsat using Example 1, where the minimal unsatisfiable sets are* $\mathbf{G} = \{\{c_1\}, \{c_2, c_4\}, \{c_2, c_3\}, \{c_3, c_4\}\}$ *and the maximal satisfiable sets are* $\{\{c_4\}, \{c_3\}, \{c_2\}\}$. *We show the values of key variables just before the* ***for*** *loop for each iteration of the* ***repeat*** *loop.*

Iter.	M	$\mathbf{X}$	$\mathbf{N} = \mathcal{HST}(\mathbf{X})$
1	$\{c_2\}$	$\{\{c_1, c_3, c_4\}\}$	$\{\{c_1\}, \{c_3\}, \{c_4\}\}$
2	$\{c_3\}$	$\{\{c_1, c_3, c_4\}\}, \{c_1, c_2, c_4\}\}$	$\{\{c_4\}, \{c_1\}, \{c_2, c_3\}\}$
3	$\{c_4\}$	$\{\{c_1, c_3, c_4\}, \{c_1, c_2, c_4\}, \{c_1, c_2, c_3\}\}$	$\{\{c_1\}, \{c_2, c_3\}, \{c_2, c_4\}, \{c_3, c_4\}\}$

Each iteration produces a new complement of a maximum satisfiable set. Once all maximal satisfiable sets have been found, the **repeat** loop terminates.

4.1 Determining the Hitting Sets $\mathcal{HST}$

A core part of the procedure is the calculation of the hitting sets on each iteration of the while loop. There are many possible methods for computing hitting sets. This problem is also known as the *hypergraph transversal problem*. We use a

```
daa_min_unsat($U$)
    $\mathbf{A} := \emptyset$
    $\mathbf{X} := \emptyset$
    $X := \emptyset$
    repeat
        $M$ := grow($X$,$U$);
        $\mathbf{X} := \mathbf{X} \cup \{U - M\}$
        $\mathbf{N} := \mathcal{HST}(\mathbf{X})$
        $X := \emptyset$
        for $(S \in \mathbf{N} - \mathbf{A})$
            if $(sat(S))$
                $X := S$
                break
            else $\mathbf{A} := \mathbf{A} \cup \{S\}$
        endfor
    until $(X = \emptyset)$
    return $(\mathbf{A})$

grow($S$,$U$)
    for $(c \in U - S)$
        if $(sat(S \cup \{c\}))$ $S := S \cup \{c\}$
    endfor
    return($S$)
```

Fig. 1. Dualize and advance algorithm for finding minimal unsatisfiable sets.

method first described by Berge [2], since it is simple to implement and behaves reasonably efficiently. More complex techniques do exist though which have better worst case complexity (see [7]) or are better in practice for large problems (see [1]). The basic idea of the Berge algorithm is that to compute the hitting sets of a set G, the sets contained in G are first ordered, and then partial cross products of these sets are computed, with the output being minimised at each step.

Let $\mathbf{G} = \{S_1, S_2, \ldots, S_k\}$ and define $\mathbf{G}_i = \{S_1, \ldots, S_i\}$. Then $\mathcal{HST}(\mathbf{G}_i)$ is given by the formulas

$\mathcal{HST}(\mathbf{G}_1) = \{\{c\} \mid c \in S_1\}$
$\mathcal{HST}(\mathbf{G}_2) = \mathcal{HST}(\mathbf{G}_1 \cup \{S_2\}) = Min(\mathcal{HST}(\mathbf{G}_1) \otimes \{\{c\} \mid c \in S_2\})$
$\ldots$
$\mathcal{HST}(\mathbf{G}_i) = \mathcal{HST}(\mathbf{G}_{i-1} \cup \{S_i\}) = Min(\mathcal{HST}(\mathbf{G}_{i-1}) \otimes \{\{c\} \mid c \in S_i\})$

where $Min(\mathbf{G})$ is the set $\mathbf{G}$ with all non-minimal subsets removed.

$$Min(\mathbf{G}) = \{S \mid S \in \mathbf{G} \wedge (\forall T \in \mathbf{G}\ (T \subseteq S) \Rightarrow (T = S)\}.$$

4.2 Incremental Hitting Set Calculation

Looking more closely at the procedure for minimal hitting set calculation, we can see that it is incremental in nature – the hitting sets $\mathcal{HST}(\mathbf{G})$ of a set

G are computed by considering each set from **G** in turn and calculating the partial hitting sets. Thus, by remembering the partial hitting sets $\mathcal{HST}(\mathbf{G}_1)$, $\mathcal{HST}(\mathbf{G}_2)$, etc, we can incrementally calculate the new hitting sets of **X** when a new set of constraints is added. Hence we can replace the line

$\mathbf{N} := \mathcal{HST}(\mathbf{X})$

by

$\mathbf{N} := Min(\mathbf{N} \otimes \{\{c\} \mid c \in U - M\})$

4.3 Complexity of the Algorithm

The complexity of the algorithm is as follows. The number of iterations of the **repeat** loop is equal to the number of maximal satisfiable sets, since in each iteration we find one more M. In each iteration we call grow once which costs at most $|U|$ incremental calls to *sat*, for Herbrand equations the cost is thus $O(|U|)$ overall. Each minimal unsatisfiable set is found within the **for** loop of daa_min_unsat and needs no further processing once it has been found to be unsatisfiable. Each minimal unsatisfiable set requires at most $|U|$ incremental calls to *sat*, thus again $O(|U|)$ overall for Herbrand equations. If **A** is the collection of all the minimal unsatisfiable sets and **X** is the collection of all the complements of maximal satisfiable sets, then overall the complexity (using the optimisation from section 4.2) is $O(|\mathbf{A}| \times |U| + |\mathbf{X}| \times |U| + cost(\mathcal{HST}(\mathbf{X})))$.

The core part of the cost is the calculation of the hitting sets $\mathcal{HST}(\mathbf{X})$, done incrementally. In general, the number of hitting sets (and thus the size of **X**) can be exponential in $|U|$. Also, as we will show shortly (Example 3), the number of partial hitting sets may also be exponential, even when the size of **X** is polynomial. The addition of a new set S_i to **G**, can either increase the number of minimal hitting sets by a factor of $|S_i|$, or cause a superpolynomial decrease in the number of minimal hitting sets [18]. The exact complexity of the Berge algorithm is not yet well understood, but an upper bound for the $cost(\mathcal{HST}(\mathbf{X}))$ is $O(2^{|U|})$.

4.4 Optimisation of Hitting Set Computation Using the Constraint Graph

The order in which the sets of **G** are considered when computing the hitting sets can have a significant impact on the running time of the hitting set calculation. This is because the size of the partial hitting sets can blow up for certain orderings. An example (based on one from [6]) is:

Example 3 *Let* $\mathbf{G} = \{\{c_i, c_j\} \mid i \leq 10, j \leq 10\}$ *Suppose we order the sets of* **G** *to be* $\{\{c_1, c_2\}, \{c_3, c_4\}, \{c_5, c_6\}, \{c_7, c_8\}, \{c_9, c_{10}\}, \ldots\}\}$*. Then* $|\mathcal{HST}(\mathbf{G}_5)| = 2^5$*, whereas if they are ordered as* $\{\{c_1, c_2\}, \{c_2, c_3\}, \{c_1, c_3\}, \{c_1, c_4\}, \{c_2, c_4\}, \ldots\}$ *then* $|\mathcal{HST}(\mathbf{G}_5)| = 3$*. In other words, a blowup of the intermediate results occurs for the first ordering, but not the second.*

To address this problem, a natural heuristic to use is that the sets contained in $\mathbf{X}$ should be ordered in increasing cardinality, to minimise the number of partial hitting sets. However, we do not have direct information about the cardinality of the next M to be generated.

A heuristic to try and achieve this is as follows: When maximising a set in the grow procedure, we should add constraints in an order that 'maximises' the number of constraints in the final grown M set. Therefore, we should add the constraints most likely to cause unsatisfiability last of all. To identify such constraints, we use a constraint graph to help identify a global ordering of all the constraints in the universe, with constraints likely to cause unsatisfiability occurring at the end of the ordering and constraints not likely to cause unsatisfiability occurring at the start of the ordering.

Given a set of constraints U, the *constraint graph* $g(U)$ is a graph where each vertex in the graph corresponds to one of the constraints and there is an edge between two vertices c_1 and c_2 iff there exists a v such that $v \in vars(c_1)$ and $v \in vars(c_2)$. We estimate the centre of the constraint graph (the vertex will the least maximal distance from all other nodes) and then order vertices according to their distance from the centre. Constraints closest to the centre are at the beginning of the ordering, since these are expected to participate in the most minimal unsatisfiable subsets and those furthest away from the centre are at the end of the ordering.

4.5 Discussion of the Algorithm

As mentioned, our hitting set algorithm is based on the Dualize and Advance algorithm for mining interesting patterns (sets of items) in a database [10]. Work in [19] also presented a practical implementation of Dualize and advance for data mining. Our approach differs from both these works in a number of ways

– The context is constraints and not sets of items.
– The size of the output in data mining problems (number of maximally satisfiable sets and number of minimal unsatisfiable sets) is huge, this means that Dualize and Advance is not practical for data mining requirements [8, 19]. However, in the constraint scenario the number of minimal unsatisfiable sets is likely to be small, since the scenario is type error debugging. We are not aware of any previous work where Dualize and Advance has been shown to be efficient for an important practical problem.
– Knowledge of the constraint graph can be used as a means for improving the algorithm.

Dualize and Advance is quite similar to Reiter's approach for model based diagnosis ([15]), which uses the computation of hitting sets to relate conflict sets (similar to unsatisfiable sets) and diagnoses (complements of maximum satisfiable sets). The key difference is that Reiter's approach uses hitting set calculation to obtain each new minimum unsatisfiable set, whereas Dualize and Advance uses hitting set calculation to obtain each new maximum satisfiable set. Dualize and Advance has the important advantage that a satisfiable set can be grown into a maximum satisfiable set using an incremental solver. Reiter's technique requires a minimal unsatisfiable subset to be obtained from a larger unsatisfiable set by removal of constraints and there isn't the same opportunity for incremental solver use. Reiter's method also requires the costly maintenance of a tree structure for computing the hitting sets.

5 Experimental Evaluation

In order to investigate the benefits of our technique, we have implemented a prototype system in SICStus Prolog. The evaluation uses a number of benchmark problems arising from type error debugging. These are taken from sets of constraints generated by the Chameleon system [17] for debugging Haskell programs and use the efficient satisfiability procedure for solving Herbrand equations provided by SICStus Prolog. Figure 2 shows the characteristics of these benchmarks: the number of constraints $|U|$, the number of minimal unsatisfiable subsets $|\mathbf{A}|$ (and in brackets the average size of each minimal unsatisfiable subset) and the number of maximal satisfiable subsets $|\mathbf{X}|$ (and in brackets the average size). Note that $|\mathbf{X}|$ is the number of complements of the maximal satisfiable sets, which is equal to the number of maximal satisfiable sets. These benchmark problems were chosen due to their challenging size and each has a constraint universe of size 72 or more constraints. Naïvely each problem then requires considering at least 2^{72} subsets.

Benchmark	$\|U\|$	$\|\mathbf{A}\|$	$\|\mathbf{X}\|$
`const`	72	6 (37)	88 (70)
`rotate`	81	8 (65)	68 (80)
`filter`	98	12 (24)	5427 (95)
`drop`	156	11 (62)	599 (152)
`rot13`	159	9 (10)	376379 (154)
`permute`	239	16 (77)	120 (237)
`plot`	448	10 (15)	300 (445)
`diff`	610	4 (46)	46 (609)
`msort`	1016	4 (34)	2699 (1013)

Fig. 2. Type error benchmark problems.

The algorithms from Sections 3 and 4 were coded in SICStus Prolog. The experiments were run on a Dell PowerEdge 2500 with Intel PIII, 1GHz CPU and 2 GB memory. All times are measured in seconds. We compare against two versions of the algorithms from [5]. The first 3.648 (using the terminology of that paper) performs (a) preprocessing to detect constraints in all unsatisfiable subsets, (b) eliminates constraints which are always satisfiable after other constraints are deleted, (c) breaks constraints up that are independent and (d) uses the incremental search approach. The second 3.8 simply combines (a) and (d). The first version is the method that (in general) examines the fewest subsets, while the second is the one that (in general) performs the fewest *isat* checks.

Figure 3 shows the running times for four different algorithms. The first two sets of times are those from the system [5] with the parameter sets 3.648 and 3.8. The second two sets of running times are for the Dualize and Advance algorithm. DAA.1 uses the basic algorithm with the optimisations from 4.2. DAA.2 uses the basic algorithm with the optimisations from 4.2 and 4.4

Looking at Figure 3, we see that the hitting set algorithms are substantially faster in the majority of cases. The one exception is the `rot13` benchmark, where there are a huge number of maximally satisfiable sets. The structure of the `rot13` minimal unsatisfiable sets gives a clue as to why this may be so. There are nine of these, with several of them being quite different from one another (i.e. sharing few constraints). This may be because there is more than one independent type error in the `rot13` program. The `rot13` benchmark also illustrates the importance of independence optimizations for the approach of [5].

Benchmark	[5]3.648	[5]3.8	DAA.2	DAA.1
`const`	1.8	1.3	1	1
`rotate`	1.3	1.5	1	1
`filter`	16441	7505	381	644
`rot13`	1269	71089	27780	30807
`drop`	2152	3037	77	145
`permute`	296	57	14	7
`plot`	1932	1868	44	46
`diff`	387	11	14	15
`msort`	908543	91412	1430	1420

Fig. 3. Comparative running times to find all minimal unsatisfiable sets (Seconds).

The ordering of the constraints only makes a slight difference in many cases (DAA.1 versus DAA.2), but in others it can reduce computation time by half.

In Figure 4 we show the number of calls to *isat* made by each of the algorithms. We see that the number of calls to *isat* is substantially reduced for the DAA algorithms versus the others. Both versions of DAA make exactly the same number of *isat* checks. Observe that the only difference in running time between DAA.2 and DAA.1 is the time taken for (incremental) hitting set calculation.

Benchmark	[5]3.648	[5]3.8	DAA
`const`	7039	147945	6567
`rotate`	6934	81440	6028
`filter`	1956330	20692654	532208
`rot13`	86421	137706583	59844542
`drop`	118046	5154734	94041
`permute`	56464	106335	30119
`plot`	1649205	1649205	135200
`diff`	179685	180864	28856
`msort`	6890081	32858078	2743337

Fig. 4. Number of incremental satisfiability checks (calls to *isat*) to find all minimal unsatisfiable sets.

Looking at Figure 3, it is clear that for some of the problems, the amount of time taken, even for the best hitting set algorithm (DAA.2) may still be too high to be useful for interactive debugging. One strategy to cope with this is to provide the user with each minimal unsatisfiable set as soon as it is found, rather than waiting until all have been computed. This way the user may begin trying to discover the source of the error earlier. Figure 5 shows the amount of time taken for each algorithm to output the first

Benchmark	[5]3.648	[5]3.8	DAA.2
`const`	1.5	0.2	0.8
`rotate`	2.6	0.2	1.1
`filter`	5.4	0.7	0.2
`rot13`	22.8	2.5	0.4
`drop`	41	4.9	23.6
`permute`	60.5	4.3	12.1
`plot`	158	8.7	41.9
`diff`	276	6.6	13.5
`msort`	4889	146	36.7

Fig. 5. Comparative running times to find one minimal unsatisfiable set (Seconds).

minimal unsatisfiable set found. We see that the amount of time taken is generally more acceptable for interactive use and indeed the non hitting set algorithm [5]3.8 is usually the fastest. A possible technique would thus be to run [5]3.8 in parallel with DAA.2 and stop [5]3.8 after finding the first minimal unsatisfiable set. This would provide the user with one minimal unsatisfiable set quickly, but would still be likely compute the entire collection of minimal unsatisfiable in an acceptable total elapsed time.

6 Conclusions and Future Work

Finding all minimal unsatisfiable sets is a challenging problem because it implicitly involves considering each possible subset of a given set of constraints. In this paper we investigated how to reduce as much as possible the number of constraints sets that need to be examined. We presented a new method which builds upon related work in data mining and showed it to be superior to the best known previous method.

A promising direction for future work is to investigate the tradeoffs between using the hitting set approach and that of [5] and see if a hybrid technique combining the advantages of both can be developed.

Acknowledgements

We would like to thank Kathryn Francis for assistance with the implementation and experimental evaluation, and Jeremy Wazny for providing the type error benchmark problems.

References

1. J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 485–488, 2003.
2. C. Berge. *Hypergraphs, North Holland Mathematical Library*, volume 45. Elsevier Science Publishers B.V (North-Holland), 1989.
3. E. Boros, G. Gurvich, L. Khachiyan, and K. Makino. Dual bounded generating problems: Partial and multiple transversals of a hypergraph. *SIAM Journal on Computing*, 30(6):2036–2050, 2000.
4. B. Davey, N. Boland, and P.J. Stuckey. Efficient intelligent backtracking using linear programming. *INFORMS Journal of Computing*, 14(4):373–386, 2002.
5. Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, 2003.
6. T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.

7. Michael L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.
8. B. Goethals and M. Zaki. Advances in frequent itemset mining implementations: Introduction to FIMI03. In *[9]*.
9. Bart Goethals and Mohammed Javeed Zaki, editors. *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, FIMI'03*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.
10. D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.
11. C. Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Proc. of ESOP'03*, LNCS, pages 284–301. Springer-Verlag, 2003.
12. B. Han and S-J. Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(2):281–286, 1999.
13. A. Hou. A theory of measurement in diagnosis from first principles. *Artificial Intelligence*, 65:281–328, 1994.
14. M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
15. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.
16. J. Silva and K Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceeding of ICCAD'96*, pages 220–228, 1996.
17. M. Sulzmann and J. Wazny. Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.
18. K. Takata. On the Sequential Method for Listing Minimal Hitting Sets. In *Proceedings Workshop on Discrete Mathematics and Data Mining, 2nd SIAM International Conference on Data Mining*, 2002.
19. T. Uno and K. Satoh. Detailed description of an algorithm for enumeration of maximal frequent sets with irredundant dualization. In *[9]*.

Solving Collaborative Fuzzy Agents Problems with CLP($\mathcal{FD}$)[*]

Susana Munoz-Hernandez and Jose Manuel Gomez-Perez

School of Computer Science, Technical University of Madrid (UPM)
{susana,jgomez}@fi.upm.es

Abstract. Truth values associated to fuzzy variables can be represented in an ordeal of different flavors, such as real numbers, percentiles, intervals, unions of intervals, and continuous or discrete functions on different domains. Many of the most interesting fuzzy problems deal with a discrete range of truth values. In this work we represent these ranges using Constraint Logic Programming over Finite Domains (CLP($\mathcal{FD}$)). This allows to produce finite enumerations of constructive answers instead of complicated, hardly self-explanatory, constraints expressions. Another advantage of representing fuzzy models through finite domains is that some of the existing techniques and algorithms of the field of distributed constraint programming can be borrowed. In this paper we exploit these considerations in order to create a new generation of collaborative fuzzy agents in a distributed environment.

Keywords: Fuzzy Prolog, Modeling Uncertainty, (Constraint) Logic Programming, Constraint Programming Application, Finite Domains, Multi-Agent Systems, Collaborative Agents.

1 Introduction

The introduction of Fuzzy Logic into Logic Programming (LP) has resulted into the development of several "Fuzzy Prolog" systems. These systems replace the inference mechanism of Prolog with a fuzzy variant which is able to handle partial truth. Most of these systems implement the fuzzy resolution introduced in [11], examples being the Prolog-Elf system, Fril Prolog system and the F-Prolog language. However, there was no common method for fuzzifying Prolog as noted in [15]. Some of these Fuzzy Prolog systems only consider the fuzziness of predicates whereas other systems consider fuzzy facts or fuzzy rules. There is no agreement about which Fuzzy Logic must be used. Most of them use min-max logic (for modeling the conjunction and disjunction operations) but other systems just use Lukasiewicz logic.

There is also an extension of Constraint Logic Programming [1], which models logics based on semi ring structures. This framework can model the only semi ring structure that is the min-max Fuzzy Logic.

[*] This work has been partially funded by the European Union IST program under contract IST-2001-34717, Amos, and by Spanish MCYT projects TIC 2002-0055, CUBICO and TIC 2003 - 01036, SOFFIE.

Recently, a theoretical model for Fuzzy Logic Programming without negation, which deals with many-value implications, has been proposed by Votjas [18]. Through the last few years a large amount of work has been published by Medina et al. ([12]) about multi-adjoint programming, which describe a theoretical model, but no means of implementation.

In [14], truth values are interpreted as intervals but, more generally, in [4] a Fuzzy Prolog Language that models interval-valued Fuzzy Logic, implemented using CLP($\mathcal{R}$) [8], was presented. This Fuzzy Prolog system uses on the one hand the original inference mechanism of Prolog, and on the other hand constraint facilities and operations provided by CLP($\mathcal{R}$) to represent and handle the concept of partial truth.

In this approach a truth value will be a finite union of sub-intervals on $[0, 1]$. An interval is a particular case of union of one element, and a unique truth value is a particular case of having an interval with only one element. In this Fuzzy Prolog a truth value will be propagated through the rules by means of an *aggregation operator*. The definition of *aggregation operator* is general in the sense that it subsumes conjunctive operators (triangular norms like min, prod, etc.), disjunctive operators (triangular co-norms, like max, sum, etc.), average operators (like arithmetic average, quasi-linear average, etc) and hybrid operators (combinations of the above operators.

In this paper we take as starting point the syntax and semantics of the continuous Fuzzy Prolog approach to develop a discrete system which handles a finite number of truth values. Our implementation is not based on CLP($\mathcal{R}$) [7] (as in [4]) but CLP($\mathcal{FD}$) [17] is used instead. As a direct consequence, Fuzzy Prolog derives into discrete Fuzzy Prolog in a very natural way, allowing to represent truth values discretely and therefore produce finite enumerations of constructive answers. CLP($\mathcal{FD}$)techniques like *propagation* and *labeling* can be applied to improve efficiency in discrete fuzzy reasoning. Another advantage of a CLP($\mathcal{FD}$)−based implementation is that existing algorithms from the field of distributed constraint programming [19] can be adopted to design and build collaborative fuzzy agents systems to solve complex, inherently distributed fuzzy problems. We have developed this work using the Ciao Prolog system [5], taking advantage of its modular design and some of its extensions (constraints over real numbers and finite domains, distributed execution, modular code expansion facilities).

The rest of the paper is organized as follows. Section 2 summarizes the syntax and semantics of the Fuzzy Prolog system (presented in [4]) but restricted to discrete fuzzy functions. Section 3 provides an intuitive introduction to CLP($\mathcal{FD}$). Section 4 describes the collaborative agents system based in CLP($\mathcal{FD}$) that we have used, and provides some motivating examples. Finally, we conclude and discuss some future work (Section 6).

2 Fuzzy Prolog

Part of the future work described in [4] was the implementation of a continous Fuzzy Prolog. In [4] fuzzy functions are continuous. Representing fuzziness by means of continuous functions is very powerful and helps expressiveness but many real fuzzy problems are modeled using a finite set of values (although the result of a fuzzy function can be more than one only value, e.g. a union of intervals with a number of consecutive values). In this work we have provided an implementation of a discrete Fuzzy Prolog. Basically, we use similar syntax to that in [4] but the semantics and the mechanism of fuzzy resolution has changed. In [4] truth values were represented using constraints over real numbers. In this paper we represent truth values using finite domains, which support discreteness.

2.1 Truth Value

Given a relevant universal set X, any arbitrary fuzzy set A is defined by a function $A : X \rightarrow [0, 1]$, unlike the crisp set that would be defined by a function $A : X \rightarrow \{0, 1\}$. This definition of fuzzy set is by far the most extended in the literature as well as in the various successful applications of the fuzzy set theory. However, several more general definitions of fuzzy sets have also been proposed. The primary reason for generalizing ordinary fuzzy sets is that their membership functions are often overly precise. They require the assignment of a particular real number to each element of the universal set. However, for some concepts and contexts, we may only be able to identify approximately appropriate membership functions. An option is considering a membership function which does not assign to each element of the universal set one real number, but an interval of real numbers. Fuzzy sets defined by membership functions of this type are called *interval-valued fuzzy sets* [14]. These sets are defined formally by functions of the form $A : X \rightarrow \mathcal{E}([0, 1])$, where $\mathcal{E}([0, 1])$ denotes the family of all closed intervals of real numbers in $[0, 1]$.

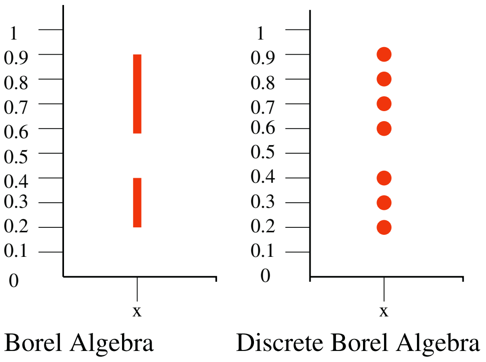


Fig. 1. Truth Value: Borel Algebra versus Discrete Borel Algebra

[4] proposes to generalize this definition, aiming to membership functions which assign one element of the Borel Algebra over the interval $[0, 1]$ to each element of the universal set. These sets are defined by functions of the form $A : X \rightarrow \mathcal{B}([0, 1])$, where an element in $\mathcal{B}([0, 1])$ is a countable union of sub-intervals of $[0, 1]$. In the present work, as continuous functions are no longer used, ranges of discrete (consecutive) values are handled instead of intervals.

Definition 1 (discrete-interval). *A* discrete-interval $[X_1, X_N]_d$ *is a set of a finite number of values,* $\{X_1, X_2, ..., X_{N-1}, X_N\}$*, between* X_1 *and* X_N*,* $0 \leq X_1 \leq X_N \leq 1$*, such that* $\exists\ 0 < \epsilon < 1.\ X_i = X_{i-1} + \epsilon,\ \ i \in \{2..N\}$.

Therefore, we use functions of the $A : X \to \mathcal{E}_d([0,1])$ form or $A : X \to \mathcal{B}_d([0,1])$ where we define $\mathcal{E}_d$ as the algebra that handles intervals of discrete values in $[0,1]$ and $\mathcal{B}_d$ as the algebra that handles union of intervals of discrete values in $[0,1]$. For example the truth value of x in $\mathcal{B}([0,1])$ is $[0.2,0.4] \cup [0.6,0.9]$ (that includes two continuous intervals) and a truth value of x in $\mathcal{B}_d([0,1])$ is $[0.2,0.4]_d \cup [0.6,0.9]_d$ (that is equivalent to $\{0.2,0.3,0.4\} \cup \{0.6,0.7,0.8,0.9\}$, i.e. $\{0.2,0.3,0.4,0.6,0.7,0.8,0.9\}$. See Figure 1.

2.2 Aggregation Operators

The truth value of a goal will depend on the truth value of the subgoals which are in the body of the clauses of its definition. In [4] *aggregation operators* are used to propagate the truth value by means of fuzzy rules. Fuzzy sets *aggregation* is done using the application of a numeric operator of the form $f : [0,1]^n \to [0,1]$. If it verifies $f(0,\ldots,0) = 0$ and $f(1,\ldots,1) = 1$, and in addition it is monotonic and continuous, then it is called *aggregation operator*. In this work we use monotonic but not continuous aggregation operators.

If we deal with the second definition of fuzzy sets it is necessary to generalize from *aggregation operator* of numbers to *aggregation operator* of intervals. Following the theorem proved by Nguyen and Walker in [14] to extend T-norms and T-conorms to intervals, [4] proposed a definition of operator for union of intervals (*union-aggregation*) where operators are continuous functions. In the presentation of the theory of possibility [21], Zadeh considers that fuzzy sets act as an elastic constraint on the values of a variable and fuzzy inference as constraint propagation.

In [4], truth values and the result of aggregations are represented by constraints. A constraint is a *Σ-formula* where Σ is a signature that contains the real numbers, the binary function symbols $+$ and $*$, and the binary predicate symbols $=$, $<$ and $\leq$. If the constraint c has a solution in the domain of real numbers in the interval $[0,1]$ then we say c is *consistent*, and we denote it as $solvable(c)$.

In this work we provide some new definitions to face the discrete case:

Definition 2 (discrete-aggregation). *Fuzzy sets* discrete-aggregation *is the application of a numeric operator of the form* $f : [0,1]^n \to [0,1]$. *If it verifies* $f(0,\ldots,0) = 0$ *and* $f(1,\ldots,1) = 1$, *and in addition it is monotonic.*

Notice the operator is not continuous.

Definition 3 (discrete-interval-aggregation). *Given a discrete-aggregation* $f : [0,1]^n \to [0,1]$, *a discrete-interval-aggregation* $F : \mathcal{E}_d([0,1])^n \to \mathcal{E}_d([0,1])$ *is defined as follows:*

$$F([x_1^l, x_1^u]_d, ..., [x_n^l, x_n^u]_d) = [f(x_1^l, ..., x_n^l), f(x_1^u, ..., x_n^u)]_d$$

Definition 4 (discrete-union-aggregation). *Given a discrete-interval-aggregation* $F : \mathcal{E}_d([0,1])^n \to \mathcal{E}_d([0,1])$ *defined over discrete-intervals, a discrete-union-aggregation* $\mathcal{F} : \mathcal{B}_d([0,1])^n \to \mathcal{B}_d([0,1])$ *is defined over union of discrete-intervals as follows:* $\mathcal{F}(B_1, \ldots, B_n) = \cup\{F(\mathcal{E}_{d,1}, ..., \mathcal{E}_{d,n}) \mid \mathcal{E}_{d,i} \in B_i\}$

2.3 Fuzzy Language

The alphabet of our language consists of the following kinds of classical symbols: *variables*, *constants*, *function symbols* and *predicate symbols*. A *term* is defined inductively as follows:

1. A *variable* is a *term*.
2. A *constant* is a *term*.
3. if f is an n-ary *function symbol* and $t_1, \ldots, t_n$ are *terms*, then $f(t_1, \ldots, t_n)$ is a *term*.

If p is an n-ary *predicate symbol*, and $t_1, \ldots, t_n$ are *terms*, then $p(t_1, \ldots, t_n)$ is an *atomic formula* or simply an *atom*. A *fuzzy program* is a finite set of *fuzzy facts*, and *fuzzy clauses* and we obtain information from the program through *fuzzy queries*. They are defined below:

Definition 5 (fuzzy fact). *If A is an atom, $A \leftarrow v$ is a* fuzzy fact*, where v, a truth value, is an element in $\mathcal{B}_d([0,1])$.*

Definition 6 (fuzzy clause). *Let $A, B_1, \ldots, B_n$ be atoms, $A \leftarrow_F B_1, \ldots, B_n$ is a* fuzzy clause *where F is a discrete-interval-aggregation operator of truth values in $\mathcal{B}_d([0,1])$, where F induces a discrete-union-aggregation as by definition 4.*

Definition 7 (fuzzy query). *A* fuzzy query *is a tuple $v \leftarrow A$? where A is an atom, and v is a variable (possibly instantiated) that represents a truth value in $\mathcal{B}_d([0,1])$.*

We represent the truth value v by means of constraints. For example, we use expressions as: $(v \geq 0.4 \;\wedge\; v \leq 0.7) \;\vee\; (v = 0.9)$ to represent a truth value in $[0.4, 0.7]_d \bigcup [0.9]_d$ (i.e. the truth value belongs to the set $\{0.4, 0.5, 0.6, 0.7, 0.9\}$).

Notice that in the above example we work with $\epsilon = 0.1$ but we can work with the precision we decide. For example if we would work with more precision we could represent the above example as $(v \geq 0.40 \;\wedge\; v \leq 0.70) \;\vee\; (v = 0.90)$ to represent a truth value in $[0.40, 0.70] \bigcup [0.90]$ (i.e. the truth value belong to this set $\{0.40, 0.41, 0.42, ..., 0.69, 0.70, 0.90\}$) where $\epsilon = 0.01$. In our implementation we consider that the precision is the minimum unit of decimals in which we represent the discrete (i.e., $[0.4, 0.7]_d$ with $\epsilon = 0.1$, $[0.40, 0.70]_d$ with $\epsilon = 0.01$, $[0.425, 0.778]_d$ with $\epsilon = 0.001$, etc.).

3 Introduction to CLP($\mathcal{FD}$)

Constraint Logic PSrogramming is an extension of Logic Programming, usually (but not necessarily) taking the Prolog language as base, which augments LP semantics with constraint (e.g., equation) handling capabilities, including the ability to generate constraints dynamically (e.g., at run time) to represent problem conditions and also to solve them by means of internal, well-tuned, user-transparent constraint solvers. Constraints can come in very different flavors, depending on the *constraint system* supported by the language. Examples of well-known constraint systems are linear [dis]equations, either over $\mathcal{R}$ or over

$\mathcal{Q}$ [7], $\mathcal{H}$ (equations over the Herbrand domain, finite trees), $\mathcal{FD}$ ([dis]equations over variables which range over finite sets with a complete order among their elements, usually represented as integers [17]).

$\mathcal{FD}$ is one of the more widely used constraint domains, since the finiteness of the domain of the variables allows, in the worst case, a complete traversal of each variable range when searching for a solution. This gives complete freedom to the type of equations an $\mathcal{FD}$ system can handle[1].

```
main(X,Y,Z) :-
    [X, Y, Z] in 1..5,
    X - Y .=. 2*Z,
    X + Y .>=. Z,
    labeling([X,Y,Z]).
```

Fig. 2. CLP($\mathcal{FD}$) program

Figure 2 shows a toy CLP($\mathcal{FD}$) program, with the same overall structure of other larger CLP($\mathcal{FD}$) programs. Briefly, the declarative reading of the program is that it succeeds for all values of X, Y and Z such that

$$X, Y, Z \in \mathbb{N} \ \wedge \ 1 \leq X, Y, Z \leq 5 \ \wedge \ X - Y = 2 * Z \ \wedge \ X + Y \geq Z$$

and fails if no values satisfies all of these constraints. Operationally, and also from the viewpoint of a programmer, the program first declares initial ranges for variables `X`, `Y`, and `Z`[2], then a set of relationships are set up among them, and finally a search procedure is called to bind the variables to *definite* values. The first phase (setting up equations) fires a process called *propagation*, in which some values can be removed from the domain of the variables (e.g., from $1 \leq X, Y, Z \leq 5$ and $X + Y \leq Z$, the values 4 and 5 can be removed from the domain of X and Y). Usually this state does not end with a definite value for each variable, which is sought for in a second search process, called *labeling* or *enumeration*: variables in the problem are assigned values within their domains until all of them have a unique value satisfying all the equations in the problem. In this process, if some assignment is inconsistent with the equations, the system backtracks, looking for alternative assignments. These two phases are radically different in that propagation is a deterministic process, while *labeling* is non-deterministic. In fact, after each assignment made by the labeling process, a series of propagation steps can take place. In a real program several propagation / labeling phases can occur.

In the example, the initial propagation phase, before the labeling, reduces the domains of the variables to be:

$$\mathtt{s}_0 : X \in \{3, 4, 5\} \wedge Y \in \{1, 2, 3\} \wedge Z \in \{1, 2\}$$

Different propagation schemes can have different effectiveness and yield domains more or less tight. This is not a soundness / completeness problem, as labeling will eventually remove inconsistent values. Removing as much values as possible is advantageous, since this will make the search space smaller, but the computational cost of a more precise domain narrowing has to be balanced with the savings in the search.

[1] Note that many constraint systems do not have a complete solution procedure.
[2] Large default ranges are automatically selected if this initialization is not present.

If we assume variables are *labeled* in lexicographical order, the next search step will generate three different states, resulting from instantiating `X` to the values in its domain. Each of these instantiations will in turn start a propagation (and simplification) which will lead to the following three states:

$$\begin{aligned} \mathsf{s}_{01} &: X = 3 \wedge Y = 1 \wedge Z = 1 \\ \mathsf{s}_{02} &: X = 4 \wedge Y = 2 \wedge Z = 1 \\ \mathsf{s}_{03} &: X = 5 \wedge Y \in \{1,2,3\} \wedge Z \in \{1,2\} \end{aligned}$$

s_{01} and s_{02} are completely determined, and are final solutions. If only one solution were needed, the execution could have finished when $X = 3$ was executed. If more solutions were required, s_{02} would be delivered and further exploration performed, starting at s_{03}, which would result in the children states s_{031}, s_{032}, and s_{033}, where Y is instantiated to the values in its domain:

$$\begin{aligned} \mathsf{s}_{031} &: X = 5 \wedge Y = 1 \wedge Z = 2 \\ \mathsf{s}_{032} &: X = 5 \wedge Y = 3 \wedge Z = 1 \end{aligned}$$

At this point, no more search either propagation is possible, and all the solutions to the constraint problem have been found. Note that the combination $X = 5, Y = 2$ leads to an inconsistent valuation, and is not shown.

3.1 Discrete Fuzzy Prolog Syntax

Each Fuzzy Prolog clause has an additional argument in the head which represents its truth value in terms of the truth values of the subgoals of the clause body. Though the syntax is analogous to the continuous case (see 2.3), a union of intervals represents in the current approach a range of discrete values. An interval of discrete values or a real number are particular cases of union of intervals. The following examples (with decimal precision) illustrate the concrete syntax of programs:

$youth(45) \leftarrow\ [0.2, 0.5] \bigcup\ [0.8, 1]$	`youth(45,V)::~[0.2,0.5]v[0.8,1].`
$tall(john) \leftarrow 0.7$	`tall(john,V)::~ 0.7.`
$swift(john) \leftarrow\ [0.6, 0.8]$	`swift(john,V)::~ [0.6,0.8].`
$goodplayer(X) \leftarrow_{min}$	`goodplayer(X,V)::~min`
$\qquad tall(X),$	`tall(X,Vt),`
$\qquad swift(X)$	`swift(X,Vs).`

These clauses are expanded at compilation time to constrained clauses that are managed by CLP($\mathcal{FD}$) at run-time. Predicates $. = ./2$, $. < ./2$, $. <= ./2$, $. > ./2$ and $. >= ./2$ are the Ciao CLP operators for representing constraint inequalities. For example the first fuzzy fact is expanded to these Prolog clauses with constraints

```
youth(45,V):- V in 2 .. 5, V in 8 .. 10.
```

And the fuzzy clause

```
    p(X, Vp) ::~ min q(X, Vq),r(X, Vr).
```

is expanded to

```
p(X,Vp) :- q(X,Vq),r(X,Vr),
           minim([Vq,Vr],Vp),
           Vp in 0..10.
```

One of the main advantages of discrete Fuzzy Prolog is that it can be applied to a distributed environment more easily than its continuous counterpart [4]. In a distributed setting, each agent has a partial knowledge of the truth values associated to the fuzzy predicates that needs to be contrasted with the rest in order to obtain a consistent global picture. That is the reason why truth value variables appear explicitly in fuzzy predicates.

The code of predicate `minim/2`[3] is included at compile-time. Its function is to add constraints to the truth value variables in order to implement the T-norm *min*, in an analogous way to that in [4]. The implementation in the case of CLP($\mathcal{FD}$) is the following:

```
minim([],_).
minim([X],X).
minim([X,Y|Rest],Min):-
        min(X,Y,M),
        minim([M|Rest],Min).

min(X,Y,Z):-
        bounds(X, Lx, _Ux),
        bounds(Y, Ly, _Uy),
        do_min(Lx, Ly, X, Y, Z).
```

```
do_min(Lx, Ly, X, _Y, Z) :-
        Lx .=<. Ly,
        X .=. Z.
do_min(Lx, Ly, _X, Y, Z) :-
        Lx .>. Ly,
        Y .=. Z.
```

Like in the continuous case new aggregation operators can be added to the system without any effort, thus it is easily user-extensible. We have implemented the discrete version of the aggregation operators provided in [4] (min, max, prod, luka).

3.2 Syntactic Sugar

Fuzzy predicates with piecewise linear continuous membership functions like `teenager/2` in Figure 3 can be written in a compact way using the operator :: #.

```
teenager ::# fuzzy_predicate([(0,0),(8,0),(12,1),(14,1),(18,0),(120,0)]).
```

This friendly syntax is translated to arithmetic constraints. We can even define the predicate directly if we so prefer. The code expansion is the following:

```
teenager(X,0):- X .>=. 0,
                X .<. 8.
teenager(X,V):- X .>=. 8,
                X .<. 12,
                4*V .=. -12+X.
teenager(X,1):- X .>=. 12,
                X .<. 14.
```

```
teenager(X,V):- X .>=. 14,
                X .<. 18,
                4*V .=. 18-X.
teenager(X,0):- X .>=. 18,
                X .=<. 120.
```

[3] minim/2 uses predicate bounds/3 from the Ciao Prolog CLP($\mathcal{FD}$)library. This predicate obtains the upper and lower bounds of a finite domain variable.

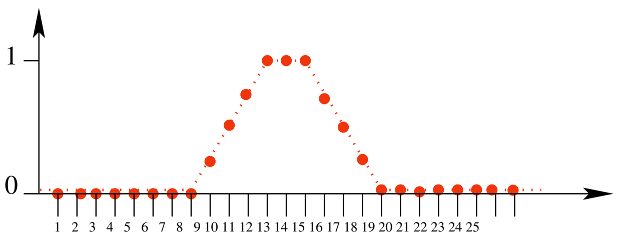


Fig. 3. Uncertainty level of a fuzzy predicate

In Fuzzy Prolog it is possible to obtain the fuzzy negation of a fuzzy predicate. For the predicate $p/3$, we will define a new fuzzy predicate called, for example, $notp/3$ with the following line:

```
notp ::# fnot  p/3.
```

that is expanded at compilation time as:

```
notp(X,Y,V) :- p(X,Y,Vp), V .=. 1 - Vp.
```

4 A Distributed CLP($\mathcal{FD}$) Approach for Collaborative Agent Systems

Collaborative agent systems is a particular case of the well known multi agent systems setting [6] where agents themselves determine their social behavior according to their particular interests. This behavior may range from sheer competitiveness to collaboration. In the first case, agents act in a scenario where negotiation [16] is the key to solve conflicts in a decentralized way. Agents exchange proposals and counter proposals iteratively until an agreement is reached. On the other hand, collaborative agents pursue common objectives using distributed scheduling policies. In early works about agents systems [9], scheduling was generally done in advance, previous to execution. Currently, work has been done in order to allow agents to coordinate dynamically, like [10]. However, it can be argued that there is not really such absolute distinction between competitive and collaborative agents. Moreover, the same agents can use different criteria in order to combine local and global objectives, though paying the price of an increased complexity that may lead to undesired effects. In approaches like [10], adaptiveness confines itself to parameterizing an utility function that defines the behavior of the agent.

Coordination in multi agents systems is a widely studied subject. Most coordination approaches are based on utility functions, in the case of centralized models, and game theory, in distributed proposals. Games theory assumes a competitive scenario where agents negotiate to maximize their own benefit. Thus, further coordination strategies are needed that favor a decentralized setting for collaborative agents. In this direction, agent systems have been proposed that act in a goal oriented manner using constraints [13] to represent (incomplete) knowledge in a dynamically changing environment.

In our context, Fuzzy problems are modeled using CLP($\mathcal{FD}$), providing an homogeneous knowledge representation that avoids the use of language gateways like KIF and KQML upon agent communication. Each agent has a partial point of view of the whole picture, represented by a subset of the constraints used to model the problem. Since agents are pursued to be as independent from each other as possible, no global control is implemented and decisions are made asynchronously. However, some mechanism has to be established in order to backtrack from globally inconsistent situations to a consistent previous state that allows to take up execution and eventually reach a global solution, thus granting a coherent behavior. Communication between agents is done via constraints exchange. Since our approach is based on a CLP($\mathcal{FD}$) implementation which handles constraints at a high-level, using attributed variables which contain first-order terms, sending the store is made simply by term transmission (which can be optimized by marshaling and compression).

Both execution mechanisms of CLP($\mathcal{FD}$), *propagation* and *labeling*, are amenable to be distributed under different perspectives. If aimed to increasing performance, or-parallelism can be applied to labeling given the independence between the partial solutions generated by the potential assignments of values to variables. But, in the environment of collaborative agents, where the knowledge of the problem is typically distributed and spread among a number of entities, we are more interested in how to reach a solution for problems which are distributed in an and-parallel fashion. In this context, *propagation* and *labeling* can be executed in a collaborative agent system where the constraints store of the problem has been split among agents, as described above. To this end, given its conceptual simplicity, we have used algorithms based on the Asynchronous Backtracking algorithm [19] (ABT) for Distributed Constraint Satisfaction problems [22]. We present an extension of the Ciao Prolog language that implements further work on ABT to execute CLP($\mathcal{FD}$) programs in a distributed fashion, and apply it to the case of collaborative fuzzy agents systems.

4.1 Asynchronous Backtracking and Collaborative Agents Systems

ABT was initially designed to model agent systems where each agent, owning a single variable, is connected to others by means of the constraints where these variables appear. ABT problems are represented as a directed graph where nodes are the agents (as well as the variables in this framework) and links are the constraints existing between their variables. Asynchronously, each agent assigns a value to its variable. This assignment is communicated to other agents, which evaluate it and determine whether it is consistent or not with their own assignment and their (partial) view of the system. In case of inconsistence a backtracking process is initiated which returns to the previous consistent state of execution.

Agents exchange two types of messages. *ok?* messages are used to send a variable assignment to an agent for evaluation and *nogood* messages are sent by the evaluating agent when an inconsistence is detected. Upon receipt of an *ok?* message the evaluating agent first checks the consistence of this assignment with his own and the rest of his partial view. If consistence is not such, it tries to find a new value for its variable that reverts the situation to a coherent state. If there is no such value, the agent starts a backtracking process and sends a *nogood* message to one of the other agents.

ABT is sound and complete (if there is a solution, it finds it and fails otherwise) but originally presented problems with infinite processing loops, and asynchronous changes. In order to solve infinite loops, where agents change their values over and over never reaching a stable state, it is necessary to set a partial order among agents which determines for every constraint, or link in the directed graph, which agent acts as an evaluator and which is the evaluated. This order sets the direction of the link and the *ok?* messages. In case of inconsistence, *nogood* messages are sent to the agent in the current agent's partial view which is the lowest according to this order.

On the other hand, an evaluating agent may send *nogood* messages to an agent which has corrected its value already. To avoid this *nogood* messages include the partial view of the agent that started backtracking. Thus, the recipient of the message only changes its value if it is consistent with the content of the message. As an optimization, known inconsistent states can be incorporated as new constraints to the systems.

ABT assignments are strongly committed. A selected value is not changed until an exhaustive search is performed by agents below in the partial order. This drawback is generally counterbalanced by the simplicity of the algorithm. Other algorithms based on ABT, like Asynchronous Weak-Commit Search [20] (AWC) get over this inconvenience.

However, ABT lacks several features which are desirable for a complete collaborative agents system. ABT only considers the case of just one variable per agent, there is not distributed propagation (only labeling), and no means to detect termination are provided. In our approach we have extended ABT with these features. Thanks to the resolution mechanism in CLP($\mathcal{FD}$) systems, agents can keep several variables by dividing execution into two stages which interact with each other, a global one, directly extracted from ABT, and one local to each agent where local consistency is ensured. Distributed propagation finds consistent intervals of truth values and reduces the search space during an eventual labeling phase increasing efficiency by reducing the number of messages exchanged by the agents in this phase. To implement distributed propagation of the constraint stores of each agent (minimizing the number of messages exchanged) a minimal, dynamic spanning tree is built using the Dijkstra-Scholten algorithm [3] that covers all the agents. When an agent first receives an *ok?* message it classifies the sending agent as its parent. After that, it replies to every *ok?* with an *ack* that contains its store so that the other agent can update its own store. When the agent is in a quiescent state it leaves the spanning tree by sending an *ack* to its parent. If it receives a further *ok?* this process starts all over.

Termination of propagation or labeling is detected by means of a Chandy-Lamport algorithm [2]. This allows to combine both phases until a problem can be declared as solved or failed.

5 A Practical Example

The distributed approach described in the previous section can be quite straightforwardly applied to handle complex fuzzy problems where global knowledge is inherently distributed and shared by a number of agents, aiming to achieve a global solution in a collaborative way. In this context we will show an example related with criminal identification of suspects. From the point of view of our test case police investments need to take into consideration the following factors:

- Physical aspects regarding the results of pattern matching between the robot portrait database and the photography of the suspect.
- A psychic diagnostic provided by the psychologist.
- The set of evidences obtained by the CSI people.

None of these data are crisp. The subject is suspect in a degree which depends on the combination of the truth values of all the factors previously enumerated. Thus, this is a fuzzy problem, and also a distributed problem as the information is obtained from different independent sources. In this case, our (collaborative) agents are the database examiner, the psychologist and the CSI department. Each agent has a share of the global knowledge, represented as fuzzy discrete variables. This partial knowledge is related with that contained in other agents by means of a series of constraints in which their respective variables are contained. For example, a constraint is that the combination of the truth values of physical and psychical analyzes is not allowed to be inferior to 0.5 (50% match with the suspect). Also, physical analysis is considered to be more reliable than psychical analysis and evidence is the most reliable information.

The discrete Fuzzy Prolog program modeling this problem is:

```
suspect(Person, V) :: ~inter_m
        allocate_vars([Vp, Vs, Ve]),
        physically_suspect(Person, Vp, Vs),
        psychically_suspect(Person, Vs, Vp),
        evidences(Person, Ve, Vp, Vs).
```

The aggregation operator we have used is *inter_m* that intends to reach a consensus amongst different fuzzy agents (intersection). It returns the minimum if consensus does not exist (i.e. intersection is void). due to its adequateness to collaborative problems. Predicate *physically_suspect*/3 provides the degree of physical match, Vp, corresponding to the portrait of the *Person* with respect to the database of the police file. Note the value of this variable is related to the value of the psychical analysis by means of a constraint, being the reason why predicate *physically_suspect*/3 handles both variables. This is also the case in the other predicates.

The translation of this program to CLP($\mathcal{FD}$) yields the following program:

```
suspect(Person, V) :-
        allocate_vars([Vp, Vs, Ve]),
        V in 0..10,
        physically_suspect(Person, Vp, Vs),
        psychically_suspect(Person, Vs, Vp),
        evidences(Person, Ve, Vp, Vs),
        inter_m([Vp, Vs, Ve], V).
```

This program produces truth values as unions of intervals of discrete values (as defined in Section 2.1) by means of distributed CLP($\mathcal{FD}$) propagation (Section 3). It is also possible to obtain the enumeration of instantiated crisp values instead of fuzzy values by adding a call to $labeling([Vp, Vs, Ve])$, which generates an enumeration of the variables with consistent values.

Predicate *allocate_vars*/1 has been included to assign variables to agents. In this case, each agent owns one variable. The agent owning a variable is the one to propose the assignment of values to it during labeling, according to the algorithm ABT described in [19]. In our case Vp is assigned to agent $a1$, Vs to agent $a2$, and Ve to agent $a3$.

Partial knowledge stored in each agent is formulated in terms of constraint expressions. In this case, the following represents the knowledge each agent has about the suspect identification problem (operator @ injects this knowledge into either agent $a1$, $a2$, or $a3$):

```
Cp:   physically_suspect(Person, Vp, Vs) :-
              scan_portrait_database(Person, Vp),
              Vp * Vs .>=. 50 @ a1.
Cs:   psychically_suspect(Person, Vs, Vp) :-
              psicologist_diagnostic(Person, Vs),
              Vs .<. Vp @ a2.
Ce:   evidences(Person, Ve, Vp, Vs) :-
              police_database(Person, Ve),
              Ve .>=. Vp,
              Ve .>=. Vs @ a3.
```

In our example each agent obtains an initial truth value for its corresponding variable which is represented by means of the following facts:

```
scan_portrait_database(peter, Vp) :- Vp in 4..10.
scan_portrait_database(jane, Vp) :- Vp in 8..10.
psicologist_diagnostic(peter, Vs) :- Vs in 3..10.
psicologist_diagnostic(jane, Vs) :- Vs in 6..8.
police_database(peter, Ve) :- Ve in 7..10.
police_database(jane, Ve) :- Ve in 1..4.
```

In order to solve the global problem, coordination between the three agents of the example is necessary. This coordination can be viewed at the light of message exchange among agents, (figures 4 and 5). As seen in section 4, the distributed fuzzy problem is solved using an evolution of ABT. The main characteristic of this extension is that not only can it handle labeling but also propagation, hence providing the means to obtain a fixpoint of the variables truth values. In order to implement an effective distributed propagation scheme it is also necessary to build a minimal spanning tree, generated according to the Dijkstra-Scholten algorithm, that can be traversed upwards (not only downwards as in ABT) by *ack* messages generated upon receipt of *ok?* messages in the circumstances described in [3]. In this case the evaluation order is $Vp > Vs > Ve$. *ack* messages contain the store of the evaluator agent so that the agent that previously sent the *ok?* can update its own store by means of executing local propagation. Upon change in the local store the corresponding part of the ABT protocol is respawned, submitting new *ok?* messages to the agent evaluators.

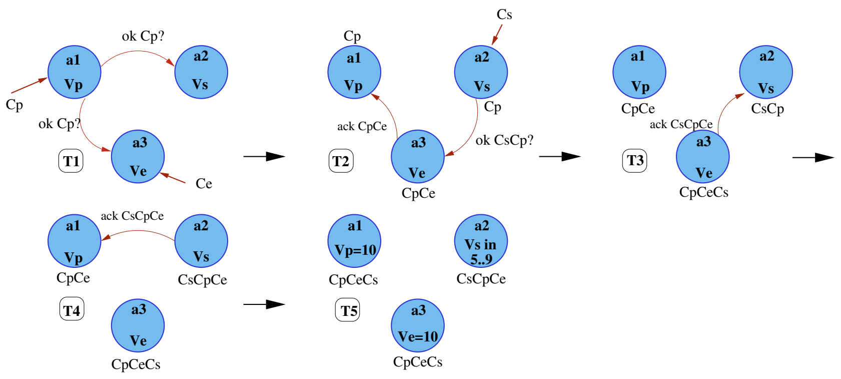


Fig. 4. Collaborative fuzzy agents interaction for suspect(peter,V)

In figure 4, agents are initially in a stable state when Cp and Ce are injected into agents Vp and Ve, respectively. This begins a propagation process that will eventually lead to a global fixpoint. In a given moment during T1 Vp sends its store to its evaluator agents Vs and Ve, which will match the store of Vp against their own. In T2, Vs sends its store to Ve, and due to a network delay the *ok?* message containing it arrives earlier than Vp's. Hence, according to the Dijkstra-Scholten algorithm, Ve considers Vs as its parent into the spanning tree. Thus, Ve sends an *ack* message containing its new store to Vp. In T3, Vp has already updated its store with the previous message and as Ve does not have any pending *ack* to receive it acknowledges its parent, Vs, with its current store. Vs updates it stores with the content of the previous message and finally, as it has no pending *acks* to receive, it acknowledges its own parent, Vp. In T5, a global fixpoint has been reached and a consensus about Vp,Vs, and Ve has been successfully agreed.

On the contrary, figure 5 shows a case in which the stores of agents Vp and Ve are inconsistent. In T1, upon evaluation of the *ok?* sent by Vp, Ve detects an inconsistency, overrides its store for the current problem, and, according to

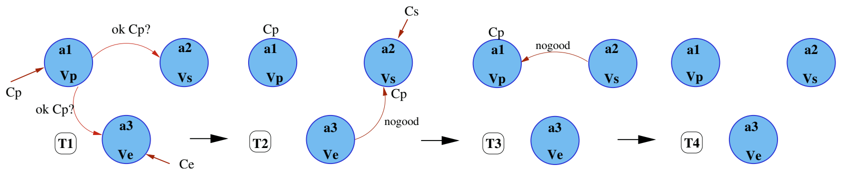


Fig. 5. Collaborative fuzzy agents interaction for suspect(jane,V)

ABT, sends a *nogood* message to Vs, which is the agent of lower evaluation order amongst the agents Ve evaluates. Vs does the same and propagates the *nogood* to Vp, which in T4 also overrides its store and produces a failure.

6 Conclusions and Future Work

From the study of fuzzy problems we have realized that most real fuzzy models are represented in practice with a discrete number of values. The range of values generally depends on the accuracy required by the problem. Although a discrete representation seems to be more limited that a continuous representation, the implementation of fuzzy problems using constraints over finite domains has several advantages as the possibility of modeling distributed problems with a simple implementation. In this work, we provide the formal framework for our discrete fuzzy Prolog, we describe the details of its implementation using CLP($\mathcal{FD}$) and we take advantage of this implementation to solve complex distributed fuzzy problems. We have used an extension of ABT to implement collaborative agent systems over CLP($\mathcal{FD}$). The power of this fuzzy model and its simplicity makes its use very interesting for taking advantage of all tools developed for finite domains. We plan to continue this research with future work like studying the impact of other tools implemented over CLP($\mathcal{FD}$) in the resolution of fuzzy tools, and comparing efficiency of continuous Fuzzy Prolog (implemented on CLP($\mathcal{R}$)) and discrete fuzzy tools. We also intend to improve the flexibility of the algorithm with respect to adding or removing agents, and finally apply our approach to fields like the semantic web or business rules to improve the distributed reasoning processes involved.

References

1. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint logic programming: syntax and semantics. In *ACM TOPLAS*, volume 23, pages 1–29, 2001.
2. K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
3. E.W. Dijkstra and C.S. Sholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, (11):1–4, 1980.

4. S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy prolog: A new approach using soft constraints propagation. *Fuzzy Sets and Systems, FSS*, 144(1):127–150, 2004. ISSN 0165-0114.
5. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
6. Michael N. Huhns and Larry M. Stephens. Multiagent Systems and Societies of Agents. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 79–120. The MIT Press, Cambridge, MA, USA, 1999.
7. J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, pages 196–219. University of Melbourne, MIT Press, 1987.
8. J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The clp(∇) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
9. Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
10. V. Lesser, K. Decker, N. Carver, D. Neiman, M. Nagendra Prasad, and T. Wagner. Evolution of the GPGP domain-independent coordination framework. Technical Report UM-CS-1998-005, 1998.
11. R.C.T. Lee. Fuzzy logic and the resolution principle. *Journal of the Association for Computing Machinery*, 19(1):119–129, 1972.
12. J. Medina, M. Ojeda-Aciego, and P. Votjas. Multi-adjoint logic programming with continous semantics. In *LPNMR*, volume 2173 of *LNCS*, pages 351–364, Boston, MA (USA), 2001. Springer-Verlag.
13. Alexander Nareyek. *Constraint-Based Agents*, volume 2062. Springer, 2001.
14. H. T. Nguyen and E. A. Walker. *A first Course in Fuzzy Logic.* Chapman & Hall/Crc, 2000.
15. Z. Shen, L. Ding, and M. Mukaidono. Fuzzy resolution principle. In *Proc. of 18th International Symposium on Multiple-valued Logic*, volume 5, 1989.
16. C. Sierra, N. R. Jennings, P. Noriega, and S. Parsons. A framework for argumentation-based negotiation. *Lecture Notes in Computer Science*, 1365.
17. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.
18. P. Vojtas. Fuzzy logic programming. *Fuzzy sets and systems*, 124(1):361–370, 2001.
19. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
20. M. Yokoo. Asynchronous Weak-Commitment Search for Solving Distributed Constraint Satisfaction Problems. In *First International Conference on Principles and Practice of Constraint Programming*, pages 88–102, 1995.
21. L. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy sets and systems*, 1(1):3–28, 1978.
22. Ying Zhang and Alan K. Mackworth. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. Technical Report TR-92-30, 1992.

Improved Fusion for Optimizing Generics

Artem Alimarine and Sjaak Smetsers

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{A.Alimarine,S.Smetsers}@cs.ru.nl

Abstract. Generic programming is accepted by the functional programming community as a valuable tool for program development. Several functional languages have adopted the generic scheme of type-indexed values. This scheme works by specialization of a generic function to a concrete type. However, the generated code is extremely inefficient compared to its hand-written counterpart. The performance penalty is so big that the practical usefulness of generic programming is compromised. In this paper we present an optimization algorithm that is able to completely eliminate the overhead introduced by the specialization scheme for a large class of generic functions. The presented technique is based on consumer–producer elimination as exploited by fusion, a standard general purpose optimization method. We show that our algorithm is able to optimize many practical examples of generic functions.

Keywords: program transformation, fusion, generic/polytypic programming.

1 Introduction

Generic programming is recognized as an important tool for minimizing boilerplate code that results from defining the same operation on different types. One of the most wide-spread generic programming techniques is the approach of type-indexed values [6]. In this approach, a generic operation is defined once for all data types. For each concrete data type an instance of this operation is generated. This instance is an ordinary function that implements the operation on the data type. We say that the generic operation is *specialized* to the data type.

The generic specialization scheme uses a structural view on a data type. In essence, an algebraic type is represented as a sum of products of types. The structural representation uses *binary* sums and products. Generic operations are defined on these structural representations. Before applying a generic operation the arguments are converted to the structural representation, then the operation is applied to the converted arguments and then the result of the operation is converted back to its original form.

A programming language's feature is only useful in practice, if its performance is adequate. Directly following the generic scheme leads to very inefficient code, involving numerous conversions between values and their structural representations. The generated code additionally uses many higher-order functions (representing dictionaries corresponding to the type arguments). The inefficiency of generated code severely compromises the utility of generic programming.

In the previous work [2] we used a partial evaluation technique to eliminate generic overhead introduced by the generic specialization scheme. We proved that the described technique completely removes the generic overhead. However, the proposed optimization technique lacks termination analysis, and therefore works only for non-recursive functions. To make the technique work for instances on recursive types we abstracted the recursion with a Y-combinator and optimized the non-recursive part. This technique is limited to generic functions that do not contain recursion in their types, though the instance types can be recursive. Another disadvantage of the proposed technique is that it is tailored specifically to optimize generics, because it performs the recursion abstraction of generic instances.

The present paper describes a general purpose optimization technique that is able to optimize a significantly larger class of generic instances. In fact, the proposed technique eliminates the generic overhead in nearly all practical generic examples. When it is not able to remove the overhead completely, it still improves the code considerably. The presented optimization algorithm is based on fusion [3,4]. In its turn, fusion is based on the consumer-producer model: a producer produces data which are consumed by the consumer. Intermediate data are eliminated by combining (*fusing*) consumer-producer pairs.

The contributions of the present paper are: (1) The original fusion algorithm is improved by refining both consumer and producer analyses. Our main goal is to achieve good fusion results for generics, but the improvements also appear to pay off for non-generic examples. (2) We describe the class of generic programs for which the generic overhead is completely removed. This class includes nearly all practical generic programs.

In the next section we introduce the code generated by the generic specialization. This code is a subject to the optimization described further in the paper. The generated code is represented in a core functional language defined in section 3. Section 4 defines the semantics of fusion with no termination analysis. Standard fusion with termination analysis [3] is described in section 5. Sections 6 and 7 introduce our extensions to the consumer and the producer analyses. Fusion of generic programs is described in section 8. The performance results are presented in section 9. Section 10 discusses related work. Section 11 reiterates our conclusions.

2 Generics

In this section we give a brief overview of the generic specialization scheme which is based on the approach by Hinze [6]. Generic functions exploit the fact that any data type can be represented in terms of sums, pairs and unit, called the *base types*. These base types can be specified by the following Haskell-like data type definitions.

$$\textbf{data}\ \mathbb{1} = \mathsf{Unit} \qquad \textbf{data}\ a \times b = \mathsf{Pair}\ a\ b \qquad \textbf{data}\ a + b = \mathsf{Inl}\ a \mid \mathsf{Inr}\ b$$

A generic (*type-indexed*) function g is specified by means of instances for these base types. The structural representation of a concrete data type, say T, is used to generate an instance of g for T. The idea is to convert an object of type T first to its structural representation, apply the generic operation g to it, and convert the resulting object back from its structural to its original representation, and vise versa.

Suppose that the generic function g has generic (*kind-indexed*) type G. Then the instance g_T of g for the concrete type T has the following form.

$$g_T f_1 \cdots f_n = adapt_{\langle G,T\rangle}(g_{T^\circ} f_1 \cdots f_n)$$

where T° denotes the structural representation of T, g_{T° represents the instance of g on T°, and the adapter $adapt_{\langle G,T\rangle}$ takes care of the conversion between T and T°. We will illustrate this generic specialization scheme with a few examples. The following familiar data types are used in the examples.

$$\textbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a) \quad \textbf{data}\ \mathsf{Tree}\ a = \mathsf{Leaf}\ a \mid \mathsf{Branch}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

They have the following structural representation

$$\textbf{type}\ \mathsf{List}^\circ\ a = \mathbb{1} + a \times \mathsf{List}\ a \qquad \textbf{type}\ \mathsf{Tree}^\circ\ a = a + \mathsf{Tree}\ a \times \mathsf{Tree}\ a$$

Note that these representations are *not* recursive: they only capture the outermost structure up to the argument types of each data constructor. A type and its structural representation are isomorphic. The isomorphism is witnessed by a pair of conversion functions. For instance, for lists these functions are

$$\begin{aligned}
&\mathsf{convTo}_{\mathsf{List}} :: \mathsf{List}\ a \to \mathsf{List}^\circ\ a\\
&\mathsf{convTo}_{\mathsf{List}}\ l \quad = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Nil} \to \mathsf{Inl}\ \mathsf{Unit};\ \ \mathsf{Cons}\ x\ xs \to \mathsf{Inr}\ (\mathsf{Pair}\ x\ xs)\\
&\mathsf{convFrom}_{\mathsf{List}} :: \mathsf{List}^\circ\ a \to \mathsf{List}\ a\\
&\mathsf{convFrom}_{\mathsf{List}}\ l = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Inl}\ \mathsf{Unit} \to \mathsf{Nil};\ \ \mathsf{Inr}\ (\mathsf{Pair}\ x\ xs) \to \mathsf{Cons}\ x\ xs
\end{aligned}$$

To define a generic function g the programmer has to provide the *generic type*, say G, and the instances on the base types (the *base cases*). For example, generic mapping is given by the following generic type and base cases

$$\begin{aligned}
&\textbf{type}\ \mathsf{Map}\ a\ b = a \to b\\
&\mathsf{map}_{\mathbb{1}} \qquad\quad\ = \mathsf{case}\ u\ \mathsf{of}\ \ \mathsf{Unit} \to \mathsf{Unit}\\
&\mathsf{map}_{\times}\ l\ r\ p \quad = \mathsf{case}\ p\ \mathsf{of}\ \ \mathsf{Pair}\ x\ y \to \mathsf{Pair}\ (l\ x)\ (r\ y)\\
&\mathsf{map}_{+}\ l\ r\ e \quad = \mathsf{case}\ e\ \mathsf{of}\ \ \mathsf{Inl}\ x \to \mathsf{Inl}\ (l\ x);\ \ \mathsf{Inr}\ y \to \mathsf{Inr}\ (r\ y)
\end{aligned}$$

This is all that is needed for the generic specializer to build an instance of map for any concrete data type T. As said before, such an instance is generated by interpreting the structural representation T° of T, and by creating an appropriate adapter. For instance, the generated mapping for List° is

$$\begin{aligned}
&\mathsf{map}_{\mathsf{List}^\circ} :: \mathsf{Map}\ a\ b \to \mathsf{Map}\ (\mathsf{List}^\circ\ a)\ (\mathsf{List}^\circ\ b)\\
&\mathsf{map}_{\mathsf{List}^\circ}\ f = \mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{\mathsf{List}}\ f))
\end{aligned}$$

Note how the structure of $\mathsf{map}_{\mathsf{List}^\circ}$ directly reflects the structure of List°. The adaptor converts the instance on the structural representation into an instance on the concrete type itself. E.g., the adapter converting $\mathsf{map}_{\mathsf{List}^\circ}$ into $\mathsf{map}_{\mathsf{List}}$ (i.e. the mapping function for List), has type

$$adapt_{\langle \mathsf{Map},\mathsf{List}\rangle} :: \mathsf{Map}\ (\mathsf{List}^\circ\ a)\ (\mathsf{List}^\circ\ b) \to \mathsf{Map}\ (\mathsf{List}\ a)\ (\mathsf{List}\ b)$$

The code for this adapter function is described below. We can now easily combine $adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}$ with $\mathsf{map}_{\mathsf{List}^\circ}$ to obtain a mapping function for the original List type.

$$\begin{array}{l} \mathsf{map}_{\mathsf{List}} :: \mathsf{Map}\ a\ b \to \mathsf{Map}\ (\mathsf{List}\ a)\ (\mathsf{List}\ b) \\ \mathsf{map}_{\mathsf{List}}\ f = adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}\ (\mathsf{map}_{\mathsf{List}^\circ}\ f) \end{array}$$

The way the adaptor works depends on the type of the generic function as well as on the concrete data type for which an instance is created. So called *embedding projections* are used to devise the automatic conversion. In essence such an embedding projection distributes the original conversion functions (the isomorphism between the type and its structural representation) over the type of the generic function. In general, the type of a generic function can contain arbitrary type constructors, including arrows. These arrows may also appear in the definition of the type for which an instance is derived. To handle such types in a uniform way, conversion functions are packed into *embedding-projection pairs*, EPs (e.g. see [7]), which are defined as follows.

$$\mathbf{data}\ a \rightleftarrows b = \mathsf{EP}\ (a \to b)\ (b \to a)$$

For instance, packing the List conversion functions into an EP leads to:

$$\begin{array}{l} \mathsf{conv}_{\mathsf{List}} :: \mathsf{List}\ a \rightleftarrows \mathsf{List}^\circ\ a \\ \mathsf{conv}_{\mathsf{List}} = \mathsf{EP}\ \mathsf{convTo}_{\mathsf{List}}\ \mathsf{convFrom}_{\mathsf{List}} \end{array}$$

Now the adapter for G and T can be specified in terms of embedding projections using the EP that corresponds to the isomorphism between T and T° as a basis. To facilitate the instance generation scheme, embedding projections are represented as instances of a single generic function ep, with generic type $a \rightleftarrows b$. The base cases for this (built in) generic function are predefined and consist, besides the usual instances for sum, pair and unit, of instances on $\to$ and $\rightleftarrows$. The generic specializer generates the instance of ep specific to a generic function, again by interpreting its generic type. E.g. for mapping (with the generic type $\mathsf{Map}\ a\ b$) we get:

$$\begin{array}{l} \mathsf{ep}_{\mathsf{Map}} :: (a_1 \rightleftarrows a_2) \to (b_1 \rightleftarrows b_2) \to (\mathsf{Map}\ a_1\ b_1 \rightleftarrows \mathsf{Map}\ a_2\ b_2) \\ \mathsf{ep}_{\mathsf{Map}}\ a\ b = \mathsf{ep}_{\to}\ a\ b \end{array}$$

Now the adaptor $adapt_{\langle \mathsf{Map},\mathsf{List}\rangle}$ is the from-component (i.e. the second argument of the $\rightleftarrows$ constructor) of this embedding projection applied to $\mathsf{conv}_{\mathsf{List}}$ twice.

$$adapt_{\langle \mathsf{Map},\mathsf{List}\rangle} = \mathsf{from}\ (\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}})$$

The adaptor itself is not recursive, but will be invoked each time $\mathsf{map}_{\mathsf{List}}$ is applied to a list element converting the whole list to its structural representation.

To compare the generated version of map with its handwritten counterpart

$$\mathsf{map}\ f\ l = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to \mathsf{Nil};\ \mathsf{Cons}\ x\ xs \to \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)$$

we have inlined the adapter and the instance for the structural representation in the definition of $\mathsf{map}_{\mathsf{List}}$ resulting in

$$\mathsf{map}_{\mathsf{List}}\ f = \mathsf{from}\ (\mathsf{ep}_{\mathsf{Map}}\ \mathsf{conv}_{\mathsf{List}}\ \mathsf{conv}_{\mathsf{List}})\ (\mathsf{map}_{+}\ \mathsf{map}_{\mathbb{1}}\ (\mathsf{map}_{\times}\ f\ (\mathsf{map}_{\mathsf{List}}\ f)))$$

Clearly, the generated version is much more complicated than the handwritten one, not only in terms of readability but also in terms of efficiency. The reasons for inefficiency are the intermediate data structures for the structural representation and the extensive usage of higher-order functions. In the rest of the paper we present an optimization technique for generic functions which is capable of removing all the generic overhead.

3 Language

In this section we present the syntax of a simple core functional language that supports essential aspects of functional programming such as pattern matching and higher-order functions. We define the syntax in two steps: expressions and functions.

Definition 3.1 (The language).

– *The set of* expressions *is defined as*

$$E ::= x \mid C\,\vec{E} \mid F\,\vec{E} \mid x\ \vec{E}.$$

Here x ranges over variables, C over data constructors and F over function symbols. The vector $\vec{V}$ *stands for* $(V_1, \ldots, V_n)$

– *Each (function or constructor) symbol has an* arity*: a natural number that indicates the maximum number of arguments to which the symbol can be applied. An expression E is* well-formed *if the actual arity of applications occurring in E never exceeds the formal arity of the applied symbols.*

– *The set of* function bodies *is defined as follows.*

$$B ::= E \mid \mathsf{case}\ x\ \mathsf{of}\ C_1\,\vec{x_1} \to E_1 \cdots C_n\,\vec{x_n} \to E_n$$

– *A* function definition *has the form* $F\,\vec{x} = B_F$ *with* $\mathrm{FV}(B_F) \subseteq \vec{x}$*. The* arity *of F is $|\vec{x}|$ (i.e. the length of $\vec{x}$).* $\mathrm{FV}(B)$ *stands for the free variables of B.*

Pattern matching is allowed only at the top level of a function definition. Moreover, only one pattern match per function is permitted and the patterns themselves have to be simple (free of nesting).

Data constructors are introduced via an *algebraic type definition.* Such a type definition not only specifies the *type* of each data constructor but also its *arity.* For readability reasons in this paper we will use a Haskell-like syntax in the examples.

4 Semantics of Fusion

Most program transformation methods use the so-called *unfold/fold* mechanism to convert expressions and functions. During an *unfold step*, a call to a function is replaced by the corresponding function body in which appropriate parameter substitutions have been performed. During a *fold step*, an expression is replaced by a call to a function of which the body matches that expression.

In the present paper we will use a slightly different way of both unfolding and folding. First of all, we do not unfold all function applications but restrict ourselves to so called *consumer–producer* pairs. In a function application $F(\ldots, S(\ldots), \ldots)$ the function F is called a consumer and the function or constructor S a producer. The intuition behind this terminology is that F consumes the result produced by S. Suppose we have localized a consumer–producer pair in an expression R. More precisely, R contains a subexpression $F\,\vec{E} \star (E_i) \star \vec{E'}$, with $E_i = S\,\vec{D}$. Here $\star$ denotes vector concatenation, and $F\,\vec{E} \star \vec{D}$ should be read as $F\,(\vec{E} \star \vec{D})$; not as $(F\,\vec{E}) \star \vec{D}$. The idea of *fusion* is to replace this pair of two calls in R by a single call to the combined function $F_i S$ resulting in the application $F_i S\,\vec{E} \star \vec{D} \star \vec{E'}$. (Here $F_i S$ stands for the name of the combined function, and should not be confused with, for instance a function F_i applied to S.) In addition, a new function is defined which contains the body of the consumer F in which $S\,\vec{y}$ is substituted for x_i. Note that this fusion mechanism does not require any explicit folding steps.

As an example consider the following definition of app, and the auxiliary function foo.

$$\begin{aligned} \mathsf{app}\ l\ t &= \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \rightarrow t;\ \ \mathsf{Cons}\ x\ xs \rightarrow \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t) \\ \mathsf{foo}\ x\ y\ z &= \mathsf{app}\ (\mathsf{app}\ x\ y)\ z \end{aligned}$$

The first fusion step leads to the creation of a new function, say $\mathsf{app_1app}$, and replaces the nested applications of app by a single application of this new function. The result is shown below.

$$\begin{aligned} \mathsf{foo}\ x\ y\ z &= \mathsf{app_1app} x\ y\ z \\ \mathsf{app_1app}\ x\ y\ z &= \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{app}\ y\ z;\ \ \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ h\ (\mathsf{app}\ (\mathsf{app}\ t\ y)\ z) \end{aligned}$$

A precise description of how the body of the new function $\mathsf{app_1app}$ is created can be found in [1]. The function foo does not contain consumer-producer pairs anymore; the only pair appears in the body of $\mathsf{app_1app}$, namely $\mathsf{app}\ (\mathsf{app}\ xs\ y)\ z$. Again these nested calls are replaced by $\mathsf{app_1app}$, and since $\mathsf{app_1app}$ has already been created, no further steps are necessary.

$$\mathsf{app_1app}\ x\ y\ z = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{app}\ y\ z;\ \ \mathsf{Cons}\ h\ t \rightarrow \mathsf{Cons}\ h\ (\mathsf{app_1app}\ t\ y\ z)$$

This example shows that we need to determine whether a function - $\mathsf{app_1app}$ in the example - has already been generated. To facilitate this, with each newly created function we associate a special unique name, a so called *symbol tree*. In fact, these symbol trees represent the creation history of the corresponding function in terms of the original set of functions. E.g. in the above example the name $\mathsf{app_1app}$ expresses that this function was obtained from fusing app with itself. A formal description of symbol trees, as well as of the fusion process itself is given in [1]. Here we restrict ourself to a few remarks on unfolding Unfolding is based on a notion of substitution for expressions. However, due to the restriction on our syntax with respect to cases and to higher-order expressions (recall the selector of a case expression as well as the first part of higher-order expression should be a variable) we cannot use a straightforward definition of substitution. Suppose we try to substitute an expression $D = F\,\vec{D'}$ for x in $(x\ \vec{E})$, which results in the application $F\,\vec{D'} \star \vec{E}$. However, this expression is not well-formed if $\mathsf{arity}(F) < |\vec{D'}| + |\vec{E}|$. To solve this problem we introduce a new function built from the definition of F by supplying it with additional arguments which *increases* its formal arity. In λ-calculus this operation is called *eta-expansion.* Besides, we have to be careful if a substitution is performed on a function body that starts with a pattern match: the result of such a substitution leads to an invalid expression if it substitutes a non-variable expression for the selector. In [1] we show that this problem can be solved by combining consumers and producers is such a way that no illegal (body) expressions are formed.

The following example illustrates how invalid case expressions are prevented. (The definition app is the same as above.)

$$\begin{aligned}\mathsf{len}\ l\ \ &= \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to 0;\ \mathsf{Cons}\ x\ xs \to \mathsf{Inc}\ (\mathsf{len}\ xs)\\ \mathsf{foo}\ x\ y &= \mathsf{len}\ (\mathsf{app}\ x\ y)\end{aligned}$$

The only consumer-producer pair occurs in foo. It will lead to the creation of a new function $\mathsf{len_1app}$. Without any precautions, the body of this new function becomes

$$\begin{aligned}&\mathsf{len_1app}\ x\ y = \mathsf{case}\ (\mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to t;\ \mathsf{Cons}\ x\ xs \to \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t))\ \mathsf{of}\\ &\qquad\qquad \mathsf{Nil} \to 0;\quad \mathsf{Cons}\ x\ xs \to \mathsf{Inc}\ (\mathsf{len}\ xs)\end{aligned}$$

which clearly violates the syntax for expressions. A correct body is obtained by pushing the outermost case into the alternatives of innermost one. In the Cons branch this leads to the elimination of the pattern match; in the Nil branch the len-function is re-introduced, leading to

$$\mathsf{len_1app}\ x\ y = \mathsf{case}\ l\ \mathsf{of}\ \ \mathsf{Nil} \to \mathsf{len}\ y;\ \ \mathsf{Cons}\ z\ zs \to \mathsf{Inc}\ (\mathsf{len}\ (\mathsf{app}\ zs\ y))$$

Now the only fusion candidate appears in this newly created function, namely $\mathsf{len}\ (\mathsf{app}\ zs\ y)$. It can directly be replaced by a recursive call to $\mathsf{len_1app}$, which eliminates the last consumer–producer pair.

Evaluation by fusion leads to a subset of expressions, so called *expressions in fusion normal form*, or briefly *fusion normal forms.* Also functions bodies are subject to fusion, leading to more or less the same kind of results. We can characterize these results by the following syntax.

Definition 4.1 (Fusion Normal Form). *The set of expressions in* fusion normal form *(FNF) is defined as follows.*

$$N ::= N' \mid F\,\vec{N'} \mid C\,\vec{N} \qquad N' ::= v \mid v\ \vec{N}$$

Observe that, in this form, functions are only applied to variables and higher-order applications, and never to constructors or functions. Our aim is show that fusion eliminates all basic data constructors. In [2] this is done by establishing a relation between the typing of an expression and the data constructors it contains after (symbolic, i.e. compile-time) evaluation: an expression in symbolic normal form does not contain any data constructors that is not included in a typing for that expression. In case of fusion normal forms (*FNF*s), we can derive a similar property. Note that this is more complicated than for symbolic evaluation because *FNF*s may still contain function applications. More specifically, let $CE(N)$ denote the collection of data constructors of the (body) expression N, and $CT(\sigma)$ denote the data constructors belonging to the type σ. (For a precise definition of $CE(\cdot)$, $CT(\cdot)$, see [2]). Then we have the following property.

Property 4.2 (Typing FNF). *Let $\mathcal{F}$ be a collection of functions in FNF. Suppose $F \in \mathcal{F}$ has type $\vec{\sigma} \to \tau$. Then $CE(F) \subseteq CT(\vec{\sigma}) \cup CT(\tau)$. In words: the data constructors possibly generated by F belong to the set of types indicated by the typing for F.*

To reach out goal, we can use this property in the following way. Let $\mathcal{F}$ be a set of functions. By applying fusion to the body of each function $F \in \mathcal{F}$ we eliminate the data constructors of all data types that do not appear in the typing for F In particular, if F is an instance of a generic function on a user defined data type, then fusion will remove all data constructors of the base types $\{\rightleftarrows, \mathbb{1}, \times, +\}$, provided that neither the generic type of the function nor the instance type itself contains any of these base types.

5 Standard Fusion

Without any precautions the process of repeatedly eliminating consumer producer pairs might not terminate To avoid non-termination we will not reduce all possible pairs but restrict reduction to pairs in which only *proper* consumers and producers are involved. In [3] a separate analysis phase is used to determine proper consumers and producers. The following definitions are more or less directly taken from [3].

Definition 5.1 (Active Parameter). *The notions of* active occurrence *and* active parameter *are defined by simultaneous induction.*

- *We say that a variable x occurs actively in a (body) expression B if B contains a subexpression E such that*
  - $E =$ case x of $\ldots$, *or*
  - $E = x\ \ldots$, *or*
  - $E = F\,\vec{D}$, *such that $D_i = x$ and $act(F)_i$.*

  By $\mathrm{AV}(E)$ *we denote the set of active variables occurring in E.*
- *Let $F\,\vec{x} = B_F$ be a function. F is active in x_i (notation $act(F)_i$) if $x_i \in$* $\mathrm{AV}(B_F)$.

The notion of *accumulating parameter* is used to detect potentially growing recursion.

Definition 5.2 (Accumulating Parameter). *Let $F_1 = B_1, \ldots, F_n = B_n$ be a set of mutually recursive functions. The function $F = F_j$ is* accumulating *in its i^{th} parameter (notation $acc(F)_i$) if either*

- *there exists a right-hand side B_k containing a subexpression $F\vec{D}$ such that D_i is open but not just a variable.*
- *B_j itself contains a subexpression $F_k\vec{D}$ such that F_k is accumulating in l, and $D_l = x_i$.*

Observe that the active as well as the accumulating predicate are defined recursively. This will amount to solving a least fixed point equation.

Definition 5.3 (Proper Consumer). *A function F is a* proper consumer *in its i^{th} parameter (notation $con(F)_i$) if $act(F)_i$ and $\neg acc(F)_i$.*

Definition 5.4 (Proper Producer). *Let $F_1 = B_1, \ldots, F_n = B_n$ be a set of mutually recursive functions.*

- *A body B_k is called* unsafe *if it contains a subexpression $G\vec{E}$, such that $con(G)_i$ and $E_i = F_j(\cdots)$, for some G, j. In words: B_k contains a call to F_j on a consuming position.*
- *All functions F_k are* proper producers *if none of their right-hand sides is unsafe.*

Consider, for example, the function for reversing the elements of a list. It can be defined in two different ways. In the first definition (rev1) an auxiliary function revacc is used; the second definition (rev2) uses app.

```
rev1 l       = revacc l Nil
revacc l a   = case l of  Nil → a;  Cons x xs → revacc xs (Cons x a)
rev2 l       = case l of Nil → Nil;  Cons x xs → app (rev2 xs) (Cons x Nil)
```

Both rev1 and revacc are proper producers; rev2 however is not: since app is consuming in its first argument, the recursive occurrence of rev2 is on a consuming position. Consequently a function like foo l = len (rev1 l) will be transformed, whereas bar l = len (rev2 l) will remain untouched. By the way, the effect of the transformation w.r.t. the gain in efficiency is negligible.

6 Improved Consumer Analysis

If functions are not too complex, standard fusion will produce good results. In particular, this also holds for many generic functions. However, in some cases the fusion algorithm fails due to both consumer and producer limitations.

We will first examine what can go wrong with the current consumer analysis. For this reason we have adjusted the definition of app slightly.

$$\begin{array}{ll} \mathsf{app}\ l\ t & = \mathsf{case}\ l\ \mathsf{of}\ \mathsf{Nil} \to t;\ \mathsf{Cons}\ x\ xs \to \mathsf{app2}\ (\mathsf{Pair}\ x\ xs)\ t \\ \mathsf{app2}\ p\ t & = \mathsf{case}\ p\ \mathsf{of}\ \mathsf{Pair}\ x\ xs \to \mathsf{Cons}\ x\ (\mathsf{app}\ xs\ t) \end{array}$$

Due to the intermediate Pair constructor the function app is no longer a proper consumer. (The (indirect) recursive call has this pair as an argument and the non-accumulating requirement prohibits this.) It is hard to imagine that a normal programmer will ever write such a function directly. However, keep in mind that the optimization algorithm, when applied to a generic function, introduces many intermediate functions that communicate with each other via basic sum and product constructors. For exactly this reason many relatively simple generic functions cannot be optimized fully. One might think that a simple inlining mechanism should be capable of removing the Pair constructor. In general, such 'append-like' functions will appear as an intermediate result of the fusion process. Hence, this inlining should be combined with fusion itself which makes it much more problematic. Experiments with very simple inlining show that it is practically impossible to avoid non-termination for the combined algorithm.

To solve the problem illustrated above, we extend fusion with *depth analysis*. Depth analysis is a refinement of the accumulation check (definition 5.2). The original accumulation check is based on a purely syntactic criterion. The improved accumulation check takes into account how the size of the result of a function application increases or decreases with respect to each argument. The idea is to count how many times constructors and destructors (pattern matches) are applied to each argument of a function. If this does not lead to an 'infinite' depth (an infinite depth is obtained if a recursive call extends the argument with one or more constructors) accumulation of constructors is harmless.

Definition 6.1 (Depth). *The functions occ and dep are specified below by simultaneous induction (again leading to a fixed point construction).*

$$\begin{array}{lll} occ(v,x) & = 0, & \text{if } v = x \\ & = \bot, & \text{otherwise} \\ occ(v, C\,\vec{E}) & = \max_i(1 + occ(v,E_i)) & \\ occ(v, F\,\vec{E}) & = \max_i(dep(F)_i + occ(v,E_i)) & \\ occ(v, x\ \vec{E}) & = \max(occ(v,x), \max_i(occ(v,E_i))) & \\ \multicolumn{3}{l}{occ(v, \mathsf{case}\ x\ \mathit{of}\ \ldots C_i\vec{y} \to E_i \ldots)} \\ & = \max(-\infty, \max_i(\max(occ(v,E_i), & \\ & \qquad \max_k(occ(y_k,E_i)) - 1))), & \text{if } v = x \\ & = \max_i(occ(v,E_i)), & \text{otherwise} \\ dep(F)_i & = occ(x_i, B_F), & \text{where } F\,\vec{x} = B_F \end{array}$$

using $\bot + x = \bot$, $\max() = \bot$, *and* $(-\infty) + (+\infty) = +\infty$.

The depths of app and app2 above are $dep(\mathsf{app}) = dep(\mathsf{app2}) = (0, +\infty)$.

The following definition gives an improved version of the accumulation property (definition 5.2).

Definition 6.2 (Accumulating With Depth). *Let $F_1 = B_1, \ldots, F_n = B_n$ be a set of mutually recursive functions. The function $F = F_j$ is* accumulating *in its i^{th} parameter (notation $acc(F)_i$) if either*

1. *$dep(F)_i = +\infty$, or*
2. *there exist a right-hand side B_k containing a subexpression $F\vec{D}$ such that $\mathrm{AV}(D_i) \neq \emptyset$, and D_i is not just a variable, or*
3. *B_j has a subexpression $F_k\vec{D}$ such that F_k is accumulating in l, and $D_l = x_i$.*

We illustrate this definition with an example in which a function ident is defined that collects all subsequent characters from an input list that obey a predicate p.

```
ident p i r = case  i  of  Nil → r
                           Cons h t → if (p h) (ident p t (app r (Cons h Nil))) r
```

This function is accumulating in its third parameter because of requirement (2). Note that the depth of ident in r is 0. Indicating this argument as accumulating will prevent infinite fusion of ident, e.g. in the body of ident itself.

7 Improved Producer Analysis

In some cases the producer classification (definition 5.4) is responsible for not getting optimal transformation results. The problem occurs, for instance, when the type of a generic function contains recursive type constructors. Take, for example, the monadic mapping function for the list monad mapl. The base type of mapl is $a \to$ List b. Recall that the specialization of mapl to any data type, e.g. Tree, will use the embedding-projection specialized to MapL (see section 2). This embedding projection is based on $\mathsf{ep}_{\mathsf{List}}$, the generic embedding projection specialized to lists. Since List is recursive, $\mathsf{ep}_{\mathsf{List}}$ is recursive as well. Moreover, one can easily show the recursive call to $\mathsf{ep}_{\mathsf{List}}$ appears on a consuming position, and hence $\mathsf{ep}_{\mathsf{List}}$ is not a proper producer. As a consequence, the transformation of a specialized version of mapl gets stuck when it hits on $\mathsf{ep}_{\mathsf{List}}$ appearing as a producer. We illustrate the essence of the problem with a much simpler example:

```
data Id a = Id a
unId i    = case i of Id x → x
foo       = Id (unId foo)
bar       = unId foo
```

Obviously, the function unId is consuming in its argument. Since the recursive call to foo appears as an argument of unId, this function foo is an improper producer. Consequently, the right-hand side of bar cannot be optimized. On the other hand, it seems to be harmless to ignore the producer requirement in this case and to perform a fusion step. To avoid non-termination in general, we use our special tree representation of new function symbols. Remember that these symbol trees contain the information of how the corresponding function was created in terms of the initial set functions and data constructors. Suppose we have a fusion pair consisting of a proper consumer and an *improper* producer. The combined symbol tree is used as follows: if it contains a path on which an improper producer appears twice, we don't fuse; otherwise a fusion step is performed. In essence this producer check avoids repetitive unfolding of the same improper producer. For a full formalization of this idea we refer to [1].

To illustrate the effect of our refinement we go back to the example. Now, the application in the body of bar is a redex (i.e. a fusion candidate). It will be replaced by $\mathsf{unId}_1\mathsf{foo}$, and a new function for this symbol is generated with unId foo as initial body. Indeed, this expression is identical to the expression from which it descended. Again the expression will be recognized as a redex and replaced by $\mathsf{unId}_1\mathsf{foo}$, finishing the fusion process.

Since we no longer fuse *all* consumer-producers pairs but restrict ourselves to *proper* consumers and producers we cannot expect that the result of a fused expression will always be in *FNF*(as defined in definition 4.1). To show that such a result does not contain any unwanted data constructors, we first give a more liberal classification of the fusion results. Remember that our main concern is not to eliminate *all* consumer–producer pairs, but only those communicating intermediate objects caused by the structural representation of data types. The new notion of fusion normal forms is based on the types of functions and data constructors.

Definition 7.1 (T-Free Forms). *Let T be a type constructor.*

- *Let S be a function or data constructor, say with arity n, and type $\vec{\sigma} \to \tau$, where $|\vec{\sigma}| = n$. We say that a k-ary version of S* excludes *T, $k \le n$, (notation $S \not\supseteq_k T$) if $CT(\sigma_{k+1}, \ldots, \sigma_n, \tau) \cap CT(T) = \emptyset$. We abbreviate $S \not\supseteq_n T$ to $S \not\supseteq T$.*
- *The set N_T of expressions in T-free form is defined as:*

$$N_T ::= N'_T \mid F\,\vec{N'_T} \mid C\,\vec{N_T} \qquad N'_T ::= v \mid v\,\vec{N_T} \mid S\,\vec{N'_T}$$

with the restriction that for each application of $S\,\vec{N'_T}$ it holds that $S \not\supseteq_{|\vec{N'_T}|} T$.

For functions in *T-FF* we have a property similar to property 4.2 of functions in *FNF*.

Property 7.2. *Let T be type constructor, and $\mathcal{F}$ be a collection of functions in T-FF. Then, for any $F \in \mathcal{F}$ we have $F \not\supseteq T \;\Rightarrow\; CE(F) \cap CT(T) = \emptyset$.*

8 Fusion of Generic Instances

In this section we deal with the optimization of instances generated by the generic specializer. An instance is considered to be optimized if the resulting code does not contain constructors belonging to the basic types $\{\rightleftarrows, \mathbb{1}, \times, +\}$. Our goal is to illustrate that under certain conditions on the generic base cases, the generic function types, and the instance types the presented fusion algorithm completely removes generic overhead. According to property 7.2 this boils down to showing that fusion leads to $\{\rightleftarrows, \mathbb{1}, \times, +\}$-*FF*.

As mentioned in section 2, an instance of a generic function consists of an adaptor and the code for the structural representation. Treating a generic function completely would require several pages of example code. For this reason we restrict ourselves to the adapter part of a generic function and illustrate how the EPs are eliminated. The first example shows how the to projection of EP is fused with a recursive instance of ep for the List type. We assume that the instance on lists $\mathsf{ep}_{\mathsf{List}}$ is already fused and is in $\{+, \times, \mathbb{1}\}$-*FF*.

```
ep_List f            = EP (epto_List f (ep_List f)) (epfrom_List f (ep_List f))
epto_List f r l      = case l of Nil      → Nil
                                 Cons h t → Cons (to f h) (to r t)
epfrom_List f r l    = case l of Nil      → Nil
                                 Cons h t → Cons (from f h) (from r t)
```

Consider the application to $(\mathsf{ep}_{\mathsf{List}}\ f)$. Fusion will introduce a function $\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}$ (we indicate new symbols by underlining the corresponding consumer and producer, and also leave out the argument number and the actual arity of the producer). The body of this function is optimized as follows:

```
$\underline{\mathsf{to\ ep_{List}}}\ f\ l$
  ⇝ to (EP (epto_List f (ep_List f)) (...)) l              unfolding ep_List
  ⇝ epto_List f (ep_List f) l                              unfolding to
  ⇝ case l of Nil      → Nil                               unfold epto_List
              Cons h t → Cons (to f h) (to (ep_List f) t)
  ⇝ case l of Nil      → Nil                               folding to, ep_List
              Cons h t → Cons (to f h) ($\underline{\mathsf{to\ ep_{List}}}$ f t)
```

Obviously, the resulting code is in $\rightleftarrows$-*FF*, and due to property 7.2 this function will not generate any EP-constructor.

The next example shows how the recursive instances of ep for lists and trees are fused together. The instance for tree after fusion is

```
ep_Tree f            = EP (epto_Tree f (ep_Tree f)) (epfrom_Tree f (ep_Tree f))
epto_Tree f r t      = case t of Leaf x       → Leaf (to f x)
                                 Branch x y   → Branch (to r x) (to r y)
epfrom_Tree f r t    = case t of Leaf x       → Leaf (from f x)
                                 Branch x y   → Branch (from r x) (from r y)
```

Fusion of $\mathsf{ep}_{\mathsf{Tree}}\ (\mathsf{ep}_{\mathsf{List}}\ f)$ proceeds as follows.

```
$\underline{\mathsf{ep_{Tree}\ ep_{List}}}\ f$
  ⇝ EP (epto_Tree (ep_List f) (ep_Tree (ep_List f))) (...)  unfolding ep_Tree
  ⇝ EP (epto_Tree (ep_List f) ($\underline{\mathsf{ep_{Tree}\ ep_{List}}}$ f)) (...)  folding ep_Tree, ep_List
  ⇝ EP ($\underline{\mathsf{epto_{Tree}\ ep_{List}}}$ f ($\underline{\mathsf{ep_{Tree}\ ep_{List}}}$ f)) (...)  fusing epto_Tree, ep_List
```

where fusion of $\mathsf{epto}_{\mathsf{Tree}}$ $(\mathsf{ep}_{\mathsf{List}}\ f)$ proceeds as

$$\begin{array}{lllll}
\multicolumn{5}{l}{\underline{\mathsf{epto}_{\mathsf{Tree}}\ \mathsf{ep}_{\mathsf{List}}}\ f\ r\ l} \\
\leadsto \mathsf{case}\ t\ \mathsf{of} & \mathsf{Leaf}\ x & \to \mathsf{Leaf}\ (\mathsf{to}\ (\mathsf{ep}_{\mathsf{Tree}}\ f)\ x) & & \textit{unfolding } \mathit{epto}_{\mathit{Tree}} \\
 & \mathsf{Branch}\ x\ y & \to \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) & & \\
\leadsto \mathsf{case}\ t\ \mathsf{of} & \mathsf{Leaf}\ x & \to \mathsf{Leaf}\ (\underline{\mathsf{to}\ \mathsf{ep}_{\mathsf{List}}}\ f\ x) & & \textit{folding } \mathit{to}, \mathit{ep}_{\mathit{List}} \\
 & \mathsf{Branch}\ x\ y & \to \mathsf{Branch}\ (\mathsf{to}\ r\ x)\ (\mathsf{to}\ r\ y) & &
\end{array}$$

Fusion has eliminated an intermediate EP produced by $\mathsf{ep}_{\mathsf{List}}$ and consumed by $\mathsf{ep}_{\mathsf{Tree}}$ from the original expression $\mathsf{ep}_{\mathsf{Tree}}$ $(\mathsf{ep}_{\mathsf{List}}\ f)$. This has led to a function whose structure is similar to the structure of $\mathsf{ep}_{\mathsf{List}}$ or $\mathsf{ep}_{\mathsf{Tree}}$. Therefor fusing the projection to with this function will yield a $\rightleftarrows$-*FF*.

It appears that not all adaptors can be fused to $\rightleftarrows$-*FF*. In particular, this occurs when the generic type contains type constructors with a *nested* or with a *contra-variant* definition. Nested types are types like

$$\textbf{data}\ \mathsf{Nest}\ a = \mathsf{NNil} \mid \mathsf{NCons}\ a\ (\mathsf{Nest}\ (a, a))$$

i.e. recursive types in which the arguments of the recursive occurrence(s) are not just variables. These nested type definitions give rise to accumulating eps, and hence to improper consumers (see section 6). Contra-variantly recursive types are types like

$$\textbf{data}\ \ \mathsf{Contra} = \mathsf{Contra}\ (\mathsf{Contra} \to \mathrm{Int})$$

i.e. recursive types in which a recursive occurrence appears on a contra-variant position (the first argument of the $\to$-constructor). For these contra-variant type definitions the improved producer check will fail (see section 7), obstructing further fusion.

Apart from these type constructor requirements, we have the additional restriction that the type of the generic function is free of *self-application*, e.g. List (List a). For self application again the producer check will fail. As an example, consider a generic non-deterministic parser with type.

$$\textbf{type}\ \ \mathsf{Parser}\ a = (\mathsf{List}\ \mathrm{Char}) \to \mathsf{List}\ (a, \mathsf{List}\ \mathrm{Char})$$

Here, the nested List application will hamper complete fusion. This problem can be avoided by choosing different types for different results: The input stream, of type List Char, and the resulting parses, of type List $(a, \mathsf{List}\ \mathrm{Char})$ could alternatively be represented by using an auxiliary list type List'.

9 Performance Evaluation

We have implemented the improved fusion algorithm as a source-to-source translator for the core language presented in section 3. The input language has been extended with syntactical constructs for specifying generic functions. These generic functions are translated into ordinary Clean functions, optimized as well as unoptimized. Then we used the Clean compiler to evaluate the performance of the optimized and unoptimized code. We have investigated many example programs, but in this section we will only present the result of examples that are realistic and/or illustrative: Simple mapping (map), monadic mapping (mapl, mapr, for the List and Rose monad respectively), generic reduce (reduce) (used to implement folding), and non-deterministic parsing (parser). We only mention execution times; heap usage is not shown. The latter because in all example programs memory consumption is improved with approximately the same factor as execution time. Moreover, all optimized programs are more or less as efficient as their handwritten counterparts. Finally, to illustrate the difference between successful and partial (unsuccessful) fusion we include the performance results of a monadic mapping function (mapn) with a nested monad type.

program	*unoptimized (sec)*	*optimized (sec)*	*speedup (times)*
map	72.9	13.4	7.9
mapl	29.8	1.5	19.9
mapr	304.5	10.9	82.3
reduce	37.9	3.7	10.2
parser	45.65	0.51	89.5
mapn	168.4	164	1.03

These figures might appear too optimistic, but other experiments with a complete generic XML-parser confirm that these results are not exaggerated.

10 Related Work

The present work is based on the earlier work [2] that used partial evaluation to optimize generic programs. To avoid non-termination we used fix-point abstraction of recursion in generic instances. This algorithm was, therefore, specifically tailored for optimization of generic programs. The algorithm presented here has also been designed with optimization of generic programs in mind. However it is a general-purpose algorithm that can improve other programs. The present algorithm completely removes generic overhead from a considerably larger class of generic programs than [2].

The present optimization algorithm is an improvement of fusion algorithm [3], which is in turn based on Chin's fusion [4] and Wadler's deforestation [9]. We have improved both consumer and producer analyses to be more semantically than syntactically based. Chin and Khoo [5] improve the consumer analysis using the *depth* of a variable in a term. In their algorithm, *depth* is only defined for constructor terms, i.e. terms that are only built from variables and constructor applications. This approach is limited to first order functions. Moreover, the functions must be represented in a special *constructor-based* form. In contrast, our *depth* is defined for arbitrary terms of our language. Our algorithm does not require functions in to be represented in a special form, and it can handle higher order functions.

The present paper uses a generic scheme based on type-indexed values [6]. However, we believe that our algorithm will also improve code generated by other generic schemes, e.g PolyP [8].

11 Conclusions and Future Work

In this paper we have presented an improved fusion algorithm, in which both producer and consumer analyses have been refined. We have shown how this algorithm eliminates generic overhead for a large class of programs. This class is described; it covers many practical examples. Performance figures show that the optimization leads to a huge improvement in both speed and memory usage.

In this paper we have ignored the aspect of data sharing. Generic specialization does not generate code that involves sharing, although sharing can occur in the base cases provided by the programmer. A general purpose optimization algorithm should take sharing into account to avoid duplication of work and code bloat. In the future we would like to extend the algorithm to take care of sharing. Additionally, we want to investigate other applications of this algorithm than generic programs. For instance, many programs are written in a combinatorial style using monadic or arrow combinators. Such combinators normally store functions in simple data types, i.e. wrap functions. To actually apply a function they need to unwrap it. It is worth looking at elimination of the overhead of wrapping-unwrapping.

References

1. Artem Alimarine and Sjaak Smetsers. Fusing generic functions. Technical report NIII-R0434, University of Nijmegen, Sep 2004.
2. Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexted Kozen, editor, *Mathematics of Program Construction*, number 3125 in LNCS, pages 16–31, Stirling, Scotland, UK, July 2004. Springer.
3. Diederik van Arkel, John van Groningen, and Sjaak Smetsers. Fusion in practice. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 51–67. Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Springer, 2003.
4. Wei-Ngan Chin. Safe fusion of functional expressions II: further improvements. *Journal of Functional Programming*, **4**(4):515–555, October 1994.
5. Wei-Ngan Chin and Siau-Cheng Khoo. Better consumers for program specializations. *Journal of Functional and Logic Programming*, 1996(4), November 1996.
6. Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universität Bonn, October 2000.
7. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.
8. P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *The 24th ACM Symposium on Principles of Programming Languages, POPL '97*, pages 470–482. ACM Press, 1997.
9. Phil Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, number 300 in LNCS, pages 344–358, Berlin, Germany, March 1988. Springer-Verlag.

The Program Inverter LRinv and Its Structure

Masahiko Kawabe[1,⋆] and Robert Glück[2]

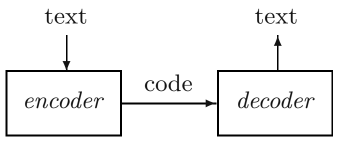
[1] Waseda University, Graduate School of Science and Engineering
Tokyo 169-8555, Japan
kawabe@suou.waseda.jp

[2] University of Copenhagen, DIKU, Dept. of Computer Science
DK-2100 Copenhagen, Denmark
glueck@acm.org

Abstract. Program inversion is a fundamental concept in program transformation. We describe the principles behind an automatic program inverter, which we developed for a first-order functional language, and show several inverse programs automatically produced by our system. The examples belong to different application areas, including encoding and decoding, printing and parsing, and bidirectional data conversion. The core of the system uses a stack-based language, local inversion, and eliminates nondeterminism by applying methods from parsing theory.

1 Introduction

The purpose of this paper is to describe a method for *automatic program inversion*, which we developed [6, 7] for a first-order functional language, and to show several inverse programs produced by our implementation. Many computational problems are inverse to each other, even though we often do not consider them from this perspective. Perhaps the most familiar example is the encoding and decoding:



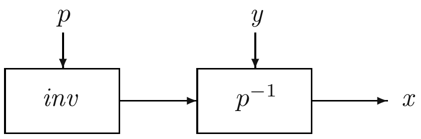
Two inverse programs

Here, a text is translated into a code by an *encoder* and then translated back into the original text by a *decoder*. Usually, both programs are written by hand with the common problem of ensuring their correctness and maintaining their consistency. But why write both programs if one can be derived from the other by a program inverter? Beside saving time and effort, the inverse program is correct by construction and, if the original program is changed for any reason, it is easy to produce a new inverse program.

⋆ This research is supported by JSPS.

One of the main challenges of program inversion is to derive a *deterministic inverse program* from an injective source program. In [7], we observed that this problem is similar to the construction of a *deterministic parser* if we view the inverse program as a context-free grammar where the traces of the program constitute its language. Determining which branch to follow in a nondeterministic inverse program is like determining which production rule to apply when constructing a parse tree. Our method makes use of this correspondence.

A *program inverter* is a program transformer, similar to a translator, but instead of producing a target program that is functionally equivalent to the source program, the source program p is transformed into a program p^{-1} that is *inverse* to p. Inversion is a fundamental concept in mathematics, science, and engineering, but has found little attention in computer science, even though many algorithmic problems are inverse to each other, such as printing and parsing, compilation and decompilation, and encoding and decoding.



Program inverter: generating an inverse program p^{-1}

In this paper, we introduce the main construction principles for building automatic program inverters, which we identified and applied in our latest system LRinv: (i) the design of an 'atomic language', (ii) local inversion of a program, and, if possible, (iii) the elimination of nondeterminism from an inverse program. Our present system[1] is based on LR(0) parsing and represents the first complete implementation of such an automatic program inverter. We use this system for inverting three non-trivial examples. The examples that we selected belong to different application areas, including encoding and decoding of data, parsing and printing syntax trees, and the bidirectional conversion of data. The examples show the need for our previously proposed parsing-based technique to eliminate nondeterminism.

There are various choices when designing an automatic program inverter, and especially when eliminating nondeterminism. This also means that they can differ substantially in their inversion power and the quality of the inverse programs they produce. The most promising approach that we identified [7] for eliminating nondeterminism is to view a program as a context-free grammar and to apply to it methods from parsing theory. We used left-factoring and a restricted form of LL(1) parsing in [6], and a method from LR(0) parsing in in [7]. We hope that our presentation will elucidate the main principles for building program inverters and show their design space.

While interpreters for logic programming languages, such as Prolog, can be used for *inverse interpretation* (to search for inputs for a given output), we are interested in *program inversion*. Our goal is to generate stand-alone *inverse*

[1] Home page of LRinv `http://www.diku.dk/topps/inv/lr/`

programs: given a program p, we want to obtain a program p^{-1}. A good reason for using program inversion instead of inverse interpretation is that, in general, inverse programs are much more efficient than inverse interpreters in computing an input x for a given output y. Our focus is on *injective programs* and the generation of *deterministic inverse programs*. For more details regarding the notions of inverse interpretation and program inversion, see references [1, 8].

After introducing the main concepts (Sect. 2), we present three construction principles for building a program inverter (Sect. 3), and show inverse programs produced by our system (Sect. 4). We next discuss related work (Sect. 5) and finally make a conclusion (Sect. 6).

2 Fundamental Concepts

We summarize the main concepts of program inversion. We consider only programs that associate, at most, only one output to every input (injective partial functions); we do not consider relations with multiple solutions. A program p is *injective* iff, whenever p terminates on input x_1 and on input x_2, and produces the same output in both cases, $[\![p]\!]\, x_1 = [\![p]\!]\, x_2$, it must be the case that $x_1 = x_2$. A program *inv* is a *program inverter*, iff for all injective programs p and for all values of x and y,

$$p^{-1} = [\![inv]\!]\, p \quad \text{and} \quad [\![p^{-1}]\!]\, y = x \Longleftrightarrow [\![p]\!]\, x = y \ .$$

Programs subjected to program inversion need not be total, but we assume they are injective. The inverse of an injective program is also injective. The following table illustrates the correspondence between injectiveness, functions and relations:

source program				inverse program
injective function	$1:1$	$\Longleftrightarrow$	$1:1$	injective function
function	$n:1$	$\Longleftrightarrow$	$1:n$	relation

A program inverter allows us to perform inverse computation in two stages. In the first stage, the source program p is transformed into an inverse program p^{-1} by the program inverter *inv*. In the second stage, the inverse program p^{-1} is evaluated with the given output y to obtain the original input x. For every programming language, there exists a (trivial) program inverter that produces a (trivial) inverse program for each source program. If we disregard the efficiency of inverse programs, a trivial inverse program can be implemented using a generate-and-test approach, first suggested by McCarthy [13], that searches for the original input x given the output y. However, our goal is to generate more efficient inverse programs.

3 Construction Principles: "How to Build an Inverter"

We now turn to the basic methods for constructing program inverters. Our aim is to develop a program inverter for a first-order functional language. Automatic inversion of a large class of programs is our main goal. We will use a constructor-based, first-order functional programming language as our source language. The language has a conventional call-by-value semantics and, for reasons of symmetry, is extended to multiple return values: functions may return multiple values and let-expressions can bind them to variables. We can think of input/output values as n-ary tuples. We omit the formal definition of the functional language due to the limited space (example programs can be seen in Figs. 3–5).

Construction Principles. Our approach to program inversion is based on *backward reading* programs, *i.e.*, in order to achieve global inversion of a program we locally invert each of its components. This idea is not new and can be found in [3, 9], but there, it is assumed that all programs are annotated (by hand) with suitable postconditions, while our goal is to achieve automatic inversion. Other approaches to program inversion exist, but most of them require human insight and understanding (*e.g.*, when using The Converse of a Function Theorem [2, 14]).

We have identified three main points in the design of an automatic program inverter. We assume that all functional programs that we consider are injective and *well-formed for inversion* in the sense that they contain no dead variables because they represent the deletion of values which, in an inverse program, need to be guessed if no other information is available.

1. **Atomization**: Given a source language, design an 'atomic language' in which each basic construct t has an inverse t^{-1} that is again a basic construct in the atomic language and where the compositional constructs can be inverted by inverting each of its components. In addition, the source language should preferably be easy to translate to and from this new language. The language which we designed is called *grammar language*.
2. **Local inversion**: Once the atomic language is designed, a program is inverted by inverting each atomic construct and their composition in the program. The result is a correct inverse program which, in many cases, is nondeterministic. This means that, at some given choice point during computation, there may be two or more alternatives that are valid. This is not a deficiency of the transformation, but rather, the nature of program inversion when applied to unidirectional programs.
3. **Elimination of nondeterminism**. Local inversion often produces a nondeterministic inverse program. Since it is undesirable in practice to discover the evaluation path only by trial and error, it is important to turn nondeterministic programs into deterministic ones. The aim of this stage is to produce a deterministic inverse program. Also, the elimination of nondeterminism is necessary in order to translate inverse programs back into a deterministic language, such as a functional language. This stage, however, is a heuristic stage since, in general, there is no algorithm that can eliminate nondeterminism from any inverse program (*cf.* there does not exist a deterministic parser for all context-free grammars and the fact that determinacy is an asymmetric property of a grammar [12]).

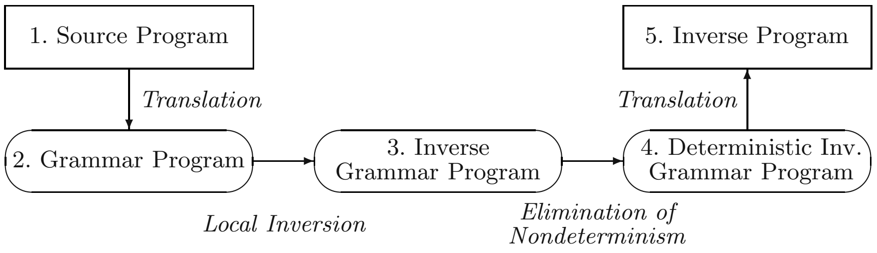

1. Source Program
Translation
2. Grammar Program
3. Inverse Grammar Program
4. Deterministic Inv. Grammar Program
Translation
5. Inverse Program
Local Inversion
Elimination of Nondeterminism

Fig. 1. Structure of the program inverter

The structure of a program inverter based on these principles is outlined in Fig. 1: first, a source program is translated into the grammar language, then local inversion is performed and, if possible, nondeterminism is eliminated. Finally, the program is translated back into the original language. We shall now explain the transformation steps with an example program that increments binary numbers.

Grammar Language. Before applying local inversion to a functional program, we translate it to the *grammar language*, a stack-based language. This is our 'atomization' of a functional program. Each operation in the grammar language operates on a stack: it takes its arguments from the stack and pushes the results onto the stack. A value v is a constructor c with zero or more values as arguments: $v ::= c(v_1, ..., v_n)$. It is not difficult to translate a first-order functional program well-formed for inversion to and from the grammar language[2].

$p ::= d_1 \dots d_m$	(program)	$a ::= c\textbf{!}$		(constructor appl.)
$d ::= f \to t_1 \dots t_m$	(definition)	$\mid$	$c\textbf{?}$	(pattern matching)
$t ::= a$	(atomic operation)	$\mid$	$\lfloor\_\rfloor$	(duplication/equality)
$\mid f$	(function call)	$\mid$	π	(permutation)

As an example, consider a program that increments a binary number by 1. The functional program *inc* and its inverse inc^{-1} are shown in Fig. 2 (1, 5)[3]. For example, $inc(11) = 100$ and $inc^{-1}(100) = 11$. We require that binary numbers have no leading zeros. In our implementation, a binary number x is represented by an improper list containing x's digits in reversed order. Thus, a list representing a binary number always ends with 1. For instance, the binary number 100 is represented by (0 0 . 1). The result of translating *inc* into the grammar language is shown in Fig. 2 (2). A function in the grammar language is defined by one or more sequences of operations which are evaluated from left to right.

Constructor applications and pattern matchings, which are the main operations in the grammar language, work as follows. The application $c\textbf{!}$ of an n-ary

[2] Operator $\lfloor\_\rfloor$ has two functionalities: duplication of values, $\lfloor\langle v\rangle\rfloor = \langle v, v\rangle$, and equality testing, $\lfloor\langle v, v\rangle\rfloor = \langle v\rangle$ and $\lfloor\langle v, w\rangle\rfloor = \langle v, w\rangle$ if $v \neq w$. The operator is self-inverse [6]. Permutation π reorders the top elements of the stack. Not essential here.

[3] The enumeration in Fig. 1 corresponds to the parts in Fig. 2.

1. Source program:

```
$inc(x) \triangleq$ case $x$ of
              $1 \to (0{:}1)$
              $x_1{:}xs \to$ case $x_1$ of
                          $0 \to (1{:}xs)$
                          $1 \to$ let $(m){=}inc(xs)$ in $(0{:}m)$
```

2. Grammar program:

$inc \to 1\mathbf{?}\ 1\mathbf{!}\ 0\mathbf{!}\ \mathrm{Cons}\mathbf{!}$

$inc \to \mathrm{Cons}\mathbf{?}\ 0\mathbf{?}\ 1\mathbf{!}\ \mathrm{Cons}\mathbf{!}$

$inc \to \mathrm{Cons}\mathbf{?}\ 1\mathbf{?}\ inc\ 0\mathbf{!}\ \mathrm{Cons}\mathbf{!}$

3. Local inversion: (nondeterministic choices are marked by boxes)

$inc^{-1} \to \boxed{\mathrm{Cons}\mathbf{?}\ 0\mathbf{?}\ 1\mathbf{?}}\ 1\mathbf{!}$

$inc^{-1} \to \mathrm{Cons}\mathbf{?}\ 1\mathbf{?}\ 0\mathbf{!}\ \mathrm{Cons}\mathbf{!}$

$inc^{-1} \to \boxed{\mathrm{Cons}\mathbf{?}\ 0\mathbf{?}\ inc^{-1}}\ 1\mathbf{!}\ \mathrm{Cons}\mathbf{!}$

4. Elimination of nondeterminism:

$inc^{-1} \to f_0$	$f_1 \to 0\mathbf{?}\ f_2$	$f_2 \to 1\mathbf{?}\ 1\mathbf{!}$
$f_0 \to \mathrm{Cons}\mathbf{?}\ f_1$	$f_1 \to 1\mathbf{?}\ 0\mathbf{!}\ \mathrm{Cons}\mathbf{!}$	$f_2 \to \mathrm{Cons}\mathbf{?}\ f_1\ 1\mathbf{!}\ \mathrm{Cons}\mathbf{!}$

5. Inverse program:

```
$inc^{-1}(x_0) \triangleq$ case $x_0$ of  $x_1{:}x_2 \to$ let $(x_3){=}f_1(x_1,x_2)$ in $(x_3)$
$f_1(x_0,x_1) \triangleq$ case $x_0$ of
                $0 \to$ case $x_1$ of
                      $1 \to (1)$
                      $x_2{:}x_3 \to$ let $(x_4){=}f_1(x_2,x_3)$ in $(1{:}x_4)$
                $1 \to (0{:}x_1)$
```

Fig. 2. Complete example: the inversion of a program for incrementing a binary number

constructor c pops n values $v_1, \ldots, v_n$ from the stack and pushes a new value $c(v_1, \ldots, v_n)$ onto the stack. A pattern matching $c\mathbf{?}$ checks the topmost value v and, if $v = c(v_1, \ldots, v_n)$, pops v and pushes its components $v_1, \ldots, v_n$ onto the stack; otherwise it fails. For instance, when we evaluate $inc(1)$, the initial stack contains 1 (a nullary constructor). The pattern matching $1\mathbf{?}$ in the first sequence succeeds and removes 1 from the stack, while pattern matchings in other sequences fail. The constructor applications 1**!** and 0**!** in the first sequence yield a stack with 0 and 1, from which the final value (0 . 1) is constructed by Cons**!**.

When a function, like *inc*, is applied to a stack, all sequences defining the function are evaluated in parallel. When a pattern matching in a sequence fails, its evaluation is stopped (*e.g.*, when we have a value 0 and a pattern matching 1**?**). Since we consider only injective programs and their inverses, which are also injective, at most, only one branch will succeed in the end, even though several branches may be pursued in parallel. The computation is *deterministic* if, for any pair of the sequences defining a function, when comparing them from left to right, the first operations that are syntactically different are both pattern matchings. In other words, left-factoring that extracts the common part of two sequences leads to a choice point where all outgoing branches require pattern matching.

A grammar program need not be deterministic. Indeed, after local inversion, the computation of the inverse programs is generally nondeterministic. For instance, this is the case for program inc^{-1} in Fig. 2 (3, boxed operations): after evaluating the first two operations of the definitions of inc^{-1}, it is not always clear which branch should be chosen. Similarly, all grammar programs that are left-recursive are nondeterministic. In order to give meaning to such inverse programs, and to talk about their correctness, the semantics of the grammar language has to be defined to allow nondeterministic computations.

Local Inversion. In a grammar program, sequences of operations are easily inverted by reading their intended meaning backwards and replacing each operation by its inverse. Each function definition in a grammar program is inverted individually. A useful property of the grammar language is that each term t has exactly one term t^{-1} as its inverse: the inverse of a constructor application c**!** is pattern matching c**?**, and vice versa; the inverse of $\lfloor\_\rfloor$ is $\lfloor\_\rfloor$ (the operator is self-inverse [6]); the inverse of a permutation π is π^{-1}, and the inverse of a function call f is a call to f's inverse function f^{-1}.

Local inversion is formally defined as follows. Given a grammar program p, each of p's definitions is inverted separately:

$$INV[\![\ p\ ]\!] = \{\ Inv[\![\ d\ ]\!] \mid d \in p\,\}$$

For a definition d in a grammar program, the sequence of terms defining function f is reversed and each term is inverted individually:

$$Inv[\![\ f \rightarrow t_1 \ldots t_n\ ]\!] = f^{-1} \rightarrow inv[\![\ t_n\ ]\!] \ldots inv[\![\ t_1\ ]\!]$$

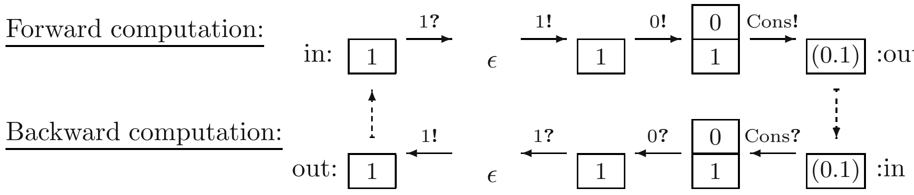
For each term t in a definition, there exists a one-to-one correspondence between t and its inverse:

$$inv[\![\ c\mathbf{!}\ ]\!] = c\mathbf{?} \qquad inv[\![\ f\ ]\!] = f^{-1} \qquad inv[\![\ \lfloor\_\rfloor\ ]\!] = \lfloor\_\rfloor$$
$$inv[\![\ c\mathbf{?}\ ]\!] = c\mathbf{!} \qquad inv[\![\ \pi\ ]\!] = \pi^{-1}$$

Local inversion does not perform unfold/fold and, thus, it terminates on all source programs. Inverse programs produced by local inversion are correct in the following sense: if there is a computation of a grammar program p with input stack vs_{in} that terminates and yields an output stack vs_{out}, then there is a computation of the inverse program p^{-1} with vs_{out} that terminates and yields the original input stack vs_{in}, and vice versa. The inverse programs that we obtain by local inversion are correct, but they are often nondeterministic and, thus, inefficient to compute. (If the inverse program resulting from local inversion is deterministic, then we are done with program inversion!)

Example Compare function *inc* before and after local inversion (Fig. 2). Each atomic operation is inverted according to the rules above, but function inc^{-1} is nondeterministic: the pattern matching **1?** in the first definition and the function call inc^{-1} in the third are evaluated at the same time, and at this point we cannot see which of them succeeds. Since it is undesirable to discover successful branches only by trial and error, it is important to turn nondeterministic programs into deterministic ones. We should stress that local inversion can invert grammar programs that are not injective. The result will be a nondeterministic program from which we cannot eliminate nondeterminism (otherwise, it would imply that the source program is injective).

The computation of $inc(1) = (0\,.\,1)$ and its inverse computation can be illustrated by the following computation sequences operating on a stack of values. We show the forward computation and the backward computation along the first branch of the program *inc* before and after inversion (Fig. 2 (2, 3)). This shows that we can compute the input and the output back and forth. Observe also the correspondence of the operations in forward and backward computation (c**?** $\leftrightarrow$ c**!**). Here, stack ϵ denotes the empty value stack.



Elimination of Nondeterminism. There are different approaches to eliminating nondeterminism from an inverse program. The most promising approach that we have identified so far is to view a program as a context-free grammar and to apply to it methods from parser construction. In [6], we used *left-factoring* and a restricted variant of *LL*(1) *parsing*, and in [7], we expanded it to methods from *LR*(0) *parsing*, which we shall also use in the current paper. The latter successfully inverts a large class of programs and is simpler than LR(k) methods since it does not require a lookahead.

To apply parsing methods to our problem domain, we regard programs as context-free grammars: an atomic operation corresponds to a terminal symbol, a function call to a nonterminal symbol, and the definition of a function to a grammar production. For example, we view the program in Fig. 2 as a context-free grammar where function symbol inc^{-1} corresponds to a nonterminal A, pattern matching Cons? to a terminal a, and so on.

$A \rightarrow abcd$	$inc^{-1} \rightarrow$ Cons? 0? 1? 1!
$A \rightarrow acef$	$inc^{-1} \rightarrow$ Cons? 1? 0! Cons!
$A \rightarrow abAdf$	$inc^{-1} \rightarrow$ Cons? 0? inc^{-1} 1! Cons!

Because of this close correspondence between programs and context-free grammars, we can operate with parsing methods on programs. The LR parsing methods are powerful deterministic bottom-up parsing methods. A principal drawback is that producing deterministic programs by hand can be tedious. One needs an automatic tool, a program inverter, to perform the transformation.

Intuitively, during transformation, the LR(0) method pursues all possibilities in parallel for as long as possible. A decision as to when to shift or reduce, and which production to use for reduction, will only take place when necessary. This corresponds to the subset construction when converting a nondeterministic finite automaton (NFA) into a deterministic finite automaton (DFA).

While grammar programs have a close correspondence to context-free grammars, and parsing theory is the main inspiration for eliminating nondeterminism, the two are different in that grammar programs operate on a *stack of values* while parsers for context-free grammars read a *sequence of tokens*. Another difference is that not all atomic operations in a grammar program are essential when making a nondeterministic choice deterministic. Not all atomic operations in the grammar language correspond directly to terminal symbols in a context-free grammar. Only *matching operations* can tell us whether to continue the computation along a branch. Moreover, we are interested in achieving a deep compilation of the LR item sets into grammar programs. Thus, in a program inverter, we cannot just use a conventional parser generator, such as `yacc`, but need to adapt the parsing techniques to the grammar language and develop a new way of generating programs from the collection of item sets.

Without going into further technical details—they can be found in our previous publication [7]—the result of transforming program inc^{-1} into a deterministic program is shown in Fig. 2 (4). We found that it is essential, at this stage of inversion, to deal with *left-recursive inverse programs* because they originate from right-recursive programs which in turn are the result of translating tail-recursive functional programs into grammar programs. For instance, left-factoring alone, as in [6], is not sufficient to eliminate nondeterministic choices in the presence of left-recursion. Without the possibility of dealing with left-recursive programs, we cannot invert tail-recursive programs, which is an important class of programs.

When generating an LR parser from a collection of conflict-free LR item sets, one usually produces a table- or procedure-driven, deterministic parser that operates on a stack of item sets as global data structure. In contrast to these classical parser generation methods, we want to achieve a deeper compilation of the collection of item sets. After computing a conflict-free collection of LR item sets, we want to generate a new grammar program, not a parser, and our ultimate goal is to translate the grammar program into a functional program.

Much like in parser generation, there is a trade-off between the number of item sets (which determines the size of the generated grammar programs) and the class of nondeterministic grammar programs for which deterministic programs can be constructed. In compiler construction, it was found that LALR(1) parsing, though not as powerful as LR(k) parsing, is sufficient for generating deterministic parsers for most practical programming languages, while keeping the size of the generated parsers to a manageable size. On the one hand, it is desirable to eliminate nondeterminism from a large class of important inverse programs; on the other hand, the size of programs generated by an LR parsing method is a concern. It is not entirely clear which variant of LR parsing has a reasonable trade-off in program inversion. In future work, we want to study LR(k) methods with lookahead in order to increase the class of invertible programs as well as the preprocessing of grammar programs for LL(k) methods.

4 LRinv: Examples of Automatic Inversion

This section shows several inverse programs that are automatically produced by our program inverter LRinv (see also the link in Footnote 1). The examples belong to three different application areas: bidirectional data conversion, encoding and decoding, and printing and parsing. The programs are written in a first-order functional language, which is the source language of our program inverter. The inverter is an implementation of the inversion methods described in Sect. 3; elimination of nondeterminism is based on LR(0) parsing. The system structure can be seen in Fig. 1.

4.1 Bidirectional Data Conversion

A familiar example is the conversion of data from one representation into another, and back. Here, we use the example of converting octal numbers into binary numbers. Clearly, the conversion is injective and we want to be able to convert numbers in both directions. We will see that we can write a program for one direction by hand, for instance, octal number to binary number, and then generate a program for the other direction automatically.

The function *octbin* is shown in Fig. 3: it converts an octal number into a binary number. Here, as in the example of incrementing binary numbers (Sect. 3), octal numbers are represented by reversed lists of octal digits ending with a non-zero digit. For example, we write the octal number 372 as (2 7 . 3) and convert it into a binary number by *octbin*((2 7 . 3)) = (0 1 0 1 1 1 1 . 1). Fig. 3 shows the inverse program $octbin^{-1}$, which was automatically produced by our system.

Local inversion, explained in Sect. 3, is sufficient to generate a deterministic inverse program. The reason can easily be understood by examining the branches of *octbin*: they have pairwise orthogonal output structures, namely (1), (0:1), ..., (1:1:1:m). Inverting these constructor applications into pattern matchings by local inversion is sufficient to obtain a deterministic inverse program.

Octal-to-binary converter (source program):

```
octbin(x) ≜
    case x of
    1 → (1); 2 → (0:1); 3 → (1:1); 4 → (0:0:1); 5 → (1:0:1); 6 → (0:1:1); 7 → (1:1:1)
    x₁:xs → let (m)=octbin(xs) in
                case x₁ of 0 → (0:0:0:m); 1 → (1:0:0:m); 2 → (0:1:0:m); 3 → (1:1:0:m)
                           4 → (0:0:1:m); 5 → (1:0:1:m); 6 → (0:1:1:m); 7 → (1:1:1:m)
```

Binary-to-octal converter (inverse program):

```
octbin⁻¹(x₀) ≜ case x₀ of 1 → (1)
                          x₁:x₂ → let (x₃)=f₃(x₁, x₂) in (x₃)
f₃(x₀, x₁) ≜
    case x₀ of
    0 → case x₁ of
        1 → (2)
        x₂:x₃ → case x₂ of
                0 → case x₃ of
                    1 → (4)
                    x₄:x₅ → case x₄ of
                            0 → case x₅ of 1 → (0:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (0:x₈)
                            1 → case x₅ of 1 → (4:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (4:x₈)
                1 → case x₃ of
                    1 → (6)
                    x₄:x₅ → case x₄ of
                            0 → case x₅ of 1 → (2:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (2:x₈)
                            1 → case x₅ of 1 → (6:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (6:x₈)
    1 → case x₁ of
        1 → (3)
        x₂:x₃ → case x₂ of
                0 → case x₃ of
                    1 → (5)
                    x₄:x₅ → case x₄ of
                            0 → case x₅ of 1 → (1:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (1:x₈)
                            1 → case x₅ of 1 → (5:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (5:x₈)
                1 → case x₃ of
                    1 → (7)
                    x₄:x₅ → case x₄ of
                            0 → case x₅ of 1 → (3:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (3:x₈)
                            1 → case x₅ of 1 → (7:1)
                                           x₆:x₇ → let (x₈)=f₃(x₆, x₇) in (7:x₈)
```

Fig. 3. Octal-to-binary converter and binary-to-octal converter

Binary tree encoder (source program):

```
    $treelist(t) \triangleq$ case $t$ of  $L \to (0{:}[\,])$
                                          $B(t_1, t_2) \to$ let $(r_1)=treelist(t_1)$ in
                                                            let $(r_2)=treelist(t_2)$ in
                                                                 let $(n, rs)=appendn(r_1, r_2)$ in $(n{:}rs)$
    $appendn(x, y) \triangleq$ case $x$ of  $[\,] \to (1, y)$
                                           $x_1{:}xs \to$ let $(n, z)=appendn(xs, y)$ in
                                                         let $(z_2)=id(x_1{:}z)$ in
                                                              let $(m)=inc(n)$ in $(m, z_2)$
    $id(x) \triangleq (x)$
```

Binary tree decoder (inverse program):

```
    $treelist^{-1}(x_0) \triangleq$ case $x_0$ of  $x_1{:}x_2 \to$ let $(x_3)=f_1(x_1, x_2)$ in $(x_3)$
    $f_1(x_0, x_1) \triangleq$
         case $x_0$ of
         $0 \to$ case $x_1$ of  $[\,] \to (L)$
         $1 \to$ case $x_1$ of  $x_2{:}x_3 \to$ let $(x_4)=f_1(x_2, x_3)$ in
                                          case $[\,]$ of  $x_5{:}x_6 \to$ let $(x_7)=f_1(x_5, x_6)$ in
                                                                     $(B(x_7, x_4))$
         $x_2{:}x_3 \to$ let $(x_4)=f_{21}(x_2, x_3)$ in
                       case $x_1$ of
                       $x_5{:}x_6 \to$ let $(x_7, x_8)=f_{17}(x_4, x_6, x_5)$ in
                                     case $x_8$ of
                                     $x_9{:}x_{10} \to$ let $(x_{11})=f_1(x_9, x_{10})$ in
                                                      case $x_7$ of
                                                      $x_{12}{:}x_{13} \to$ let $(x_{14})=f_1(x_{12}, x_{13})$ in
                                                                         $(B(x_{14}, x_{11}))$
    $f_{21}(x_0, x_1) \triangleq$ case $x_0$ of  $0 \to$ case $x_1$ of  $1 \to (1)$
                                                              $x_2{:}x_3 \to$ let $(x_4)=f_{21}(x_2, x_3)$ in $(1{:}x_4)$
                                             $1 \to (0{:}x_1)$
    $f_{17}(x_0, x_1, x_2) \triangleq$
         case $x_0$ of  $1 \to (x_2{:}[\,], x_1)$
                      $x_3{:}x_4 \to$ let $(x_5)=f_{21}(x_3, x_4)$ in
                                   case $x_1$ of  $x_6{:}x_7 \to$ let $(x_8, x_9)=f_{17}(x_5, x_7, x_6)$ in
                                                                  $(x_2{:}x_8, x_9)$
```

Fig. 4. Binary tree encoder and binary tree decoder

4.2 Encoding and Decoding

Another example is the lossless encoding of data. Here, we show the inversion of a program that encodes the structure of a binary tree as a list of binary numbers where each number indicates the number of nodes in the left subtree of a branch. The program is shown in Fig. 4. Auxiliary function *appendn* appends two lists and counts the length of the first list plus one. For instance, $appendn([A], [B, C]) = ((0\ .\ 1), [A, B, C])$. Again, numbers are binary numbers written in reversed order. An example is $B(B(L, B(L, L)), L)$ which represents the binary tree below (B – branch, L – leaf).

```
     B
    / \
   B   L
  / \
 L   B
    / \
   L   L
```

The result of encoding this binary tree is (recall that these are binary numbers with digits reversed – using decimal numbers we have as output $[6, 2, 0, 2, 0, 0, 0]$):

$$\mathit{treelist}(\mathrm{B}(\mathrm{B}(\mathrm{L}, \mathrm{B}(\mathrm{L}, \mathrm{L})), \mathrm{L})) = [(0\ 1\ .\ 1), (0\ .\ 1), 0, (0\ .\ 1), 0, 0, 0].$$

The program $\mathit{treelist}^{-1}$ that decodes such a list and returns the original binary tree is shown in Fig. 4. It was automatically produced by the program inverter using the source program in Fig. 4. Left-factoring of the inverse grammar program is sufficient to produce a deterministic program. Note that the source program uses the increment function *inc* defined in Fig. 2 and that the inverse program in Fig. 4 contains *inc*'s inverse (here, named f_{21}). In contrast to the previous example, not all the branches in the program *treelist* have different 'patterns'. This requires us to eliminate nondeterminism after local inversion.

4.3 Printing and Parsing

Another interesting application is printing and parsing. Here, printing means that we produce a list of tokens from a given abstract syntax tree. Parsing means that we read a list of tokens and create the corresponding abstract syntax tree. The operations are inverse to each other. It is usually easier to write a program that prints a list of tokens than to write a program that constructs an abstract syntax tree. We will illustrate this application of program inversion by a program that prints S-expressions, the data structure familiar from Lisp, which we will invert to obtain a *parser for S-expressions*.

The function *prnS* shown in Fig. 5 prints S-expression. We represent nil, cons and symbol by $[\,]$, $x{:}y$, and $\mathrm{S}(x)$, respectively. Output is a list of tokens which include symbols $\mathrm{S}(x)$, left and right parentheses L, R, and dots D. For example, $\mathit{prnS}(\mathrm{S}(a){:}[\mathrm{S}(b)]{:}\mathrm{S}(c){:}\mathrm{S}(d))$ yields a list $[\mathrm{L}, \mathrm{S}(a), \mathrm{L}, \mathrm{S}(b), \mathrm{R}, \mathrm{S}(c), \mathrm{D}, \mathrm{S}(d), \mathrm{R}]$ which corresponds to an S-expression $(a\ (b)\ c\ .\ d)$. The result of program inversion of *prnS* is shown in Fig. 5. Since we use methods of LR parsing to eliminate nondeterminism, the inverse program prnS^{-1} works like an LR parser for S-expressions: reading a token and calling another function corresponds to a shift action and returning from a function corresponds to a reduce action. As with the LR method used for parser generation, not only the power of the program inverter, but also the size of the generated inverse program is an important concern for the scalability of the approach. For example, it is desirable to increase the sharing of program code (*e.g.*, compare f_{12} and f_{30} in Fig. 5).

4.4 Other Applications

We have used the program inverter to invert a number of programs, including a program for run-length encoding, a small expression-to-bytecode compiler, a small printer of XML expressions, and a variety of miscellaneous functions such as naive reverse, fast reverse, and tailcons (also known as *snoc*). These are not shown here; some can be found in our previous publications [6, 7] and on the web site (Footnote 1).

S-expression printer (source program):

```
$prnS(s) \triangleq$ let $(t)=pcar([\,], s)$ in $(t)$
$pcar(t, s) \triangleq$ case $s$ of $S(x) \to (S(x){:}t)$
                        $[\,] \to (L{:}R{:}t)$
                        $x{:}y \to$ let $(t_2)=pcdr(t, y)$ in let $(t_3)=pcar(t_2, x)$ in $(L{:}t_3)$
$pcdr(t, s) \triangleq$ case $s$ of $S(x) \to (D{:}S(x){:}R{:}t)$
                        $[\,] \to (R{:}t)$
                        $x{:}y \to$ let $(t_2)=pcdr(t, y)$ in let $(t_3)=pcar(t_2, x)$ in $(t_3)$
```

S-expression parser (inverse program):

```
$prnS^{-1}(x_0) \triangleq$ case $x_0$ of $x_1{:}x_2 \to$ case $x_1$ of
                                  $S(x_3) \to$ case $x_2$ of $[\,] \to (S(x_3))$
                                  $L \to$ let $(x_3, x_4)=f_7(x_2)$ in case $x_3$ of $[\,] \to (x_4)$
$f_7(x_0) \triangleq$ case $x_0$ of
        $x_1{:}x_2 \to$ case $x_1$ of
                $R \to (x_2, [\,])$
                $S(x_3) \to$ let $(x_4, x_5)=f_{12}(x_2, S(x_3))$ in $(x_4, x_5)$
                $L \to$ let $(x_3, x_4)=f_7(x_2)$ in let $(x_5, x_6)=f_{12}(x_3, x_4)$ in $(x_5, x_6)$
$f_{12}(x_0, x_1) \triangleq$ case $x_0$ of
        $x_2{:}x_3 \to$ case $x_2$ of
                $D \to$ case $x_3$ of
                        $x_4{:}x_5 \to$ case $x_4$ of
                                $S(x_6) \to$ case $x_5$ of
                                        $x_7{:}x_8 \to$ case $x_7$ of $R \to (x_8, x_1{:}S(x_6))$
                $R \to (x_3, x_1{:}[\,])$
                $S(x_4) \to$ let $(x_5, x_6)=f_{30}(x_3, S(x_4))$ in $(x_5, x_1{:}x_6)$
                $L \to$ let $(x_4, x_5)=f_7(x_3)$ in let $(x_6, x_7)=f_{30}(x_4, x_5)$ in $(x_6, x_1{:}x_7)$
$f_{30}(x_0, x_1) \triangleq$ case $x_0$ of
        $x_2{:}x_3 \to$ case $x_2$ of
                $D \to$ case $x_3$ of
                        $x_4{:}x_5 \to$ case $x_4$ of
                                $S(x_6) \to$ case $x_5$ of
                                        $x_7{:}x_8 \to$ case $x_7$ of $R \to (x_8, x_1{:}S(x_6))$
                $R \to (x_3, x_1{:}[\,])$
                $S(x_4) \to$ let $(x_5, x_6)=f_{30}(x_3, S(x_4))$ in $(x_5, x_1{:}x_6)$
                $L \to$ let $(x_4, x_5)=f_7(x_3)$ in let $(x_6, x_7)=f_{30}(x_4, x_5)$ in $(x_6, x_1{:}x_7)$
```

Fig. 5. S-expression printer and S-expressions parser

5 Related Work

The idea of program inversion can be traced back to [3, 9] where inversion was achieved by local inversion of imperative programs annotated by hand with pre- and postconditions. To speedup combinatorial search programs, a form of local inversion was used in [5] to obtain backtracking commands that undo commands.

Recent work on program inversion uses The Converse of a Function Theorem [2, 14] and studies programmable editors based on bidirectional transformations [10] and injective languages [15]. The only automatic inverters, which we are aware of, are described in [4, 11]. The inversion of functional programs is related to the transformation of logic programs in that logic programs lend themselves to bidirectional computation and reduction of nondeterminism is a major concern [16]. An inference method for Prolog that applies an SLR parsing method for building a proof tree instead of an SLD-resolution was studied in [17].

It would be interesting to study how part of our inversion method, in particular the elimination of nondeterminism, can be utilized in the context of logic programming. Our principles for constructing automatic program inverter which we presented in this paper are based on two insights: the duplication/equality operator introduced in [6] and the idea of using LR parsing for eliminating nondeterminism [7]. A different approach was taken in [8], where a self-applicable partial evaluator was used to convert an inverse interpreter into a program inverter; a method described in [1].

6 Conclusion

We presented a system for inverting programs by local inversion followed by a method to generate deterministic programs based on the observation that the elimination of nondeterminism is similar to LR parsing. We introduced three construction principles for building automatic program inverters, discussed their structure, and applied our system to three non-trivial examples.

Local inversion of grammar programs turned out to be straightforward due to the variable-free design and the use of operations that have operations as their inverses. We should stress that local inversion is sufficient to produce correct inverse programs, even though they will often be nondeterministic. The elimination of nondeterminism introduced as a second step after local inversion is independent of local inversion and can be applied to any nondeterministic program. It may well be applicable to problems outside the domain of program inversion. We have focused on the inversion of injective programs, keeping in mind that our goal is to invert functional programs.

A problem is that programming style matters: some programs can be inverted, while other functionally equivalent programs cannot. This is a common problem for automatic program transformers and well researched in partial evaluation, but more work is needed for program inversion. We plan to study this in more depth, now that we have completed the inverter system.

Among the three construction principles which we identified, local inversion is surprisingly simple and effective, while the other two principles, atomization and elimination of nondeterminism, allow more design choices. In the future, we want to explore further design choices for the atomic language and examine the more powerful LR(k) parsing methods for eliminating more forms of nondeterminism.

Acknowledgements. Thanks to Zhenjiang Hu and Shin-Cheng Mu for many useful discussions related to inversion. Special thanks to Nils Andersen, Yoshihiko Futamura, Neil D. Jones, Torben Mogensen and Jakob Simonsen for helpful comments. We are grateful to the anonymous reviewers for their feedback.

References

1. S. M. Abramov, R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. Schmidt, I. H. Sudborough (eds.), *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, 269–295. Springer-Verlag, 2002.
2. R. Bird, O. de Moor. *Algebra of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
3. E. W. Dijkstra. Program inversion. In F. L. Bauer, M. Broy (eds.), *Program Construction: International Summer School*, LNCS 69, 54–57. Springer-Verlag, 1978.
4. D. Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*, 219–221. Morgan Kaufmann, Inc., 1985.
5. R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, 1967.
6. R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2895, 246–264. Springer-Verlag, 2003.
7. R. Glück, M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In Y. Kameyama, P. J. Stuckey (eds.), *Functional and Logic Programming. Proceedings*, LNCS 2998, 291–306. Springer-Verlag, 2004.
8. R. Glück, Y. Kawada, T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 10–19. ACM Press, 2003.
9. D. Gries. *The Science of Programming*, chapter 21 Inverting Programs, 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
10. Z. Hu, S.-C. Mu, M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 178–189. ACM Press, 2004.
11. H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications. Proceedings*, LNCS 355, 564–568. Springer-Verlag, 1989.
12. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
13. J. McCarthy. The inversion of functions defined by Turing machines. In C. E. Shannon, J. McCarthy (eds.), *Automata Studies*, 177–181. Princeton University Press, 1956.
14. S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 2386, 209–232. Springer-Verlag, 2002.
15. S.-C. Mu, Z. Hu, M. Takeichi. An injective language for reversible computation. In D. Kozen (ed.), *Mathematics of Program Construction. Proceedings*, LNCS 3125, 289–313. Springer-Verlag, 2004.
16. A. Pettorossi, M. Proietti, S. Renault. Reducing nondeterminism while specializing logic programs. In *Proceedings of the Twenty Fourth ACM Symposium on Principles of Programming Languages*, 414–427. ACM Press, 1997.
17. D. A. Rosenblueth, J. C. Peralta. SLR inference an inference system for fixed-mode logic programs based on SLR parsing. *Journal of Logic Programming*, 34(3):227–259, 1998.

A Full Pattern-Based Paradigm for XML Query Processing

Véronique Benzaken[1], Giuseppe Castagna[2], and Cédric Miachon[2]

[1] LRI, UMR 8623, C.N.R.S., Université Paris-Sud, Orsay, France
[2] C.N.R.S., Département d'Informatique, École Normale Supérieure, Paris, France

Abstract. In this article we investigate a novel execution paradigm – ML-like pattern-matching – for XML query processing. We show that such a paradigm is well adapted for a common and frequent set of queries and advocate that it constitutes a candidate for efficient execution of XML queries far better than the current XPath-based query mechanisms. We support our claim by comparing performances of XPath-based queries with pattern based ones, and by comparing the latter with the two efficiency-best XQuery processor we are aware of.

1 Introduction, Motivations, and Goals

In this article we investigate a novel execution paradigm – namely ML-like pattern-matching – for XML query processing. We show that such a paradigm is well adapted for a common and frequent set of queries and thus could be used as a compilation target for XQuery. More precisely, to do so, we endow the pattern-matching based language ℂDuce with an SQL-like query language that we introduce in this article and dub ℂQL. ℂDuce [4, 20] (pronounce "seduce") is a strongly and statically typed *pattern-based* higher-order functional programming language for XML. It is standard compliant (XML, Namespaces, Unicode, XML Schema validation, DTD, etc.) and fully operative and implemented (the distribution of ℂDuce/ℂQL is available at www.cduce.org). One of the distinguishing characteristics of ℂDuce is its pattern algebra. ℂDuce inherits and extends XDuce [21] pattern algebra and implements it by a very efficient "just in time" compilation [19]. ℂQL is a query language in which queries are written using patterns (in the sense of ℂDuce patterns) and where the execution mechanism is based on pattern-matching (in the sense of ML-like languages). ℂQL/ℂDuce patterns are more similar to ML patterns than to XPath expressions. With respect to XPath expressions, ℂDuce patterns are far more declarative inasmuch as while the former strictly indicate navigation paths, the latter reflect the whole structure of matched values and they can be composed by applying boolean combinators.

To demonstrate that pattern-matching is relevant for query compilation and evaluation in the XQuery context, we also introduce, for free, some syntactic sugar to yield an XQuery-like programming style. We call this extension ℂQL$_X$. We chose to experiment with ℂQL$_X$ as we wanted to fully exploit the already implemented ℂDuce's pattern compilation schema and runtime support rather than re-implementing it in the context of XQuery.

Several proposals for defining query languages for XML have been made [2, 6, 14, 15] and a comparative study can be found in [1]. Among them we choose to briefly recall the main features of XQuery[15] as (one of) our purpose is to show that the experiments performed with ℂQL apply obviously to it.

XQuery [15, 17] is becoming the W3C standard in terms of query languages. An earlier proposal was Quilt [11], which borrowed many functionality from XPath [13], XQL [23], XML-QL [16], SQL, and OQL. XQuery is a strongly and statically typed functional language whose type system was largely inspired by XDuce [21].

```
<books-with-prices>
 { for $b in $biblio//book,
       $a in $amazon//entry
   where $b/title = $a/title
   return
     <book-with-prices>
        { $b/title }
       <price-amazon>{$a/price/text()}
         </price-amazon>
         <price-bn>
            { $b/price/text() }
         </price-bn>
     </book-with-prices> }
</books-with-prices>
```

A query is expressed by a FLWR expression: for (iteration on a set of nodes), let (variables binding), where (filter the elements according to a condition), and return (construct the result for each node satisfying the where clause) The query on the side is an example of FLWR expression (it is the Q_5 query of the XML Query Use Cases). Pattern expressions in XQuery, such as $amazon//entry or $a/price/text(), are based on XPath [13]. Many works among them [12] attempts to optimise XQuery evaluation. The immanent purpose of this article is to investigate whether pattern-matching *à la* ℂDuce is well adapted for main memory XML query processing. The answer is positive and ℂDuce's patterns and pattern-matching mechanism can serve as an execution mechanism for XQuery. Indeed, the need for efficient main memory query processing is still of crucial interest. As pointed out by many works a bunch of application scenarios such as message passing, online processing do not manipulate huge documents. We advocate that ℂDuce patterns are a better candidate for XML patterns (again in the sense of [10]) than path expressions. We base our plea on the following observations:

1. ℂDuce patterns are more declarative: different patterns can be combined by boolean combinators, thus, in a declarative way. Furthermore, they can represent the whole structure of matched values. This allows the capture of larger quantities of information in a single match.
2. ℂDuce patterns are more efficient: our measurements show that a query written in $\mathbb{C}QL_X$ using the navigational style is always slower (sometimes even after some optimisation) than the same query written in ℂQL (even when the latter is obtained from the former by an automatic translation). Of course this claim must be counterbalanced by the fact that our comparison takes place in ℂDuce, a language whose implementation was specifically designed for efficient pattern matching resolution. Nevertheless we believe that this holds true also in other settings: the fact that ℂDuce patterns can capture the whole structure of a matched value (compared with paths that can capture only a subpart of it) makes it possible to collect physically distant data in a single match, avoiding in this way further search iterations. To put it simply, while a path expression pinpoints in a tree only subtrees that all share a common property (such as having the same tag) a ℂDuce pattern does more as it can also simultaneously capture several unrelated subtrees.

Our claim is supported by benchmark results. We performed our experiments in ℂDuce rather than XQuery since this is of immediate set up: XPath/XQuery patterns are implemented in ℂDuce as simple syntactic sugar, while an efficient integration of ℂDuce patterns in XQuery would have demanded a complete rewriting of the runtime of a

XQuery processor (but, again, we think that this is the way to go). So instead of comparing results between standard XQuery and a version of XQuery enriched with ℂDuce patterns, we rather compare the results between ℂQL (the standard ℂDuce query language) and ℂQL$_X$ (that is ℂQL in which we only use XQuery patterns and no ℂDuce pattern).

Furthermore, in order not to bias the results with implementation issues, in all our experiments with ℂQL$_X$ we avoided, when this was possible[1], the use of "//" (even if the "//"-operator is available in ℂDuce): whenever in our tests we met a (XQuery) query that used "//" (e.g. the query earlier in this section) we always implemented it by translating (by hand) every occurrence of "//" into a minimal number of "/". Such a solution clearly is much more efficient (we program by hand a minimal number of "/" searches instead of using "//" that performs a complete search on the XML tree) and does not depend on how "//" is implemented in ℂDuce (in ℂDuce "//" is implemented by a recursive function whose execution is much less optimised than that of "/" which, instead, enjoys all the optimisations of the ℂDuce runtime). Therefore it is important to stress that in this article we are comparing hand-optimised XQuery patterns in ℂQL$_X$ with automatically generated (therefore not very optimised) ℂDuce patterns in ℂQL: the results of our tests, which always give the advantage to the second, are thus very strong and robust.

The existence of an automatic translation from (a subset of) XPath patterns to ℂDuce ones, is a central result of our work. This work demonstrates that XPath-like projections are redundant and in a certain sense, with respect to patterns, problematic as they induce a level of query nesting which penalises the overall execution performance. We thus defined a formal translation of ℂQL$_X$ to ℂQL and showed that it preserves typing. This translation maps every ℂQL$_X$ query into a (flat) ℂQL one (i.e., with all nesting levels induced by projections removed), and is automatically implemented by the ℂDuce/ℂQL compiler. Not only such a translation is useful from a theoretical point of view, but (i) it accounts for optimising queries and (ii) shows that the approach can be applied both to standard XQuery (in which case the translation would be used to compile XQuery into a pattern aware runtime) and to a conservative extension of XQuery enriched with ℂDuce patterns (in which case the translation would optimise the code by a source to source translation, as we do for ℂQL$_{(X)}$). Whatever in ℂDuce or in XQuery this transformation allows the programmer to use the preferred style since the more efficient pattern-based code will be always executed. We also adapt logical optimisation techniques which are classical in the database field to the context of pattern based queries and show through performance measurement that they are relevant also in this setting.

To further validate the feasibility of pattern-matching as an execution model we also compared ℂQL performances with those of XQuery processors. Since our language is evaluated in main memory (we do not have any persistent store, yet) as a first choice we compared ℂQL performance with Galax [3] that besides being a reference implementation of XQuery, uses the same technologies as ℂDuce (noticeably, it is implemented in OCaml). However, the primary goal of Galax is compliance with standards rather than

[1] Of course there exist types (such as t = <a>[t | <b>[]]) and queries (//<a>) for which such a translation is not possible.

efficiency and for the time being the (web) available version does not implement any real optimisation and has poor memory management (even if [22] proposes some improvements), which explains that ℂQL outperformed Galax (of an order of magnitude up to tens of thousands of time faster). Therefore we decided to run a second series of tests against Qizx [18] and Qexo [7], the two efficiency best XQuery implementations we are aware of. The tests were performed on the first five XML Query Use Cases [9] and on queries Q1, Q8, Q12, and Q16 of the XMark benchmark [24]. This set of tests gave a first positive answer to practical feasibility of ℂQL-pattern matching. We were pleased to notice that ℂQL was on the average noticeably faster than Qizx and Qexo especially when computing intensive queries such as joins[2] (cf. Q4 and Q5 use cases in Section 4 and query Q8 and Q12 of XMark). These results are even astounding if we consider that while Qizx and Qexo are compiled into bytecode and run on mature and highly optimised Java Virtual Machines (that of course we parametrised to obtain the best possible results), ℂDuce essentially is an interpreted language (it produces some intermediate code just to solve accesses to the heap) with just-in-time compilation of pattern matching. In the "todo" list of ℂDuce a high priority place is taken by the compilation of ℂDuce into OCaml bytecode. We are eager to redo our tests in such a setting, which would constitute a more fair comparison with the Java bytecode and should further increase the advantage of the ℂQL execution model.

Outline. The article is organised as follows. In Section 2 we briefly recall ℂDuce features which are useful for understanding the definition of ℂQL: types, expressions and patterns. In Section 3 we present ℂQL's syntax and semantics. We give the typing rules for the defined language. We then present ℂQL$_X$ showing how to define projections. We formally define the translation from ℂQL$_X$ to ℂQL and show that such a translation yields an unnested ℂQL query and preserves typing. In Section 4 we propose several optimisations and in Section 5 we report on performance measurements we did. We draw our conclusions and present our current and future research directions in Section 6.

2 Types, Expressions and Patterns

A ℂQL query is written as

$$\textsf{select } e_0 \textsf{ from } p_1 \textsf{ in } e_1,\ p_2 \textsf{ in } e_2, \ldots, p_n \textsf{ in } e_n \textsf{ where } c$$

where the p_i's and e_i's respectively denote ℂDuce patterns and expressions. To define ℂQL then we have to define ℂDuce patterns and (a minimal subset of) expressions. A complete presentation of ℂDuce is beyond the scope of this paper (see instead the documentation – tutorial and user manual – and do try the prototype available at `www.cduce.org`), therefore we present here only (a subset of) ℂDuce values and just one complex expression, transform, used to define the semantics of ℂQL queries.

Since in ℂDuce/ℂQL patterns are types with capture variables let us start our presentation with them.

[2] We would like the reader to notice that we did not perform any further optimisation relying on specific data structure such as hash tables. Our very purpose was to assess ℂDuce pattern matching as an execution primitive for XML query processing in which XQuery could be compiled.

2.1 Types

ℂDuce type algebra includes three family of scalar types: (*i*) Integers, that are classified either by the type identifier Int, or by interval types i--j (where i and j are integer literals), or by single integer literals like 42 that denotes the singleton type containing the integer 42. (*ii*) Characters, classified by the type identifiers Char (the Unicode character set) and Byte (the Latin1 character set), by intervals c--d (where c and d are Character literals that is single quoted characters like 'a', 'b', ..., or backslash-escaped directives for special characters, Unicode characters, ...), or by single character literals denoting the corresponding singleton types. (*iii*) Atoms that are user defined constants; they are ℂDuce identifiers escaped by a back-quote such as ‘nil, ‘true, ... and are ranged over by the type identifier Atom or by singleton types.

The other types of ℂDuce's type algebra are (possibly recursively) defined from the previous scalar types and the types Any and Empty (denoting respectively the universal and empty type) by the application of type *constructors* and type *combinators*.

Type combinators. ℂDuce has a complete set of Boolean combinators. Thus if t_1 and t_2 are types, then $t_1\&t_2$ is their intersection type, $t_1 \,|\, t_2$ their union, and $t_1 \setminus t_2$ their difference. For instance the type Bool is defined in ℂDuce as the union of the two singleton types containing the atoms true and false, that is ‘true | ‘false.

Type constructors. ℂDuce has type constructors for record types $\{\, a_1 = t_1; \ldots ; a_n = t_n \,\}$, product types (t_1,t_2), and functional types $(t_1\text{–>}t_2)$. For this paper the most interesting constructors are those for sequences and XML.

Sequence types are denoted by square brackets enclosing a regular expression on types. For instance, in ℂDuce strings are possibly empty sequences of characters of arbitrary length, and thus String is defined and implemented as [Char$*$] (i.e. it is just a handy shortcut). The previous type shows that the content of a sequence type can be conveniently expressed by regular expressions on types, which use standard syntax[3]:

$$R ::= t \mid R\,R \mid R|R \mid R* \mid R+ \mid R?$$

The general form of an XML type is $< t_1\ t_2 > t_3$ with t_i's arbitrary types. In practise t_1 is always a singleton type containing the atom of the tag, t_2 is a record type (of attributes), and t_3 a sequence type (of elements). As a syntactic facility it is possible to omit the back-quote in the atom of t_1 and the curly braces and semicolons in t_2, so that XML types are usually written in the following form: $<\!tag\ \ a_1{=}t_1\ \ a_2{=}t_2\ \ldots\ a_n{=}t_n\!>\![\ R\ ]$.

In the first row Figure 1 we report a DTD for bibliographies followed by the corresponding ℂDuce types: note the use of regular expression types to define the sequence types of elements (PCDATA is yet another ℂDuce convention to denote the regular expression Char*).

2.2 Expressions and Patterns

Expression constructors mostly follow the same syntax as their corresponding type constructors, so a record expression has the form $\{\, a_1 = e_1; \ldots ; a_n = e_n \,\}$, while a pair expression is (e_1,e_2). The same conventions on XML types apply to XML expressions: instead

[3] These are just a very convenient syntactic sugar (very XML-oriented) for particular recursive types.

XML	CDuce
	ℂDuce Types:
`<!ELEMENT bib  (book* )>`	`type Bib = <bib>[Book*]`
`<!ELEMENT book  (title,  (author+ | editor+ ),`	`type Book = <book year=String>[Title`
`                         publisher, price )>`	`          (Author+ | Editor+ ) Publisher Price]`
`<!ATTLIST book  year CDATA  #REQUIRED >`	`type Author = <author>[Last First]`
`<!ELEMENT author  (last, first )>`	`type Editor = <editor>[Last First Affiliation]`
`<!ELEMENT editor  (last, first, affiliation )>`	`type Title  = <title>[PCDATA]`
`<!ELEMENT title  (#PCDATA )>`	`type Last  = <last>[PCDATA]`
`<!ELEMENT last  (#PCDATA )>`	`type First  = <first>[PCDATA]`
`<!ELEMENT first  (#PCDATA )>`	`type Affiliation = <affiliation>[PCDATA]`
`<!ELEMENT affiliation  (#PCDATA )>`	`type Publisher  = <publisher>[PCDATA]`
`<!ELEMENT publisher  (#PCDATA )>`	`type Price  = <price>[PCDATA]`
`<!ELEMENT price  (#PCDATA )>`	
`<?xml version="1.0"?>`	
`<bib>`	`<bib>[`
`  <book year="1994">`	`  <book year="1994">[`
`    <title>TCP/IP Illustrated</title>`	`      <title>"TCP/IP Illustrated"`
`    <author>`	`    <author>[`
`        <last>Stevens</last>`	`      <last>"Stevens"`
`        <first>Richard</first>`	`      <first>"Richard"`
`    </author>`	`    ]`
`    <publisher>Addison-Wesley</publisher>`	`    <publisher>"Addison-Wesley"`
`    <price> 65.95</price>`	`    <price>"65.95"`
`  </book>`	`  ]`
`  <book year="1984">`	`  <book year="1984">[`
`    <title>The Lambda Calculus</title>`	`      <title>"The Lambda Calculus"`
`    <author>`	`    <author>[`
`        <last>Barendegt</last>`	`      <last>"Barendegt"`
`        <first>Henk</first>`	`      <first>"Henk"`
`    </author>`	`    ]`
`    <publisher>North-Holland</publisher>`	`    <publisher>"North-Holland"`
`    <price>92.00</price>`	`    <price>"92.00"`
`  </book>`	`  ]`
`</bib>`	`]`

Fig. 1. DTD/ℂDuce-types and document/values for bibliographies

of writing <‘book {year="1990"}>[...] we rather write <book year="1990">[...]. In the second row of Figure 1 we report on the left a document validating the DTD of the first row and on the right the corresponding (well-typed) value in ℂDuce: note that strings are not enclosed in brackets since they already denote sequences (of characters). Besides expression constructors there are also function definitions and operators (expression destructors). For the purpose of this article we are particularly interested in operators that work on sequences. Besides some standard operators, the most important operator for processing XML data (and the only ℂDuce iterator we present here) is transform, whose syntax is:

$$\text{transform } e \text{ with } p_1 \text{ -> } e_1 \mid p_2 \text{ -> } e_2 \mid \ldots \mid p_n \text{ -> } e_n$$

with $n \geq 1$ and where e, e_1, e_2, ..., e_n are (expressions that return) sequences and p_1, p_2, ..., p_n are *patterns* whose semantics is explained below.

The expression above scans the sequence e and matches each element of e against the patterns, following *a first match policy* (that is, first against p_1 then, only if it fails, against p_2, and so on). If some p_i matches, then the corresponding e_i is evaluated in an environment where variables are substituted by the values captured by the pattern. If no pattern matches, then the empty sequence is returned. When all the elements of *e* have been scanned, transform returns the concatenation of all results[4].

Clearly, in order to fully understand the semantics of transform we need to explain the semantics of patterns. The simplest patterns are variables and types: a variable pattern, say, x always succeeds and captures in x the value it is matched against. If *e* is a sequence of integers then transform *e* with x -> (if x>=0 then [x] else [ ]) returns the subsequence of *e* containing all the positive integers. A type pattern *t* instead succeeds only when matched against a value of type *t*. More complex patterns can be constructed by using type constructors and/or the combinators "&" and "|". So $p_1\&p_2$ succeeds only if both p_1 and p_2 succeed (p_1 and p_2 must have pairwise distinct capture variables), while $p_1|p_2$ succeeds if p_1 succeeds or p_1 fails and p_2 succeeds (p_1 and p_2 must have the same set of capture variables). For instance the pattern x&Int succeeds only if matched against an integer, and in that case the integer at issue is bound to x. Since the type of positive integers can be expressed by the interval 0--∗ (in integer intervals ∗ stands for infinity) then the previous transformation can be also written as transform *e* with x&(0--∗) -> [x] . We can use more alternatives to transform the negative elements into positive ones instead of discarding them: transform *e* with x&(0--∗) -> [x] | x&(∗--0) -> [-x].
If we know that *e* is a sequence formed only of integers, then in the expression above we can omit "&(∗--0)" from the second pattern as it constitutes redundant information (actually ℂDuce automatically gets rid at compile time of all redundant information).

Patterns built by type constructors are equally simple. For instance, the pattern <book year=y>[<title>t <author>[_ f] ;_] matches any bibliographic entry binding to y the value of the attribute year, to t the string of the title, and to f the <first> element of the first author name. The wildcard _ is often used in patterns as a shorthand for the type Any (it matches any value, in the case above it matches the <last> element in the name) while ";_" matches tails of sequences.

Assuming that books denotes a variable of type [Book∗] the code below:

```
transform books with
   | <book year=("1999"|"2000")>[ _ <author>[ _ <first> f ] ;_ ] -> [ f ]
   | <_>[ _ <author>[<last>s ;_] ;_ ] -> [s]
```

scans the sequence of elements of books and for each book it returns the string of the first name if the book was published in 1999 or 2000, or the string of the last name otherwise.

Besides variables and types there are two (last) more building blocks for patterns: default patterns and sequence capture variables.

Default patterns are of the form (x:=*v*); the pattern always succeeds and binds x to the value *v*. Default patterns are useful in the last position of an alternative pattern in order to assign a default value to capture variables that did not match in the preceding alternatives. This allows the programmer to assign default values to optional elements or attributes. For instance imagine that we want to change the previous transform so that if the publication year is not 1999 or 2000 it returns the last name of the *second* author element if it exists, or the string "none" otherwise. It will be changed to:

[4] In short, transform differs from the classic map (also present in ℂDuce) since it uses pattens to filter elements of a sequence and therefore, contrary to map it does not preserve the length of the sequence.

```
transform books with
   | <book year=("1999"|"2000")>[ _  <author>[ _ <first> f ] ;_  ] -> [ f ]
   | <_>[ _  Author  <author>[<last>s _]  ;_  ]  |  (s:="none") -> [ s ]
```

We guarded the second branch by an alternative pattern assigning "none" to s when the first pattern fails (that is, when there is no second author). Note that the string "none" is returned also when the book has editors instead of authors (see the definition of Book type in Figure 1). To filter out books with only editors, the pattern of the second branch should be <_>[_ (Author (<author>[<last>s _] | (s:="none")) ;_]. The pattern succeeds if and only if the title is followed by an author, in which case either it is followed by a second author (whose lastname is then captured by s), or by a publisher (and s is then bound to "none").

Sequence capture variables patterns are of the form x::R where R is a type regular expression; the pattern binds x to a *sequence* of elements. More precisely it binds x to the sequence of all elements matching R (regular expressions match sequences of elements rather than single elements). Such patterns are useful to capture whole subsequences of a sequence. For instance, to return for each book in the bibliography the list of *all* authors and to enclose it in a <authors> tag can be done compactly as follows:

```
  transform books with <book>[ _  (a::Author+) ;_ ] -> [ <authors>a ]
```

Note the difference between [x::Int] and [(x & Int)]. Both accept sequences formed of a single integer, but the first one binds x to a sequence (of a single integer), whereas the second one binds it to the integer itself.

Finally we want to stress that the type inference algorithm of ℂDuce/ℂQL is better than that of XQuery since it always infer a type more precise than the one inferred by XQuery. An example can be found in the extended version of this paper [5].

3 ℂQL: A Pattern-Based Query Language for XML Processing

The formal syntax of ℂQL is given by the following grammar:

Queries

q::=select e from f where c | select e from f

Bindings

f::=p in e , f | p in e

Conditions

c::=‘true | ‘false | not(c) | c or c | c and c | member(e , e) | e bop e

Expressions

e::=x | v | [$e \ldots e$] | flatten(e) | q | $\langle\, e \;\; \ell = e \ldots \ell = e \,\rangle\, e$ | op(e)

Patterns

p::=x | t | p**&**p | p **|** p | (p,p) | $\langle\, p \;\; \ell = p \ldots \ell = p \,\rangle\, p$ | $[r]$ | (x:=v)

Pattern regular expressions

r::=p | $(x :: r)$ | r **|** r | $r\, r$ | $r+$ | $r*$ | $r?$

Types

t::=B | t **|** t | t**&**t | $t \setminus t$ | $\langle\, t \;\; \ell = t \ldots \ell = t \,\rangle\, t$ | $[R]$ | Empty | Any

where op ranges over sequence operators (op∈{distinct_values, count, avg, max, min, sum}), bop over boolean relations (bop∈{=, >>, >=, <<, <=}), x over variables, and v

$$\frac{\Gamma,(t_1/p_1),\ldots,(t_{i-1}/p_{i-1})\vdash e_i:[t_i+] \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n)\vdash e:t,\ c:Bool}{\Gamma\vdash \mathsf{select}\ e\ \mathsf{from}\ p_1\ \mathsf{in}\ e_1,\ \ldots, p_n\ \mathsf{in}\ e_n\ \mathsf{where}\ c\ :\ [\,t*\,]}\ (select)$$

$(\ var(p_i)\wedge var(p_j)=\varnothing$, for $i\neq j\ )$

Fig. 2. Typing rule for queries

over *values* (*viz.* closed expressions in normal formal and constants for integers and characters); flatten takes a sequence of sequences and returns their concatenation (thus, for instance, the infix operator @ that denotes the concatenation of two sequences is encoded as $e_1 @ e_2 =$ flatten $[e_1\ e_2]$).

The non-terminal R used in the definition of types is the one defined in Section 2.1. Patterns, ranged over by p, and types, ranged over by t, are simplified versions of those present in ℂDuce and have already been described; note that types include full boolean combinations: intersection ($t\,\&t$), union ($t\ |\ t$), and difference ($t \setminus t$). The reader can refer to [4] for a more detailed and complete presentation.

```
<books-with-prices>
select <book-with-price>[t1
                  <price-amazon>p2
                  <price-bn>p1 ]
from <bib>[b::Book*] in [biblio],
     <book>[t1&Title _* <price>p1] in b,
     <reviews>[e::Entry*] in [amazon],
     <entry>[t2&Title <price>p2 ;_] in e
where t1=t2
```

As an example, the query Q$_5$ of XQuery described in the introduction would be written in ℂQL as shown on the left-hand side.

The typing rule for the select-from-where construction is given in Figure 2. It states that the condition c must be of type Bool and that the e_i's must be non-empty homogeneous sequences. In this rule (t/p) is the type environment that assigns to the variables of p the best type deduced when matching p against a value of type t and $var(p)$ is the set of variables occurring in p [5] (see [4] for formal definitions and the optimality of the deduced types).

```
transform $e_1$ with $p_1$ ->
   transform $e_2$ with $p_2$ ->
         ⋱
            transform $e_n$ with $p_n$ ->
               if $c$ then [$e_0$] else [ ]
```

The semantics of a select-from-where expression (in the form as it is at the beginning of Section 3) is defined in terms of the translation into ℂDuce given on the left-hand side. In our context transform plays exactly the same role as the "for" construct of XQuery core does [17]. However, the peculiar difference is that our pattern matching compilation schema is based on non-uniform tree automata which fully exploit types to optimise the code [19] as well as its execution. This translation is given only to define in an unambiguous way the semantics of the new term. It is not intended to impose any execution order, since such a choice is left to the query optimiser. In fact the optimiser can implement this more efficiently; for instance, if c does not involve the capture variables of some p_i, the query optimiser can, as customary in databases, push selections (and/or projections) on some components as shown in Section 4.

3.1 ℂQL$_X$

In order to investigate and compare pattern-matching with XQuery, we have extended ℂQL with projection operators *à la* XPath. Let e, be an expression denoting a sequence

[5] The condition $var(p_i)\wedge var(p_j)=\varnothing$ for $i\neq j$ is not strictly necessary but may be useful in practise and simplifies both the proofs and the definition of the optimisations.

XQuery:

```
<bib>
 {
  for $b in $biblio/bib/book
  where $b/publisher = "Addison-Wesley"
           and $b/@year > 1990
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

$\mathbb{C}$QL$_X$:

```
<bib> select <book year=y>[t]
from  b in [biblio]/<book>_ ,
        p in [b]/<publisher>_ ,
        t  in [b]/<title>_ ,
        y in [b]/@year
where (p = <publisher>"Addison-Wesley")
          and (y>>"1990"));;
```

$\mathbb{C}$QL:

```
<bib> select <book year=y>[t]
from  <bib>[b::Book*] in [biblio],
        <book year=y>[t&Title  _+
                      <publisher>"Addison-Wesley";_] in b;;
where y>>"1990"
```

Fig. 3. XQuery and the two $\mathbb{C}$QL programming styles

of XML elements, and t be a $\mathbb{C}$Duce type, then e/t denotes the set of children of the elements in e whose type is t. The formal semantics is defined by encoding: e/t is encoded as transform e with <_>[(x:: t | _)$*$] –> x. It is convenient to introduce the syntax $e/@a$ to extract the sequence of all values bound to the attribute a of elements in e, which is encoded in $\mathbb{C}$Duce as transform e with <_ a=x>_ –> [x].

Figure 3 illustrates how to code the same query in XQuery, $\mathbb{C}$QL$_X$, and $\mathbb{C}$QL. The query at issue is the query Q1 from the XQuery Use Cases. While XQuery and $\mathbb{C}$QL$_X$ code make use of simple variables that are bound to the results of projections, the $\mathbb{C}$QL one fully exploits the pattern algebra of $\mathbb{C}$Duce (we leave as an exercise to the reader how to use regular expressions on types to modify the pattern in the second from clause of the $\mathbb{C}$QL query so as to completely get rid of the where clause).

Finally, since we use $\mathbb{C}$QL$_X$ to mimic XQuery in a full pattern setting, it is important to stress that the semantics of where clauses in $\mathbb{C}$QL (hence in $\mathbb{C}$QL$_X$) is not exactly the same as in XQuery. The latter uses a set semantics according to which a value satisfying the where close will occur at most once in the result (e.g. as for `SELECT DISTINCT` of SQL), while $\mathbb{C}$QL/$\mathbb{C}$QL$_X$ follow the SQL convention and use a multi-set semantics. As usual, the multi-set semantics is more general since the existential semantics can easily obtained by using the distinct_values operator (which takes a sequence and elides multiple occurrences of the same element). The presence of such an operator has no impact on the translation from $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL.

3.2 Translation from $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL

It is quite easy to see that projections can be encoded in $\mathbb{C}$QL, since the two transform expressions used to encode projections correspond, respectively, to: flatten(select x from <_>[(x::t | _)$*$] in e), and to select x from <_ a=x>_ in e. Nonetheless it is interesting to define a more complex translation from $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL that fully exploits the power of $\mathbb{C}$Duce patterns to optimise the code. Therefore in this section we formally define a translation that aims at (i) eliminating projections and pushing them into patterns (ii) transforming as many as possible selection conditions into patterns.

As a result of this translation the number of iterations is reduced and several where clauses are pushed into patterns. In order to formally define the translation we first need to introduce the expression and evaluation contexts $E\{\ \}$ and $C\{\ \}$:

$\bar{q}$ ::= select $\bar{e}$ from $\bar{f}$ where $\bar{c}$ $\mid$ select $\bar{e}$ from $\bar{f}$

$\bar{e}$ ::= x $\mid$ v $\mid$ $[\bar{e}\ldots\bar{e}]$ $\mid$ flatten($\bar{e}$) $\mid$ $\bar{e}\langle\ \ell=\bar{e}\ldots\ell=\bar{e}\ \rangle\ \bar{e}$ $\mid$ ($\bar{e}$,$\bar{e}$) $\mid$ $op(\bar{e})$ $\mid$ $\bar{q}$

$\bar{f}$ ::= p in $\bar{e}$, $\bar{f}$ $\mid$ p in $\bar{e}$

$\bar{c}$ ::= ‘true $\mid$ ‘false $\mid$ not($\bar{c}$) $\mid$ $\bar{c}$ or $\bar{c}$ $\mid$ $\bar{c}$ and $\bar{c}$ $\mid$ $\bar{e}\ bop\ \bar{e}$ $\mid$ member($\bar{e}$,$\bar{e}$)

$E\{\ \}$ ::= $\{\ \}$ $\mid$ $[e_1\ldots e_n\,E\{\ \}\,\bar{e}_1\ldots\bar{e}_m]$ $\mid$ flatten($E\{\ \}$) $\mid$ $\langle\,\bar{e}\ \ell=e\ldots\ell=e\,\rangle\ E\{\ \}$
$\mid$ $\langle\,\bar{e}\ \ell=e\ldots\ell=E\{\ \}\ \ell=\bar{e}\ldots\rangle\ \bar{e}$ $\mid$ ($E\{\ \}$,$\bar{e}$) $\mid$ $E\{\ \}/t$ $\mid$ $E\{\ \}/@a$ $\mid$ $op(E\{\ \})$

$C\{\ \}$ ::= $\{\ \}$ $\mid$ not($C\{\ \}$) $\mid$ c or $C\{\ \}$ $\mid$ $C\{\ \}$ or $\bar{c}$ $\mid$ c and $C\{\ \}$ $\mid$ $C\{\ \}$ and $\bar{c}$
$\mid$ $e\ bop\ E\{\ \}$ $\mid$ $E\{\ \}\ bop\ \bar{e}$ $\mid$ member(e,$E\{\ \}$) $\mid$ member($E\{\ \}$,$\bar{e}$)

where $m,n \geq 0$.

Definition 1. *The translation $\mathcal{P}\llbracket\ \rrbracket$ is defined by the following rewriting rules:*

- $\mathcal{P}\llbracket$ select $E\{q\}$ from f where c $\rrbracket$ = $\mathcal{P}\llbracket$ select $E\{\mathcal{P}\llbracket q\rrbracket\}$ from f where c $\rrbracket$, *q is not a $\bar{q}$ expression*
- $\mathcal{P}\llbracket$ select $E\{\bar{e}/t\}$ from f where c $\rrbracket$ = $\mathcal{P}\llbracket$ select $E\{x\}$ from f, x in $[\bar{e}/t]$ where c $\rrbracket$, $x \notin bv(f)$
- $\mathcal{P}\llbracket$ select $E\{\bar{e}/@a\}$ from f where c $\rrbracket$ = $\mathcal{P}\llbracket$ select $E\{x\}$ from f, x in $[\bar{e}/@a]$ where c $\rrbracket$, $x \notin bv(f)$
- $\mathcal{P}\llbracket$ select $\bar{e}$ from f where $C\{q\}$ $\rrbracket$ = $\mathcal{P}\llbracket$ select $\bar{e}$ from f where $C\{\mathcal{P}\llbracket q\rrbracket\}$ $\rrbracket$, *q is not a $\bar{q}$ expression*
- $\mathcal{P}\llbracket$ select $\bar{e}_0$ from f where $C\{\bar{e}/t\}$ $\rrbracket$ = $\mathcal{P}\llbracket$ select $\bar{e}_0$ from f, x in $[\bar{e}/t]$ where $C\{\mathrm{x}\}$ $\rrbracket$, $x \notin bv(f)$
- $\mathcal{P}\llbracket$select $\bar{e}_0$ from f where $C\{\bar{e}/@a\}\rrbracket$ = $\mathcal{P}\llbracket$ select $\bar{e}_0$ from f,x in $[\bar{e}/@a]$ where $C\{[x]\}$ $\rrbracket$, $x\notin bv(f)$
- $\mathcal{P}\llbracket$ select $\bar{e}$ from f where $\bar{c}$ $\rrbracket$ = $\mathcal{P}\llbracket$ select $\bar{e}$ from $\mathcal{F}\llbracket f \rrbracket_{bv(f)}$ where $\bar{c}$ $\rrbracket$

where $\mathcal{F}\llbracket\ \rrbracket$ is defined as:

- $\mathcal{F}\llbracket p$ in $e, f\rrbracket_\Gamma$ = $\mathcal{F}\llbracket p$ in $e\rrbracket_\Gamma$, $\mathcal{F}\llbracket f\rrbracket_{\Gamma\cup bv(\mathcal{F}\llbracket p \text{ in } e\rrbracket_\Gamma)}$
- $\mathcal{F}\llbracket p$ in $E\{q\}\rrbracket_\Gamma$ = $\mathcal{F}\llbracket p$ in $E\{\mathcal{P}\llbracket q\rrbracket\}\rrbracket_\Gamma$
- $\mathcal{F}\llbracket p$ in $E\{\bar{e}/t\}\rrbracket_\Gamma$ = *if $\bar{e}$ has type* [Any] *then* <_>[(x::t|_)*] in $\bar{e}$, $\mathcal{F}\llbracket p$ in $E\{x\}\rrbracket_{\Gamma\cup\{x\}}$, $x \notin \Gamma$
  else [(<_>[(x::t|_)∗] | x::‘nil)*] in $[\bar{e}]$, $\mathcal{F}\llbracket p$ in $E\{$flatten(x)$\}\rrbracket_{\Gamma\cup\{x\}}$, $x \notin \Gamma$
- $\mathcal{F}\llbracket p$ in $E\{\bar{e}/@a\}\rrbracket_\Gamma$ = *if $\bar{e}$ has type* [Any] *then* <_ a=x>_ in $\bar{e}$, $\mathcal{F}\llbracket p$ in $E\{[x]\}\rrbracket_{\Gamma\cup\{x\}}$, $x \notin \Gamma$
  else [(<_ a=x>_ | x::‘nil)∗] in $[\bar{e}]$, $\mathcal{F}\llbracket p$ in $E\{x\}\rrbracket_{\Gamma\cup\{x\}}$, $x \notin \Gamma$
- $\mathcal{F}\llbracket p$ in $\bar{e}\rrbracket_\Gamma$ = p in $\bar{e}$

Apart from technical details, the translation is rather simple: contexts are defined so that projections are replaced according to an innermost-rightmost strategy. For instance if we have to translate x/t1/t2, (x/t1) will be considered first thus removing the projection on t1 prior to performing the same translation on the remainder. The first three rules replace projections in the select part, 2nd and 3rd rules incidentally perform a slight optimisation (they get rid of one level of sequence nesting) in the case the projection is applied to a sequence with just one element (this case is very common and dealing with it allows for further optimisation later on); the 4th, 5th, and 6th rules replace projections in the “where” part by defining new patterns in the “from” part, while the 7th rule handles projections in the “from” part. The latter resorts to an auxiliary function $\mathcal{F}$ that needs to store in a set Γ the capture variables freshly introduced.

So far, we have proved just a partial correctness property of the translation, namely that it preserves the types (the proof is omitted for space reasons):

Theorem 1 (Type preservation). $\Gamma \vdash q : t \Rightarrow \Gamma \vdash \mathcal{P}[\![\, q \,]\!] : t$

The result of $\mathcal{P}$ is a query in $\mathbb{C}$QL since all projections have been removed. From a practical viewpoint, the use of a projection in a query is equivalent to that of a nested query.

```
<bib>
select <book year=y>[t]
from b in (select v from <_>[(yb::Book| _)*] in [biblio], v in yb)
     p in (select v from <_>[(yp::Publisher| _)*] in [b], v in yp)
      t  in (select v from <_>[(yt::Title| _)*] in [b], v in yt)
      y in (select yy from <_year=yy>_  in [b])
  where (p = <publisher>"Addison-Wesley") and (y>>"1990")
```

Let Q be the query obtained from the $\mathbb{C}$QL$_X$ query in Figure 3 by replacing <book>_, <publisher>_, and <title>_ respectively by Book, Publisher, and Title (the resulting query is semantically equivalent but more readable and compact). If we expand the projections of Q into its corresponding sequence of select's we obtain the query at the end of the previous page (we used boldface to outline modifications).

The translation $\mathcal{P}[\![\ ]\!]$ unnests these select's yielding the query of Figure 4. In the next section we show that this has a very positive impact on performances.

```
<bib>
  select <book year=y>[t]
  from <_>[(yb::Book|_)*] in [biblio],
       b in yb,
       <_>[(yp::Publisher|_)*] in [b],
       p in yp,
       <_>[(yt::Title|_)*] in [b],
       t in yt,
       <_ year=y>_ in [b]
  where (p = <publisher>"Addison-Wesley")
          and (y>>"1990")
```

Fig. 4. $\mathcal{P}[\![\, Q \,]\!]$

4 Pattern Query Optimisation

In this section we adapt classical database optimisation techniques to our patterns. Such optimisations evaluate, as customary in the database world, conditions just when needed thus reducing the size of intermediate data contributing in the result construction and also avoiding to visit useless parts of the document. More precisely, we proceed in four steps.

Conjunctive normal form. The first step consists in putting the where condition in conjunctive normal form and then moving into the from clause the parts of the condition that can be expressed by a pattern. This is expressed by the following rewriting rule:

$$\text{select } e \text{ from } f \text{ where } c \quad \rightsquigarrow \quad \text{select } e \text{ from } f, \Theta^1(CNF(c)) \text{ where } \Theta^2(CNF(c))$$

where CNF(c) is a conjunctive normal form of c, $\Theta^1(c)$ represents the part of c that can be expressed by a pattern and thus remounted in the "from" part, and $\Theta^2(c)$ is the part of c that remains in the "where" condition. Formally, Θ^1 and Θ^2 are the first and second projections of the function Θ defined as:

Definition 2. *Let i denote a scalar (i.e. an integer or a character) and v a value*

$\Theta(v{=}e) = (v \text{ in } [e], \text{'true})$ $\qquad$ $\Theta(\text{count}(e) = i) = (\,[\,\_^{i}] \text{ in } [e], \text{'true})$

$\Theta(e >= i) = (\, i{-}{-}{*} \text{ in } [e], \text{'true})$ $\qquad$ $\Theta(\text{count}(e) >> i) = (\,[\,\_^{i}\ \_{+}] \text{ in } [e], \text{'true})$

$\Theta(e >> i) = (\, [\![i+1]\!]{-}{-}{*} \text{ in } [e], \text{'true})$ $\qquad$ $\Theta(\text{count}(e) >= i) = (\,[\,\_^{i}\ \_^{*}] \text{ in } [e]\ , \text{'true})$

$\Theta(e <= i) = (\, {*}{-}{-}i \text{ in } [e], \text{'true})$ $\qquad$ $\Theta(\text{count}(e) << i) = (\,[(\_?)^{i-1}] \text{ in } [e]\ , \text{'true})$

$\Theta(e << i) = (\, {*}{-}{-}[\![i-1]\!] \text{ in } [e], \text{'true})$ $\qquad$ $\Theta(\text{count}(e) <= i) = (\,[(\_?)^{i}] \text{ in } [e], \text{'true})$

$\Theta(\text{member}(v,e)) = (\,[\,\_^{*}\ v\ \_^{*}] \text{ in } [e], \text{'true})$

$\Theta(c_1 \text{ and} \ldots \text{and } c_n) = \big(\, (\Theta^1(c_1), \ldots, \Theta^1(c_n))\ , \Theta^2(c_1) \text{ and} \ldots \text{and } \Theta^2(c_n) \,\big).$

$\Theta(c) = (\varepsilon, c)$ *(if none of the above applies)*

```
<bib>
  select <book year=y>[t]
  from <_>[(yb::Book_)*] in [biblio],
       b in yb,
       <_>[(yp::Publisher|_)*] in [b],
       p in yp,
       <_>[(yt::Title|_)*] in [b],
       t in yt,
       <_ year=y>_ in [b],
       <publisher>"Addison-Wesley" in [p]
  where y>>"1990"
```

Fig. 5. Θ on $\mathcal{P}[\![\, Q \,]\!]$

```
<bib>
select <book year=y>[t]
from <_>[(yb::Book|_)*] in [biblio],
        <_ year=y>_&
        <_>[(yp::Publisher|_)*]&
        <_>[(yt::Title|_)*] in yb,
        <publisher>"Addison-Wesley" in yp,
        t in yt
where y>>"1990"
```

Fig. 6. $\mathcal{P}[\![\, Q \,]\!]$ after the first 3 optimisation steps

where we use the notation $\_^i$ to denote the juxtaposition of i occurrences of "_". So for instance the third rule indicates that the constraint, say, count(e) = 3 can be equivalently checked by matching e against the pattern [_ _ _] (i.e. [$\_^3$]). We also used the notation $[\![f(i)]\!]$ for the constant that is the result of $f(i)$.

If we apply the rewriting to the query in Figure 4, then we can use the first case of Definition 2 to express the condition p = <publisher>"Addison-Wesley" by a pattern, yielding the query of Figure 5 (the rewriting does not apply to y >> "1990" since "1990" is not a scalar but a string).

Useless declarations elimination. The second step consists in getting rid of useless intermediate in-declarations that are likely to be generated by the translation of $\mathbb{C}\text{QL}_X$ into $\mathbb{C}$QL:

select $e_\circ$ from f_1, x in e, f_2, p in $[x], f_3$ where c $\rightsquigarrow$ select $e_\circ$ from $f_1, x\&p$ in e, f_2, f_3 where c

Pattern consolidation. The third step of optimisation consists in gathering together patterns that are matched against the same sequences.

select $e_\circ$ from f_1, p_1 in e, f_2, p_2 in e, f_3 where c $\rightsquigarrow$ select $e_\circ$ from $f_1, p_1\&p_2$ in e, f_2, f_3 where c

Note that both rewriting systems are confluent and noetherian (when applied in the order). The result of applying these rewriting rules to the query in Figure 5 is shown in Figure 6.

Pushing selections. The fourth and last step is the classical technique that pushes selections as close as possible to the declarations of the variables they use. So for instance the clause y >> "1990" in Figure 6 is checked right after the capture of y. This is obtained by anticipating an if_then_else in the implementation[6]. The algorithm, which is standard, is omitted.

This kind of optimisation is definitely not new. It corresponds to a classical logical optimisation for the relational model. The benefit is to reduce the size of the intermediate data that is used to the obtain the result.

[6] More precisely, the query in Figure 6 is implemented by

```
transform [biblio] with <_>[(yb::Book|_)*] ->
   transform yb with <_ year=y>[(yp::Publisher|yt::Title|_)*] ->
     if (y>>"1990") then transform yp with <publisher>"Addison-Wesley" -> [<book year=y>[t]]
        else [ ]
```

5 Experimental Validation

Performance measurements were executed on a Pentium IV 3.2GHz with 1GB of RAM running under FreeBSD 5.2.1. We compared performance results of the same query programmed in the different flavors of $\mathbb{C}$QL. So we tested a same query in: (i) $\mathbb{C}\mathrm{QL}_X$, that is $\mathbb{C}$QL in which we use the same path expressions as the XQuery query and no $\mathbb{C}$Duce pattern, (ii) $\mathbb{C}\mathrm{QL}_P$, the $\mathbb{C}$QL program automatically generated by applying to the $\mathbb{C}\mathrm{QL}_X$ query the transformation $\Theta(\mathcal{P}[\![\ ]\!])$ of Sections 3.2 and 4 and the two rewritings for pattern consolidation and useless declaration elimination that clean up the "garbage" introduced by the translation, (iii, iv) $\mathbb{C}\mathrm{QL}_X{}^{opt}$ $\mathbb{C}\mathrm{QL}_P{}^{opt}$, which are obtained by optimising the two previous queries by the classical optimisation algorithm of pushing selections, presented at the end of Section 4, and finally (v) $\mathbb{C}$QL that is a handwritten (and hand optimised) query in $\mathbb{C}$QL. As we explained in the introduction, in order to strengthen our results we chose not to use "//" in $\mathbb{C}\mathrm{QL}_X$ (since this is less heavily optimised by the $\mathbb{C}$Duce runtime than "/") and instead always use the static types to translate it in terms of "/" (as all the queries we considered allowed us to do so). This gives a clear further advantage to $\mathbb{C}\mathrm{QL}_X$.

To perform our tests we chose, somewhat arbitrarily, queries Q1, Q2, Q3, Q4, and Q5 of the XML Query Use Cases. We then performed a second set of tests based on the XMark benchmarks. Here our choice of queries was less random as we explain below.

Finally to test $\mathbb{C}$QL runtime we compared our results with three different implementations of XQuery: Galax [3], Qizx [18], and Qexo [7, 8]. Galax is a reference implementation of XQuery and, as $\mathbb{C}$Duce, it is implemented in OCaml. Qizx/open is an open-source Java implementation of XQuery specifications developed for commercial distribution and whose target is the efficient execution of queries over large databases. Qexo is a partial implementation of the XQuery language that achieves high performance by compiling queries down to Java bytecode using the Kawa framework. The sources of the queries are omitted for space reasons but they can all be found in the extended version [5].

5.1 Use Cases

Briefly, query Q1 performs a simple selection. Queries Q2 and Q3 are "reconstructing" queries, they both scan the whole bibliography, while the first one returns a flat list of title author pairs (each pair being enclosed in a <result> element), the second returns the title and all authors grouped in a <result> element. For each author in the bibliography, query Q4 lists the author's name and the titles of all books by that author, grouped inside a "result" element. Last, query Q5 performs a join between two documents: for each book found both in the document bound to biblio and in that bound to amazon Q5 lists the title of the book and its price from each source.

The results of our tests are presented in Figure 7, from which we omitted the times for Galax: as a matter of fact we did not completed all the tests of Galax since it was soon clear that the performances of Galax are several orders of magnitude worse than those of Qizx and Qexo.

Measurements were performed for each query on randomly generated documents of different sizes (expressed in KBytes). We also followed the spirit of the benchmark and we generated documents with a selectivity rate (that we judged) typical of the bib-

	Size	Size2	flt$_{\mathbb{C}QL}$	$\mathbb{C}$QL$_X$	$\mathbb{C}$QL$_X^{opt}$	$\mathbb{C}$QL$_P$	$\mathbb{C}$QL$_P^{opt}$	$\mathbb{C}$QL	Qizx	Qexo
Q1	36 Kb		0.01	0.01	0.01	0.02	0.01	0.01	0.45	0.60
Q1	1.8 Mb		0.23	0.26	0.25	0.26	0.26	0.24	0.76	1.01
Q1	14 Mb		1.90	2.00	1.99	1.98	2.07	1.93	2.18	2.89
Q1	35 Mb		4.79	5.13	5.04	5.03	5.24	4.90	4.44	5.80
Q2	36 Kb		0.01	0.01	0.01	0.01	0.01	0.01	0.46	0.61
Q2	1.8 Mb		0.24	0.26	0.26	0.25	0.25	0.25	1.00	1.04
Q2	14 Mb		1.87	2.06	2.06	2.01	2.01	1.99	3.77	3.55
Q2	35 Mb		4.74	5.27	5.27	5.14	5.14	5.08	8.16	7.79
Q3	36 Kb		0.01	0.01	0.01	0.01	0.01	0.01	0.47	0.60
Q3	1.8 Mb		0.24	0.25	0.26	0.25	0.25	0.25	0.99	1.03
Q3	14 Mb		1.90	2.03	2.02	2.01	2.02	2.01	3.66	3.27
Q3	35 Mb		4.81	5.18	5.18	5.14	5.14	5.13	7.90	6.86
Q4	36 Kb		0.01	0.05	0.05	0.05	0.05	0.05	0.53	
Q4	70 Kb		0.02	0.17	0.17	0.14	0.14	0.14	0.68	
Q4	144 Kb		0.02	0.61	0.61	0.52	0.52	0.49	1.17	
Q4	575 Kb		0.09	10.73	10.73	9.94	9.94	8.63	10.97	
Q4	1.8 Mb		0.24	113.01	113.01	89.31	89.31	88.70	104.12	
Q5	36 Kb	535 Kb	0.08	1.69	0.79	1.17	0.71	0.54	4.44	27.88
Q5	144 Kb	43 Kb	0.03	0.52	0.24	0.38	0.24	0.17	1.79	9.31
Q5	575 Kb	171 Kb	0.11	7.87	3.49	5.92	3.34	2.46	20.74	127.39
Q5	1.8 Mb	535 Kb	0.31	78.27	36.54	53.25	31.04	22.93	197	>1h
Q5	3.5 Mb	535 Kb	0.55	157.70	72.28	105.38	62.24	45.02	392	

(flt = file load time, Size2 column reports the sizes of the second document in joins)

Fig. 7. Summary of all test results on the XQuery Use Cases

liographic application (that is quite low). Each test was repeated several times and the table reports the average evaluation times (in seconds). We have reported the loading time (in the column headed by "flt", file load time) of the XML document from the global execution time in order to separate the weight of the query engine and that of the parser in the overall performances (of course we are interested in the former and not in the latter). The execution times always include the time for performing optimisation, when this applies, for type checking (just for $\mathbb{C}$QL variants) and the file load time.

By comparing the load time with the overall execution time (we recall that the latter includes the former) it is clear that the only computationally meaningful queries are the Q4 and Q5 ones (Q4 was not executed in Qexo since it does not implement the distinct_values operator). In these two cases the best performances are obtained by $\mathbb{C}$QL[7].

[7] A word must be spent on the performances of Q5 for Qizx. Whenever Qizx syntactically detects a conjunctive join condition it dynamically generates a hash table to index it (for low selective benchmarks, as the one we performed in our tests, this brings far better performance, of course). Since we wanted to compare the performances of the query engines (rather than the OCaml and Java libraries for hash tables) and because we believe that index generation must be delegated to the query optimiser rather than implemented by the compiler, then in our test we disabled this index generation (this is done by just adding an "or false" to the join condition).

	Size	flt_{CQL}	$\mathbb{C}\text{QL}_X$	$\mathbb{C}\text{QL}_P$	$\mathbb{C}$QL	Qizx	Qexo
Q1	1.5 Mb	0.15	0.15	0.15	0.15	0.57	0.74
Q1	29 Mb	2.57	2.58	2.58	2.58	2.16	2.58
Q1	72 Mb	6.61	6.65	6.64	6.62	4.42	5.08
Q1	145 Mb	14.10	14.18	14.15	14.13	8.16	9.31
Q8	1.5 Mb	0.15	0.21	0.21	0.17	1.00	34.51
Q8	29 Mb	2.57	26.03	22.96	13.09	75.90	>1h
Q8	72 Mb	6.61	156	133	72.81	476	
Q8	145 Mb	14.19	630	542	285	1838	
Q12	1.5 Mb	0.16	0.21	0.21	0.20	0.87	
Q12	29 Mb	2.59	21.22	20.57	14.70	38.30	
Q12	72 Mb	6.68	127	122	86.35	216	
Q12	145 Mb	14.36	481	457	319	824	
Q16	1.5 Mb	0.15	0.16	0.16	0.16	0.62	0.78
Q16	29 Mb	2.57	2.65	2.64	2.63	2.15	2.63
Q16	72 Mb	6.63	6.87	6.85	6.82	4.42	5.08
Q16	145 Mb	14.24	14.60	14.54	14.50	8.16	9.31

(flt = file load time)

Fig. 8. Summary of all test results on XMark

5.2 XMark

Following the suggestion of Ioana Manolescu (one of the XMark authors) we chose four queries that should give a good overview of the main memory behaviour of the query engines.

More precisely, our choice went on Q1, just because it is the simplest one, Q8 of the "chasing references" section since it performs horizontal traversals with increasing complexity, Q12 of the "join on values" section since it tests the ability to handle large intermediate results, and finally on Q16 of the "path traversals" section to test vertical traversals.

The results are summarised in Figure 8. We did not perform the tests on the optimised versions of $\mathbb{C}\text{QL}_X$ and $\mathbb{C}\text{QL}_P$ since on the specific queries they are the identity function. Q12 times for Qexo are absent because execution always ended with an unhandled exception (probably because of a bug in the implementation of the arithmetic library of Qexo but we did not further investigate).

Once more if we compare the load time with the execution time we see that the only interesting queries to judge the quality of the engines are Q8 and Q12. In the other queries the execution time is very close to the load time, so the performance is completely determined by parsers. In these cases it is interesting to note that while $\mathbb{C}$QL uses an external parser, namely Expat (but the $\mathbb{C}$Duce interpreter has an option that the programmer can specify to use the less efficient but more debug-friendly Pxp parser), Qexo and Qizx have event driven parsers that interact with the query engines in order to avoid loading of useless parts of the document, whence the better performances. That said, when performances are determined by the query engine, as in Q8 and Q12, $\mathbb{C}$QL shows once more the best results[8].

[8] Once more the test for Qizx was performed with the dynamic generation of hash tables disabled.

6 Conclusion and Perspectives

In this article we presented ℂQL, a full pattern-matching based query language for XML embedded in ℂDuce. Our main purpose was to demonstrate that patterns and pattern matching (in ℂDuce sense) are a good candidate as an evaluation mechanism for XML query processing. To do so, we first coupled ℂDuce patterns with a syntax very common in the database area (select-from-where) and defined the new syntax in terms of a translation semantics. Then, we extended this syntax to allow for an XQuery-like style (ℂQL$_X$). We chose to experiment with ℂQL$_X$ rather than with XQuery because we wanted to rely on the very efficient pattern matching compilation and execution of ℂDuce. Indeed, patterns are compiled into non-uniform tree automata [19] which fully exploit type information. In order to demonstrate the power of pure patterns, we provided an automatic translation from the former into the latter. Such a translation not only shows that projections are useless in the context of ℂDuce patterns but also gives a formal way of unnesting queries. Then we investigated further optimisations: the well-known database logical optimisations. We therefore adapted such optimisations to the context of ℂQL, providing a first step toward devising a general query optimiser in such a context.

In order to validate our approach we performed performance measurements. The purposes of such measurements were twofold: first, we wanted to demonstrate the efficiency of pattern-matching based query languages by comparing ℂQL with efficiency-oriented implementations of XQuery and, second, we wanted to check the relevance of classical database logical optimisation techniques in a pattern-based framework. In both cases, the obtained results were encouraging.

We are currently working on the following topics: (*i*) develop further (classical) query optimisation techniques such as joins re-ordering, (*ii*) extend the pattern algebra in order to partially account for descendants axes (at the expense of loosing type precision but still reaching a typed construction) (*iii*) formally study the expressive power of ℂQL and finally, in a longer-term, (*iv*) coupling ℂQL with a persistent store and study physical optimisation techniques.

Acknowledgements

The authors want to thank Massimo Marchiori for the pointer to Qexo, Xavier Franc for the details on Qizx implementation, and Alain Frisch for the comments on this paper and his help with the implementation of ℂDuce. Thanks to Dario Colazzo for his help on XQuery type system. Very special thanks to Ioana Manolescu for her invaluable suggestions and careful reading of a preliminary version of this paper.

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : from Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
3. Bell-labs. *Galax*. `http://db.bell-labs.com/galax/`.
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM Int. Conf. on Functional Programming*, pages 51–63, 2003.

5. V. Benzaken, G. Castagna, and C. Miachon. CQL: a pattern-based query language for XML. Complete version. Available at `http://www.cduce.org/papers`, 2005.
6. N. Bidoit and M. Ykhlef. Fixpoint calculus for querying semistructured data. In *Int. Workshop on World Wide Web and Databases (WebDB)*, 1998.
7. P. Bothner. Qexo - the GNU Kawa implementation of XQuery. Available at http://www.gnu.org/software/qexo/.
8. P. Bothner. Compiling XQuery to java bytecodes. In *Proceedings of the First Int. Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>*, pages 31–37, 2004.
9. Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. T.-R. 20030822, World Wide Web Consortium, 2003.
10. Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML query (XQuery) requirements. Technical Report 20030627, World Wide Web Consortium, 2003.
11. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB 2000 (selected papers)*, volume 1997 of LNCS, pages 1–25, 2001.
12. Z. Chen, H. V. Jagadish, L. Lakshmanam, and S Paparizos. From tree patterns to generalised tree paterns: On efficient evaluation of xquery. In *VLDB'03*, pages 237–248, 2003.
13. J. Clark and S. DeRose. *XML Path Language (XPath).* W3C Recommendation, `http://www.w3.org/TR/xpath/`, November 1999.
14. G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. "The Query Language TQL". In *In 5th Int. Workshop on the Web and Databases (WebDB)*, 2002.
15. World Wide Web Consortium. XQuery: the W3C query language for XML – W3C working draft, 2001.
16. A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, 1998.
17. M. Fernández, J. Siméon, and P. Wadler. An algebra for XML query. In *Foundations of Software Technology and Theoretical Computer Science*, number 1974 in LNCS, 2000.
18. X. Franc. Qizx/open. http://www.xfra.net/qizxopen.
19. A. Frisch. Regular tree language recognition with static information. In *Proc. of the 3rd IFIP Conference on Theoretical Computer Science (TCS)*, Toulouse, Kluwer, 2004.
20. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.
21. H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
22. Amélie Marian and Jérôme Siméon. Projecting XML elements. In *Int. Conference on Very Large Databases VLDB'03*, pages 213–224, 2003.
23. A. J. Robie, J. Lapp, and D. Schach. "XML Query Language (XQL). In *WWW The Query Language Workshop (QL)*, Cambridge, MA, , 1998.
24. Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *Proceedings of the Int'l. Conference on Very Large Database Management (VLDB)*, pages 974–985, 2002.

Type Class Directives

Bastiaan Heeren and Jurriaan Hage

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{bastiaan,jur}@cs.uu.nl

Abstract. The goal of this paper is to improve the type error messages in the presence of `Haskell 98` type classes, in particular for the non-expert user. As a language feature, type classes are very pervasive, and strongly influence what is reported and when, even in relatively simple programs. We propose four type class directives, and specialized type rules, to lend high-level support to compilers to improve the type error messages. Both have been implemented, and can be used to easily modify the behavior of the type inference process.

Keywords: type error messages, type classes, directives, domain-specific programming

1 Introduction

Improving the type error messages for higher-order, polymorphic, functional programming languages continues to be an area of activity [1–4]. Type classes have been studied thoroughly, both in theory and practice. In spite of this, very little attention has been devoted to compensate the effects type classes have on the quality of type error messages. In this paper, we present a practical and original solution to improve the quality of type error messages by scripting the type inference process.

To illustrate the problem type classes introduce, consider the following attempt to decrement the elements of a list.

```
f xs = map -1 xs
```

The parse tree for this expression does not correspond to what the spacing suggests: the literal `1` is applied to `xs`, the result of which is subtracted from `map`. Notwithstanding, `GHC` will infer the following type for `f`. (`Hugs` will reject `f` because of an illegal `Haskell 98` class constraint in the inferred type.)

```
f :: (Num (t -> (a -> b) -> [a] -> [b]),
      Num ((a -> b) -> [a] -> [b])) => t -> (a -> b) -> [a] -> [b]
```

Both subtraction and the literal `1` are overloaded in `f`'s definition[1]. Although the polymorphic type of `1` is constrained to the type class `Num`, this restriction does

[1] In `Haskell`, we have `1 :: Num a => a` and `(-) :: Num a => a -> a -> a`.

not lead to a type error. Instead, the constraint is propagated into the type of `f`. Moreover, unifications change the constrained type into a type which is unlikely to be a member of `Num`. A compiler cannot reject `f` since the instances required could be given later on. This *open-world* approach for type classes is likely to cause problems at the site where `f` is used. One of our directives allows us to specify that function types will *never* be part of the `Num` type class. With this knowledge we can reject `f` at the site of definition.

In this paper, we will improve the type error messages for `Haskell 98` type classes [5], in particular for the non-expert user. (Extensions to the class system, like multi-parameter type classes, are outside the scope of this paper.) Our approach is to design a language for type inference directives to adapt the type inference process, and, in particular, to influence the error message reporting facility.

For a given module `X.hs`, the directives can be found in a file called `X.type`. If `X.hs` imports a module `Y.hs`, then all its directives are included as well. This applies transitively to the modules that `Y.hs` imports, which we achieve by storing directive information in object files. Our approach is similar to the one followed in an earlier paper [6], in which we did not consider overloading.

The use of compiler directives has a number of advantages: They are not part of the programming language, and can be easily turned off. The directives function as a high-level specification language for parts of the type inference process, precluding the need to know anything about how type inferencing is implemented in the compiler. In tandem with our automatic soundness and sanity checks for each directive, this makes them relatively easy to use. Also, it should be straightforward for other compilers to support our directives as well. Finally, although the focus of this paper is on the type inference process, the compiler directive approach lends itself to other program analyses as well.

This paper makes the following contributions.

1. We present four type class directives to improve the resolution of overloading (Section 2). With these directives we can report special purpose error messages, reject suspicious class contexts, improve inferred types, and disambiguate programs in a precise way.
2. We discuss how the proposed directives can be incorporated into the process of context reduction (Section 3).
3. We give a general language to describe invariants over type classes (Section 4). This language generalizes some of our proposed directives.
4. We extend the specialized type rules [6] to handle type classes (Section 5). As a consequence, we can report precise and tailor-made error messages for incorrect uses of an overloaded function.

The type directives proposed in Section 2 have been implemented in our type inference framework. The `Helium` compiler [7] is based on this framework, and supports the extended specialized type rules.

2 Type Class Directives

In `Haskell`, new type classes are introduced with a *class declaration*. If a list of superclasses is given at this point, then the instances of the type class must also be member of each superclass; this is enforced by the compiler. To make a type a member of a type class, we simply provide an *instance declaration*. Other means for specifying type classes do not exist in `Haskell`.

Therefore, some properties of a type class cannot be described: for example, we cannot exclude a type from a type class. To gain more flexibility, we propose type class directives to enrich the specification of type classes. Each of these have been implemented in our type inference framework.

The first directive we introduce is the `never` directive (Section 2.1), which excludes a single type from a type class. This is the exact opposite of an instance declaration, and limits the *open-world* character of that type class. Similar to this case-by-case directive, we introduce a second directive to disallow new instances for a type class altogether (Section 2.2). A closed type class has the advantage that we know its limited set of instances.

Knowing the set of instances of a type class opens the door for two optimizations. In the exceptional case that a type class is empty, we can reject every function that requires some instance of that class. If the type class `X` has only one member, say the type `t`, then a predicate of the form `X a` can improve `a` to `t`. This is, in fact, an improvement substitution in Jones' theory of qualified types [8]. If we have `(X a, Y a)`, and the set of shared instances is empty or a singleton, then the same reasoning applies. For example, if the instances of `X` are `Int` and `Bool`, and those of `Y` are `Bool` and `Char`, then `a` must be `Bool`. This is easily discovered for `Haskell 98` type classes by taking intersections of sets of instances.

Our next directive, the `disjoint` directive, specifies that the intersection of two type classes should be empty (Section 2.3). This is another instance of an invariant over sets of types, which is formulated by the programmer, and maintained by the compiler. In Section 4, we present a small language to capture this invariant, and many others besides.

Finally, Section 2.4 discusses a `default` directive for type classes, which helps to disambiguate in case overloading cannot be resolved. This directive refines the ad hoc default declarations supported by `Haskell`.

In the remainder of this section, we explore the directives in more detail, and conclude with a short section on error message attributes.

2.1 The `never` Directive

Our first directive lets us formulate explicitly that a type should never become a member of a certain type class. This statement can be accompanied with a special purpose error message, reported in case the forbidden instance is needed to resolve overloading. The main advantage of the `never` directive is the tailor-made error message for a particular case in which overloading cannot be resolved. In addition, the directive guarantees that the outlawed instance will not be given in future. We illustrate the `never` directive with an example. For the sake of brevity, we keep the error messages in our examples rather terse. Error message attributes, which we will discuss in Section 2.5, can be used to create a more verbose message that depends on the actual program.

```
never Eq (a -> b): functions cannot be tested for equality
never Num Bool: arithmetic on booleans is not supported
```

These two directives should be placed in a `.type` file[2], which is considered prior to type inference, but after collecting all the type classes and instances in scope. Before type inference, we should check the validity of the directives. Each inconsistency between the directives and the instance declarations results in an error message or warning. For example, the following could be reported at this point.

```
The instance declaration for
   Num Bool at (3,1) in A.hs
is in contradiction with the directive
   never Num Bool defined at (1,1) in A.type
```

We proceed with type inference if no inconsistency is found. If arithmetic on booleans results in a `Num Bool` predicate, we report our special purpose error message. For the definition

```
f x = if x then x+1 else x
```

we simply report that arithmetic on booleans is not supported, and highlight the arithmetical operator `+`. An extreme of concision results in the following type error message.

```
(1,19): arithmetic on booleans is not supported
```

The `never` directive is subject to the same restrictions as any instance declaration in `Haskell 98`: a class name followed by a type constructor and a list of unique type variables (we took the liberty of writing function arrow infix in the example presented earlier). `Haskell 98` does not allow overlapping instances, and similarly we prohibit overlapping `never`s. This ensures that there is always at most one directive which we can use for constructing an error message. If we drop this restriction, then it becomes arbitrary which message is generated.

```
never Eq (Int -> a): message #1
never Eq (b -> Bool): message #2
```

In this example, it is unclear what will be reported for the type class predicate `Eq (Int -> Bool)`. One way to cope with this situation is to require a third directive for the overlapping case, namely `never Eq (Int -> Bool)`. This implies that we can always find and report a most specific directive. Note that in the context of overlapping `never` directives, we have to postpone reporting a violating class predicate since more information about a type variable in this assertion may make a more specific directive a better candidate.

[2] Our convention in this paper is to write all type inference directives on a light gray background.

2.2 The close Directive

With the never directive we can exclude one type from a type class. Similar to this case-by-case directive, we introduce a second type class directive which closes a type class in the sense that no new instances can be defined. As a result of this directive, we can report special error messages for unresolved overloading for a particular type class. A second advantage is that the compiler can assume to know all instances of the given type class since new instances are prohibited, which can be exploited when generating the error message.

One subtle issue is to establish at which point the type class should be closed. This can be either *before* or *after* having considered the instance declarations defined in the module. In this section we discuss only the former. A possible use for the latter is to close the Num type class in Prelude.type so that everybody who imports it may not extend the type class, but the Prelude module itself may specify new instances for Num.

Before we start with type inference, we check for each closed type class that no new instance declarations are provided. A special purpose error message is attached to each close directive, which is reported if we require a non-instance type to resolve overloading for the closed type class. Such a directive can live side by side with a never directive. Since the latter is strictly more informative, we give it precedence over a close directive if we have to create a message. As an example, we close the type class for Integral types, defined at the standard Prelude. Hence, this type class will only have Int and Integer as its members.

```
close Integral: the only instances of Integral are Int and Integer
```

The main advantage of a closed type class is that we know the fixed set of instances. Using this knowledge, we can influence the type inference process. As discussed in the introduction to Section 2, we can reject definitions early on (in case the set of instances for a certain type class is empty) or improve a type variable to a certain type (in case the set of instances is a singleton).

For example, consider a function f :: (Bounded a, Num a) => a -> a. The type class Bounded contains all types that have a minimal and maximal value, including Int and Char. However, Int is the only numeric type among these. Hence, if both Bounded and Num are closed, then we may safely improve f's type to Int -> Int.

The advantages of the close directive would be even higher if we drop the restrictions of Haskell 98, because this directive allows us to reject incorrect usage of a type class early on. We illustrate this with the following example.

```
class Similar a where
  (~=) :: a -> a -> Bool

instance Similar Int where
  (~=) = (==)
```

Assume that the previous code is imported in a module that closes the type class Similar.

```
close Similar: the only instance of Similar is Int.
```

```
f x xs = [x] ~= xs
```

GHC version 6.2 (without extensions) accepts the program above, although an instance for Similar [a] must be provided to resolve overloading. The type inferred for f is

```
f :: forall t. (Similar [t]) => t -> t -> Bool
```

although this type cannot be declared in a type signature for f[3]. This type makes sense: the function f can be used in a different module, on the condition that the missing instance declaration is provided. However, if we intentionally close the type class, then we can generate an error for f at this point.

In this light, the close directive may become a way to moderate the power of some of the language extensions by specifying cases where such generality is not desired. An alternative would be to take Haskell 98 as the starting point, and devise type class directives to selectively overrule some of the language restrictions. For instance, a directive such as general X could tell the compiler not to complain about predicates concerning the type class X that cannot be reduced to head-normal form. Such a directive would allow more programs. In conclusion, type class directives give an easy and flexible way to specify these local extensions and restrictions.

2.3 The disjoint Directive

Our next directive deliberately reduces the set of accepted programs. In other words: the programs will be subjected to a stricter type discipline. The disjoint directive specifies that the instances of two type classes are disjoint, i.e., no type is shared by the two classes. A typical example of two type classes that are intentionally disjoint are Integral and Fractional (see the Haskell 98 Report [5]). If we end up with a type (Fractional a, Integral a) => .... after reduction, then we can immediately generate an error message, which can also explain that "fractions" are necessarily distinct from "integers". Note that without this directive, a context containing these two class assertions is happily accepted by the compiler, although it undoubtedly results in problems when we try to use this function. Acknowledging the senselessness of such a type prevents misunderstanding in the future. A disjoint directive can be defined as follows.

```
disjoint Integral Fractional:
   something which is fractional can never be integral
```

Because Floating is a subclass of Fractional (each type in the former must also be present in the latter), the directive above implies that the type classes Integral and Floating are also disjoint.

[3] In our opinion, it should be possible to include each type inferred by the compiler in the program. In this particular case, GHC suggests to use the Glasgow extensions, although these extensions are not required to infer the type.

In determining that two type classes are disjoint, we base our judgements on the set of instance declarations for these classes, and not on the types implied by the instances. Therefore, we reject instance declarations `C a => C [a]` and `D b => D [b]` if `C` and `D` must be disjoint. A more liberal approach is to consider the set of instance types for `C` and `D`, so that their disjointness depends on other instances given for these type classes.

Take a look at the following example which mixes fractions and integrals.

```
wrong = div 3 8 + 1/2
```

The `disjoint` directive helps to report an appropriate error message for the definition of `wrong`. In fact, without this directive we end up with the type `(Integral a, Fractional a) => a`. `GHC` reports an ambiguous type variable as a result of the monomorphism restriction.

```
Disjoint.hs:1:
 Ambiguous type variable ‘a’ in these top-level constraints:
   ‘Integral a’ arising from use of ‘div’ at Disjoint.hs:1
   ‘Fractional a’ arising from use of ‘/’ at Disjoint.hs:1
 Possible cause: the monomorphism restriction applied
                 to the following:
   wrong :: a (bound at Disjoint.hs:1)
 Probable fix: give these definition(s) an explicit type
               signature
```

Ironically, it is the combination of the two class predicates that makes the defaulting mechanism fail (no numeric type is instance of both classes), which in turn activates the monomorphism restriction rule.

2.4 The `default` Directive

One annoying aspect of overloading is that seemingly innocent programs are in fact ambiguous. For example, `show []` is not well-defined, since the type of the elements must be known (and showable) in order to display the empty list. This problem can only be circumvented by an explicit type annotation. A default declaration is included as special syntax in `Haskell` to help disambiguate overloaded numeric operations. This approach is fairly ad hoc, since it only covers the (standard) numeric type classes. Our example suggests that a programmer could also benefit from a more liberal defaulting strategy, which extends to other type classes. Secondly, the exact rules when defaulting should be applied are unnecessarily complicated (see the Haskell Report [5] for the exact specification). We think that a `default` declaration is nothing but a type class directive, and that it should be placed amongst the other directives instead of being considered part of the programming language. Taking this viewpoint paves the way for other, more complex defaulting strategies as well.

One might wonder at this point why the original design is so conservative. Actually, the caution to apply a general defaulting strategy is justified since it changes the semantics of a program. Inappropriate defaulting unnoticed by a programmer is unquestionably harmful. By specifying `default` directives, the user has full control over the defaulting mechanism. A warning should be raised to inform the programmer that a class predicate has been defaulted. Although we do not advocate defaulting in large programming projects, it is unquestionably useful at times, for instance, to show the result of an evaluated expression in an interpreter. Note that GHC departs from the standard, and applies a more liberal defaulting strategy in combination with the emission of warnings, which works fine in practice.

Take a look at the following datatype definition for a binary tree with values of type `a`.

```
data Tree a = Bin (Tree a) a (Tree a) | Leaf   deriving Show
```

A function to show such a tree can be derived automatically, but it requires a show function for the values stored in the tree. This brings us to the problem: `show Leaf` is of type `String`, but it is ambiguous since the tree that we want to display is polymorphic in the values it contains. We define default directives to remedy this problem.

```
default Num (Int, Integer, Float, Double)
default Show ((), String, Bool, Int)
```

The first directive is similar to the original default declaration, the second defaults predicates concerning the `Show` type class. Obviously, the types which we use as default for a type class must be a member of the class.

Defaulting works as follows. For a given type variable `a`, let `P = {`$\mathtt{X}_1$ `a,` $\mathtt{X}_2$ `a, ..., ` $\mathtt{X}_n$ `a}` be the set of all predicates in the context which contain `a`. Note that only these predicates determine which type may be selected for `a` as a default, and that other predicates in the context are not influenced by this choice. If at least one of the $\mathtt{X}_i$ has a default directive, then we consider the default directives for each of the predicates in P in turn (if they exist). For each of these default directives, we determine the first type which satisfies all of `P`. If this type is the same for all default directives of `P`, then we choose this type for `a`. If the default directives cannot agree on their first choice, then defaulting does not take place.

If default directives are given for a type class and for its subclass, we should check that the two directives are coherent. For instance, `Integral` is a subclass of `Num`, and hence we expect that defaulting `Integral a` and `Num a` has the same result as defaulting only `Integral a`.

Considering defaulting as a directive allows us to design more precise defaulting strategies. For instance, we could have a specific default strategy for showing values of type `Tree a`: this requires some extra information about the instantiated type of the overloaded function `show`.

2.5 Error Message Attributes

The error messages given so far are context-insensitive, but for a real implementation this is not sufficient. Therefore, we use error message attributes, which may hold context dependent information. For example, location information is present in an attribute `@range@` (attributes are written between `@` signs). Due to space limitations we restrict ourselves to an example for the `close` directive.

```
close Show:
   The expression @expr.pp@ at @expr.range@ has the type @expr.gentype@.
   This type is responsible for the introduction of the class predicate
   @errorpredicate@, which is not an instance of @typeclass@ due to
   the close directive defined at @directive.range@.
```

The attributes in the error message are replaced by information from the actual program. For instance, `@directive.range@` is changed into the location where the `close` directive is defined, and `@expr.pp@` is unfolded to a pretty printed version of the expression responsible for the introduction of the erroneous predicate. We can devise a list of attributes for each directive. These lists differ: in case of the `disjoint` directive, for instance, we want to refer to the origin of both class predicates that contradict.

A complicating factor is that the predicate at fault may not be the predicate which was introduced. Reducing the predicate `Eq [(String, Int -> Int)]` will eventually lead to `Eq (Int -> Int)`. We would like to communicate this reasoning to the programmer as well, perhaps by showing some of the reduction steps.

3 Implementation

Type inference for `Haskell` is based on the Hindley-Milner [9] type system. Along the way, types are unified, and when unification fails, an error message is reported. An alternative method is to collect *equality constraints* to encapsulate the relation between various types of parts of the program, and solve these afterwards. Type inference for `Haskell 98` is widely studied and well understood: we suggest “Typing Haskell in Haskell” [10] for an in-depth study. To support overloading, we extend the Hindley-Milner system and propagate sets of type class predicates. For each binding group we perform *context reduction*, which serves to simplify sets of type class predicates, and to report predicates that cannot be resolved. Context reduction can be divided into three phases.

In the first phase, the predicates are simplified by using the instance declarations until they are in head-normal form, that is, of the form `X (a` $t_1 \ldots t_n$`)` where `a` is a type variable. Typically, we get predicates where n is zero. Predicates that cannot be reduced to this form are reported as incorrect. For instance, `Eq [a]` can be simplified to `Eq a`, the predicate `Eq Int` can be removed altogether, and an error message is created for `Num Bool`.

Duplicate predicates are removed in the second phase, and we use the type class hierarchy to eliminate predicates entailed by assertions about a subclass. For instance, `Eq` is the superclass of `Ord`, and, hence, `Ord a` implies `Eq a`. If we have both predicates, then `Eq a` can be safely discarded.

In the final phase, we report predicates that give rise to an ambiguous type. For instance, the type `(Read a, Show a) => String -> String`, inferred for

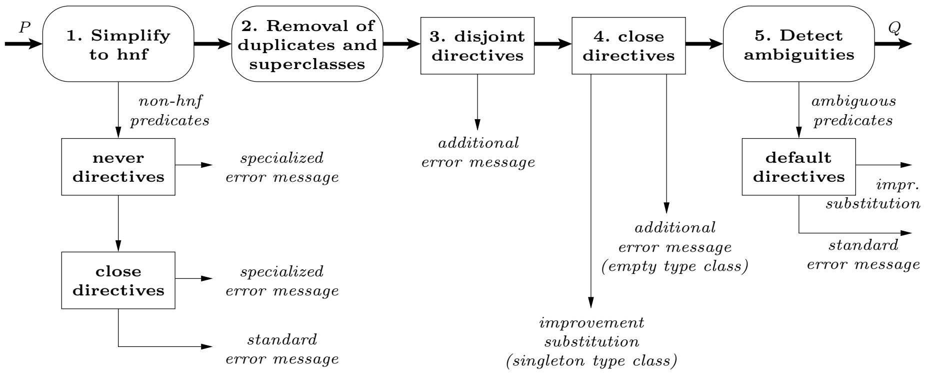


Fig. 1. Context reduction with type class directives for `Haskell 98`

the famous `show . read` example, is ambiguous since there is no way we can determine the type of `a`, which is needed to resolve overloading. Note that we can postpone dealing with predicates containing monomorphic type variables.

We continue with a discussion on how the four type class directives can be incorporated into context reduction. Figure 1 gives an overview. The first, second, and fifth step correspond to the three phases of the traditional approach. The thickened horizontal line reflects the main process in which the set of predicates P is transformed into a set of predicates Q.

The first modification concerns the predicates that cannot be simplified to head-normal form. If a `never` or `close` directive is specified for such a predicate, then we report the specialized error message that was declared with the directive. Otherwise, we proceed as usual and report a standard error message.

The `disjoint` directives and closed type classes are handled after removal of duplicates and super-classes. At this point, the predicates to consider are in head-normal form. A `disjoint` directive creates an error message for a pair of predicates that is in conflict. Similarly, if we can conclude from the closed type classes that no type meets all the requirements imposed by the predicates for a given type variable, then an error message is constructed. If we, on the other hand, discover that there is a single type which meets the restrictions, then we assume this type variable to be that particular type. This is an improvement substitution [8]. Because we consider all predicates involving a certain type variable at once, the restrictions of `Haskell 98` guarantee that improvement substitutions cannot lead to more reduction steps.

Finally, we try to avoid reporting ambiguous predicates by inspecting the given `default` directives, as described in Section 2.4. Defaulting a type variable by applying these directives results again in an improvement substitution.

The use of improvement substitutions leads to more programs being accepted, while others are now rejected. The sum of their effects can be hard to predict, and not something to rely on in large programming projects. Even without improvement substitutions, the `never`, `close`, and `disjoint` directives can be quite useful.

4 Generalization of Directives

In this section, we sketch a generalization of the first three directives described in Section 2. This part has not been implemented, but gives an idea how far we expect type class directives can go, and what benefits accrue.

Essentially, a type class describes a (possibly infinite) set of types, and most of the proposed directives can be understood as constraints over such sets. In fact, they describe invariants on these sets of types, enriching the means of specification in Haskell, which is limited to membership of a type class (instance declaration), and a subset relation between type classes (class declaration).

We present a small language to specify invariants on the class system. The language is very expressive, and it may be necessary to restrict its power for reasons of efficiency and decidability, depending on the type (class) system to which it is added.

$$\begin{array}{lll}
\mathit{Constraint} & ::= & \mathit{Type}\ \mathit{EltOp}\ \mathit{Set} \mid \mathit{Set}\ \mathit{SetOp}\ \mathit{Set} \\
\mathit{Set} & ::= & \mathit{BinOp}\ \mathit{Set}\ \mathit{Set} \mid \mathit{SetLiteral} \mid \mathit{Class} \\
\mathit{SetLiteral} & ::= & \{\} \mid \{\ \mathit{Type}\ (,\ \mathit{Type})^*\ \} \\
\mathit{EltOp} & ::= & \texttt{isin} \mid \texttt{isnotin} \\
\mathit{SetOp} & ::= & <= \mid == \mid >= \\
\mathit{BinOp} & ::= & \texttt{intersect} \mid \texttt{union} \mid \texttt{difference}
\end{array}$$

Each constraint can be followed by an error message. If necessary, syntactic sugar can be introduced for special directives such as `never` and `disjoint`.

```
Monad  ==  {Maybe, [], IO}: only Maybe, [], and IO are monads today.
Read   ==  Show
intersect Egglayer Mammal  <=  {Platypus}
```

The first example directive prevents new instances for the `Monad` class, while `Read == Show` demands that in this module (and all modules that import it) the instances for `Show` and `Read` are the same. A nice example of an invariant is the third directive, which states that only the duckbilled platypus can be both in the type class for egg layers and in `Mammal`. This directive might be used to obtain an improvement substitution (as discussed in Section 2): if we have the predicates `Mammal a` and `Egglayer a`, then `a` must be `Platypus`. This example shows that the directives can be used to describe domain specific invariants over class hierarchies.

5 Specialized Type Rules

In an earlier paper [6], we introduced specialized type rules to improve type error messages. The main benefits of this facility are that for certain collections of expressions, the programmer can

1. change the order in which unifications are performed, and
2. provide special type error messages, which can exploit this knowledge,
3. with the guarantee that the underlying type system is unchanged.

This facility is especially useful for domain specific extensions to a base language (such as Haskell), because the developer of such a language can now specify error messages which refer to concepts in the domain to replace the error messages phrased in terms of the underlying language. We present an extension of these type rules which allows class assertions among the equality constraints to deal with overloading. This extension has been implemented in the Helium compiler [7].

Consider the function spread, which returns the difference between the smallest and largest value of a list, and a specialized type rule for this function, given that it is applied to one argument.

```
spread :: (Ord a, Num a) => [a] -> a
spread xs = maximum xs - minimum xs
```

```
    xs :: t1;
--------------------
  spread xs :: t2;

t1 == [t3]: @xs.pp@ must be a list
t3 == t2: @expr.pp@ should return a value of type @t3@
Eq t2: @t2@ is not an instance of Eq, let alone Ord or Num
Ord t2: @t2@ should have a linear ordering imposed on it
Num t2: @t2@ should allow numerical operations
```

A specialized type rule consists of a deduction rule, followed by a list of constraints. In the consequent of the deduction rule, `spread xs :: t2`, we describe the expressions of interest. Since xs also occurs above the line, it is considered to be a meta-variable which functions as a placeholder for an arbitrary expression (with a type to which we can refer as t1).

The deduction rule is followed by a number of constraints. The first of these states that the type t1 is a list type, with elements of type t3 (t3 is still unconstrained at this point). The next equality constraint constrains the type t3 to be the same as the type of spread xs. Note that the listed constraints are verified from top to bottom, and this fact can be exploited to yield very precise error messages.

For class assertions we can also exploit the order of specification. Although membership of Ord or Num implies membership of Eq, we can check the latter first, and give a more precise error message in case it fails. Only when Eq t2 holds, do we consider the class assertions Ord t2 and Num t2. Note that the assertion Eq t2 does not change the validity of the rule.

Context reduction takes place after having solved the unification constraints. This implies that listing class assertions before the unification constraints makes little sense, and only serves to confuse people. Therefore, we disallow this.

Equality constraints can be moved into the deduction rule, in which case it is given a standard error message. This facility is essential for practical reasons: it should be possible to only list those constraints for which we expect special treatment. Similarly, we may move a class assertion into the deduction rule. Notwithstanding, this assertion is checked *after* all the unification constraints.

All specialized type rules are automatically examined in that they leave the underlying type system unchanged. This is an essential feature, since a mistake is easily made in these rules. We compare the set of constraints implied by the specialized type rule (say S) with the set that would have been generated by the standard inference rules (say T). A type rule is only accepted if S equals T under an entailment relation. This relation is a combination of entailment for class predicates and for equality constraints.

6 Related Work

A number of papers address the problem of improving the type error messages produced by compilers for functional programming languages. Several approaches to improve on the quality of error messages have been suggested. One of the first proposals is by Wand [11], who suggests to modify the unification algorithm such that it keeps track of reasons for deductions about the types of type variables. Many papers that followed elaborate on his idea. At the same time, Walz and Johnson [12] suggested to use maximum flow techniques to isolate and report the most likely source of an inconsistency. This can be considered the first heuristic-based system.

Recently, Yang, Michaelson, and Trinder [2] have reported on a human-like type inference algorithm, which mimics the manner in which an expert would explain a type inconsistency. This algorithm produces explanations in plain English for inferred (polymorphic) types. McAdam [1] suggested to use unification of types modulo linear isomorphism to automatically repair ill-typed programs. Haack and Wells [3] compute a minimal set of program locations (a type error slice) that contribute to a type inconsistency. The `Chameleon` type debugger is developed by Stuckey, Sulzmann, and Wazny [4], and helps to locate type errors in an interactive way.

Elements of our work can be found in earlier papers: closed type classes were mentioned by Shields and Peyton Jones [13], while the concepts of disjoint type classes and type class complements were considered by Glynn et al. [14]. Type class directives lead to improvement substitutions which are part of the framework as laid down by Jones [8]. All these efforts are focused on the type system, while we concentrate on giving good feedback by adding high-level support to compilers via compiler directives. Moreover, we generalize these directives to invariants over type classes.

A different approach to tackle language extensions is followed in the `DrScheme` project [15], which introduces language levels (syntactically restricted variants) to gradually become familiar with a language.

7 Conclusion and Future Work

This paper offers a solution to compensate the effect that the introduction of overloading (type classes) has on the quality of reported error messages. In general, the types of overloaded functions are less restrictive, and therefore some errors may remain undetected. At the same time, a different kind of error message is produced for unresolved overloading, and these errors are often hard to interpret.

To remedy the loss of clarity in error messages, a number of type class directives have been proposed, and we have indicated how context reduction can be extended to incorporate these directives. The directives have the following advantages.

- Tailor-made, domain-specific error messages can be reported for special cases.
- Functions for which we infer a type scheme with a suspicious class context can be detected (and rejected) at an early stage.
- An effective defaulting mechanism assists to disambiguate overloading.
- Type classes with a limited set of instances help to improve and simplify types.

Furthermore, we have added type class predicates to the specialized type rules, and the soundness check has been generalized accordingly.

We see several possible directions for future research. A small language to specify invariants on the class system (see Section 4) seems to be a promising direction, and this requires further investigation. The more expressive such a language becomes, the more need there is for some form of analysis of these invariants. Another direction is to explore directives for a number of the proposed extensions to the type class system [16], and to come up with new directives to alleviate the problems introduced by these extensions.

In both these cases, we see the need for a formal approach, so that the effects of our (more general) directives on our (extended) language can be fully understood. The constraint handling rules (used in [14]) are a good starting point for such an approach.

Furthermore, we would like to lift our ideas on directives to Jones' theory of qualified types [8]. As a result, we want to look for directives to support other *qualifiers* that fit in his framework. Besides the type class predicates discussed in this paper, we plan to investigate predicates for extensible records, and for subtyping.

Acknowledgements

We like to thank Daan Leijen, Alexey Rodriguez Yakushev, Doaitse Swierstra, and the anonymous reviewers for their suggestions and comments on this paper.

References

1. McAdam, B.: How to repair type errors automatically. In: 3rd Scottish Workshop on Functional Programming, Stirling, U.K. (2001) 121–135
2. Yang, J., Michaelson, G., Trinder, P.: Explaining polymorphic types. The Computer Journal **45** (2002) 436–452

3. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. In: Proceedings of the 12th European Symposium on Programming. (2003) 284–301
4. Stuckey, P., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Haskell Workshop, New York, ACM Press (2003) 72 – 83
5. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003) http://www.haskell.org/onlinereport/.
6. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Eighth ACM Sigplan International Conference on Functional Programming, New York, ACM Press (2003) 3 – 13
7. Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for learning Haskell. In: ACM Sigplan 2003 Haskell Workshop, New York, ACM Press (2003) 62 – 71 http://www.cs.uu.nl/helium.
8. Jones, M.P.: Simplifying and improving qualified types. In: International Conference on Functional Programming Languages and Computer Architecture. (1995) 160–169
9. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
10. Jones, M.P.: Typing Haskell in Haskell. In: Haskell Workshop. (1999)
11. Wand, M.: Finding the source of type errors. In: Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, FL (1986) 38–43
12. Walz, J.A., Johnson, G.F.: A maximum flow approach to anomaly isolation in unification-based incremental type inference. In: Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, FL (1986) 44–57
13. Shields, M., Peyton Jones, S.: Object-oriented style overloading for Haskell. In: Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01). (2001)
14. Glynn, K., Stuckey, P., Sulzmann, M.: Type classes and constraint handling rules. In: First Workshop on Rule-Based Constraint Reasoning and Programming. (2000)
15. Findler, R.B., Clements, J., Cormac Flanagan, M.F., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A programming environment for Scheme. Journal of Functional Programming **12** (2002) 159–182
16. Peyton Jones, S., Jones, M., Meijer, E.: Type classes: an exploration of the design space. In: Haskell Workshop. (1997)

Author Index