

Jean-Christophe Filliâtre  
Christine Paulin-Mohring  
Benjamin Werner (Eds.)

LNCS 3839

# Types for Proofs and Programs

International Workshop, TYPES 2004  
Jouy-en-Josas, France, December 2004  
Revised Selected Papers

 Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*New York University, NY, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Moshe Y. Vardi

*Rice University, Houston, TX, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Jean-Christophe Filliâtre  
Christine Paulin-Mohring  
Benjamin Werner (Eds.)

# Types for Proofs and Programs

International Workshop, TYPES 2004  
Jouy-en-Josas, France, December 15-18, 2004  
Revised Selected Papers



Springer

Volume Editors

Jean-Christophe Filliâtre  
Christine Paulin-Mohring  
LRI, Inria Futurs, LIX, Université Paris Sud  
LRI Bât 490, 91405 Orsay Cedex, France  
E-mail: {jean-christophe.filliatre,christine.paulin}@lri.fr

Benjamin Werner  
Inria Futurs, LIX, LRI, Laboratoire d'Informatique (LIX)  
École Polytechnique  
91128 Palaiseau Cedex, France  
E-mail: werner@lix.polytechnique.fr

Library of Congress Control Number: 2005938814

CR Subject Classification (1998): F.3.1, F.4.1, D.3.3, I.2.3

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN           0302-9743  
ISBN-10       3-540-31428-8 Springer Berlin Heidelberg New York  
ISBN-13       978-3-540-31428-8 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

[springer.com](http://springer.com)

© Springer-Verlag Berlin Heidelberg 2006  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India  
Printed on acid-free paper   SPIN: 11617990   06/3142   5 4 3 2 1 0

# Preface

These proceedings contain a selection of refereed papers presented at or related to the Annual Workshop of the TYPES project (EU coordination action 510996), which was held from December 15 to 18, 2004, in Jouy en Josas, France.

The topic of this workshop was formal reasoning and computer programming based on type theory: languages and computerized tools for reasoning, and applications in several domains such as analysis of programming languages, certified software, formalization of mathematics and mathematics education.

The workshop was attended by more than 100 researchers and proposed more than 50 presentations. Out of 33 post-workshop submitted papers, 17 were selected after a reviewing process. The final decisions were made by the editors.

This workshop followed a series of meetings of the TYPES working group funded by the European Union (IST project 29001, ESPRIT Working Group 21900, ESPRIT BRA 6435). The proceedings of these workshop were published in the LNCS series:

**TYPES 93** Nijmegen, The Netherlands, LNCS 806

**TYPES 94** Båstad, Sweden, LNCS 996

**TYPES 95** Torino, Italy, LNCS 1158

**TYPES 96** Aussois, France, LNCS 1512

**TYPES 98** Kloster Irsee, Germany , LNCS 1657

**TYPES 2000** Durham, United Kingdom, LNCS 2277

**TYPES 2002** Berg en Dal, The Netherlands, LNCS 2646

**TYPES 2003** Torino, Italy, LNCS 3085

ESPRIT BRA 6453 was a continuation of ESPRIT Action 3245, Logical Frameworks: Design, Implementation and Experiments. Proceedings for annual meetings under this action were published by Cambridge University Press in the books *Logical Frameworks* and *Logical Environments*, edited by G. Huet and G. Plotkin.

We are very grateful to INRIA for supporting the TYPES meeting. We especially want to thank Catherine Girard and Catherine Moreau (organization), Chantal Girodon (registration) and Laurent Steff (technical support). Hugo Herbelin was in charge of the programme in the organizing committee. Finally, Marie-Carol Lopes took care of the organization of the post-workshop proceedings and carefully prepared the final composition of the volume.

November 2005

Jean-Christophe Filliâtre  
Christine Paulin-Mohring  
Benjamin Werner

# Organization

## Referees

P. Aczel	N. Gambino	D. Miller
M. Baaz	H. Geuvers	A. Miquel
C. Ballarin	B. Grégoire	J-F. Monin
B. Barras	P. Hancock	Z. Petric
S. Berardi	D. Hendricks	D. Pichardie
S. Berghofer	O. Hermant	R. Pollack
Y. Bertot	J. Hickey	L. Pottier
F. Blanqui	F. Honsell	C. Raffalli
S. Boulmé	G. Klein	E. Ritter
A. Bove	S. Kremer	C. Sacerdoti Coen
V. Capretta	Z. Luo	P. Sobocinski
T. Coquand	A. Mahboubi	B. Spitters
L. Cruz-Filipe	R. Matthes	M. Strecker
J. Despeyroux	M. Mayero	L. Thery
P. Di Gianantonio	C. McBride	F. Wiedjick
L. Dixon	P-A. Melliès	K. Wirt
G. Dowek	M. Miculan	R. Zumkeller
N. Gahni	R. Milewski	

# Table of Contents

Formalized Metatheory with Terms Represented by an Indexed Family of Types <i>Robin Adams</i> .....	1
A Content Based Mathematical Search Engine: Whelp <i>Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, Stefano Zacchiroli</i> .....	17
A Machine-Checked Formalization of the Random Oracle Model <i>Gilles Barthe, Sabrina Tarento</i> .....	33
Extracting a Normalization Algorithm in Isabelle/HOL <i>Stefan Berghofer</i> .....	50
A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis <i>Yves Bertot, Benjamin Grégoire, Xavier Leroy</i> .....	66
Formalising Bitonic Sort in Type Theory <i>Ana Bove, Thierry Coquand</i> .....	82
A Semi-reflexive Tactic for (Sub-)Equational Reasoning <i>Claudio Sacerdoti Coen</i> .....	98
A Uniform and Certified Approach for Two Static Analyses <i>Solange Coupet-Grimal, William Delobel</i> .....	115
Solving Two Problems in General Topology Via Types <i>Adam Grabowski</i> .....	138
A Tool for Automated Theorem Proving in Agda <i>Fredrik Lindblad, Marcin Benke</i> .....	154
Surreal Numbers in Coq <i>Lionel Elie Mamane</i> .....	170
A Few Constructions on Constructors <i>Conor McBride, Healfdene Goguen, James McKinna</i> .....	186
Tactic-Based Optimized Compilation of Functional Programs <i>Thomas Meyer, Burkhart Wolff</i> .....	201

Interfaces as Games, Programs as Strategies <i>Markus Michelbrink</i> .....	215
$\lambda$ Z: Zermelo's Set Theory as a PTS with 4 Sorts <i>Alexandre Miquel</i> .....	232
Exploring the Regular Tree Types <i>Peter Morris, Thorsten Altenkirch, Conor McBride</i> .....	252
On Constructive Existence <i>Michel Parigot</i> .....	268
<b>Author Index</b> .....	275



# Formalized Metatheory with Terms Represented by an Indexed Family of Types

Robin Adams

Royal Holloway, University of London  
robin@cs.rhul.ac.uk

**Abstract.** It is possible to represent the terms of a syntax with binding constructors by a family of types, indexed by the free variables that may occur. This approach has been used several times for the study of syntax and substitution, but never for the formalization of the metatheory of a typing system. We describe a recent formalization of the metatheory of Pure Type Systems in Coq as an example of such a formalization. In general, careful thought is required as to how each definition and theorem should be stated, usually in an unfamiliar ‘big-step’ form; but, once the correct form has been found, the proofs are very elegant and direct.

## 1 Introduction

In [1], Bellegarde and Hook show how the terms of a language with binding constructors can be represented as a *nested datatype* — a type constructor that takes types (including possibly its own values) as arguments. This idea has since been used several times for the study of the syntax of such languages, for example in Altenkirch and Reus [2], and Bird and Paterson [3]. However, to the best of the author’s knowledge, it has never been used in a formalization of the metatheory of a formal system.

We present here a formalization in Coq of the metatheory of Pure Type Systems (PTSs) using this representation for the set of terms. We prove all of the results about arbitrary PTSs given in Barendregt [4], including Subject Reduction and Uniqueness of Types for functional PTSs. The formalization also includes van Benthem Jutting’s proof of Strengthening [5].

There have been several formalizations of the metatheory of formal systems in the past, two of the largest being McKinna and Pollack [6] and Barras [7]; and so we shall be able to compare the strengths and weaknesses of this approach with those of the previous.

The indexed family approach proves to have quite limited expressive power. We cannot define all the operations nor state all the results in the form we are used to. Careful thought was often needed as to what form a definition or theorem could take. In general, it was found that operations involving all variables simultaneously were easy to represent in this formalization, while those involving single variables were difficult to represent. For example, we can define the operation of substituting for every variable simultaneously, but not that of

substituting a given term for an arbitrary single variable. The author found himself calling operations of the first kind ‘big-step’ operations, and those of the second kind ‘small-step’. These are as good names as any, and we shall continue to use them.

To set against this, it was found that, once the correct form for each definition and theorem had been arrived at, the proofs themselves were simple, short, elegant and easily constructed. We find ourselves spending much less time on technical details than in most formalizations of metatheoretic work. Perhaps most importantly, those technicalities that we do have to deal with have a more type-theoretic flavour: the objects with which one deals are those one would expect to find in the metatheory of a type theory, rather than low-level aspects of the mechanisms of the formalization.

In particular, each specific instance of a small-step operation we need (such as the substitution involved in  $\beta$ -reduction) is definable as a special case of a big-step operation. The results that we need about it are likewise derivable as special cases of results about the big-step operations.

There were two maxims that, at several points in this work, it proved wise to follow, and which would seem to be more widely applicable to the formalization of mathematics in general. They can be stated briefly as: *favour recursive definitions over inductive ones*, and *avoid arithmetic in types*. We discuss them in Section 2. We proceed in Section 3 to a description of the formalization itself, showing how the two maxims are applied, and showing the big-step form of each definition and theorem.

For those who wish to examine the formalization, the source code and documentation is available at <http://www.cs.rhul.ac.uk/~robin/coqPTS>.

## 2 Maxims for Formalization

There were three general principles that it proved wise to follow in this work. One — that ‘big-step’ definitions should be preferred to ‘small-step’ — is peculiar to this work, and we shall discuss it in Section 3. We wish here to discuss the other two maxims, which should be more generally applicable to other formalizations.

### 2.1 Recursive Definitions Versus Inductive Families

Our first maxim is:

When defining a family of types  $F : I \rightarrow \mathbf{Set}$  over an inductive type  $I$ , if possible define  $F$  by recursion over  $I$ , rather than as an inductive family

or more briefly, *favour recursive definitions over inductive ones*.

The difficulties of using inductive families are well known among the Coq community. The standard example is the family  $\mathcal{V}_n$  of *vectors* of length  $n$  (over a given type  $A$ ). We can define this either as an inductive family, with constructors:

$$\frac{}{\langle \rangle : \mathcal{V}_0} \qquad \frac{a : A \quad v : \mathcal{V}_n}{(a :: v) : \mathcal{V}_{n+1}}$$

or by recursion on the index  $n$ , as follows:

$$\mathcal{V}_0 = \mathbf{1} \quad \mathcal{V}_{n+1} = A \times \mathcal{V}_n$$

(Here,  $\mathbf{1}$  is the type with a single canonical object, called `unit` in the Coq library, and  $\times$  of course denotes the non-dependent product of two types.)

The practical difference between these two definitions lies in how easy it is to deduce information about an object of type  $\mathcal{V}_t$  from the shape of the term  $t : \mathbb{N}$ . In particular, we frequently want to use the fact that a term  $s : \mathcal{V}_0$  must be equal to  $\langle \rangle$ , and that a term  $s : \mathcal{V}_{t+1}$  must have the form  $a :: v$  for some  $a : A$  and  $v : \mathcal{V}_t$ . This can be deduced immediately if we are using the recursive definition, but not with the inductive definition.

There is a standard technical trick for overcoming this difficulty that is known as folklore among the Coq community. It appears unattributed in Letouzey's message [8]. To the author's knowledge, this technique has never before been put into print, and Letouzey may well be its inventor.

It can be summarised as follows:

- Define a function which should be the identity function on the inductive family.
- Prove that the function is indeed the identity function.
- Deduce from this fact theorems allowing case analysis on the objects of the inductive family.

In the case of  $\mathcal{V}_n$ , we provide ourselves with the destructors `head` :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{V}_n \rightarrow A$  and `tail` :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{V}_n \rightarrow \mathcal{V}_n$ :

$$\text{head } n (a :: v) = a \quad \text{tail } n (a :: v) = v$$

(These are two instances where case analysis on  $\mathcal{V}_n$  can be applied successfully.) We can then define our 'pseudo-identity' function `pseudoid` :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathcal{V}_n \rightarrow \mathcal{V}_n$  by recursion *on*  $n$  as follows:

$$\begin{aligned} \text{pseudoid } 0 v &= \langle \rangle & (v : \mathcal{V}_0) \\ \text{pseudoid } n + 1 v &= (\text{head } n v) :: (\text{tail } n v) & (v : \mathcal{V}_{n+1}) \end{aligned}$$

We can prove the following result by induction *on*  $v$ :

$$(\forall n : \mathbb{N})(\forall v : \mathcal{V}_n) \text{pseudoid } n v = v \quad . \quad (1)$$

From this, the following case analysis theorems can be easily deduced:

$$(\forall v : \mathcal{V}_0) v = \langle \rangle \quad (2)$$

$$(\forall n : \mathbb{N})(\forall v : \mathcal{V}_{n+1}) v = (\text{head } n v) :: (\text{tail } n v) \quad (3)$$

One way of viewing this construction is as *building a bijection between the inductively defined family and the recursively defined family*. More precisely, we have built a bijection between  $\mathcal{V}_0$  and  $\mathbf{1}$ , and between  $\mathcal{V}_{n+1}$  and  $A \times \mathcal{V}_n$ . The

constructors form one half of the bijection: the ‘cons’ constructor  $::$  is a (Curried) mapping  $A \times \mathcal{V}_n \rightarrow \mathcal{V}_{n+1}$ , and  $\langle \rangle$  can be seen as a mapping  $\mathbf{1} \rightarrow \mathcal{V}_0$ . Our destructors form the other half: **head** and **tail** together give a mapping  $\mathcal{V}_{n+1} \rightarrow A \times \mathcal{V}_n$ , and we take the trivial mapping  $\mathcal{V}_0 \rightarrow \mathbf{1}$ .

The function **pseudoid** is the composition of these two halves: **pseudoid** 0 is the composition

$$\mathcal{V}_0 \rightarrow \mathbf{1} \xrightarrow{\langle \rangle} \mathcal{V}_0 ,$$

and **pseudoid**  $n + 1$  is the composition

$$\mathcal{V}_{n+1} \xrightarrow{\langle \mathbf{head}, \mathbf{tail} \rangle} A \times \mathcal{V}_n \xrightarrow{::} \mathcal{V}_{n+1} .$$

Our theorem (1) then verifies that these compositions do produce the identity mapping; that is, that **cons** and **emp** are left inverses to  $\langle \mathbf{head}, \mathbf{tail} \rangle$  and the trivial mapping, respectively. (That they are also right inverses is immediate from the definitions of **head** and **tail**.) In theorems (2) and (3), we then use the fact that case analysis is possible on  $\mathbf{1}$  and  $A \times \mathcal{V}_n$  to prove it is possible on  $\mathcal{V}_0$  and  $\mathcal{V}_{n+1}$ .

Conversely, we can see intuitively that any way of performing case analysis on the elements of  $\mathcal{V}_n$  would yield appropriate functions  $\mathcal{V}_0 \rightarrow \mathbf{1}$  and  $\mathcal{V}_{n+1} \rightarrow A \times \mathcal{V}_n$ . So we make the following, as yet rather imprecise, conjecture:

Case analysis is possible on the objects of an inductively defined family of types if and only if the family is isomorphic to a recursively defined family.

## 2.2 Arithmetic Within Types

Our second maxim is:

When using a family of types indexed by **nat**, make sure that the index term never involves **plus** or **times**

or, more briefly, *avoid arithmetic within types*.

Let us continue with our example of the family  $\mathcal{V}_n$  of types of vectors that we have introduced. A natural operation to define is the action of *appending* two vectors:

$$\frac{u : \mathcal{V}_m \quad v : \mathcal{V}_n}{\hat{u}v : \mathcal{V}_{m+n}}$$

$$\langle \rangle \hat{v} = v, \quad (a :: u) \hat{v} = a :: (\hat{u}v)$$

We now try to prove that this operation is associative:

$$\hat{u}(\hat{v}w) = (\hat{u}v)\hat{w}$$

We have a problem. The left-hand side of this equation has type  $\mathcal{V}_{m+(n+p)}$ , while the right-hand side has type  $\mathcal{V}_{(m+n)+p}$ . These two types, while provably equal, are not convertible, so the proposition as we have written it is not well-formed.

We are trying to treat two terms with different types as if they had the same type.

We can also meet the converse problem, where we require one term to have two different types within the same formula; say type  $\mathcal{V}_{m+S_n}$  at one point and type  $\mathcal{V}_{S_{m+n}}$  at another.

Should these two maxims ever conflict, the second maxim should take priority over the first. The first involves a known quantity of work: once the case analysis lemmas have been proven by the known method described above, then we can forget about the fact the family was defined inductively. Our two maxims conflicted at only one point in this formalization: the definition of the family of types of *strings*. We shall say more about this in Section 3.3.

### 3 Description of the Formalization

We proceed to a description of the formalization of the metatheory of PTSs. Most of the formalization shall not be described in any detail; we shall concentrate instead on the parts that are novel to this formalization, particularly those definitions and theorems which need to be stated in an unfamiliar form owing to the fact that we are working with the indexed family representation of terms. A detailed description of the formalization can be found at [9].

We have already discussed two themes that occurred several times in this work: the preference for recursive definitions of families of types over inductive ones, and the need to avoid arithmetic within types. A third is that, when terms are defined as an indexed family, it is easier to define operations and prove theorems that deal with all variables simultaneously, than those that deal with a single variable. For example, it is easier to define the operation of substituting for every variable simultaneously, than that of substituting for a single given variable. We shall give the name of ‘big-step’ operations and theorems to the first kind, and ‘small-step’ to the second.

Our definitions of replacement (substitution of variables for variables), substitution, and the subcontext relation all get big-step definitions, and we also find ourselves needing a *satisfaction* relation, which can be seen as a big-step version of the typing relation. We found it easiest to base the definition of reduction and conversion on parallel reduction, rather than one-step reduction; this again could possibly be seen as a big-step/small-step distinction. (In this choice, we follow McKinna and Pollack [6].) The Weakening, Substitution, Subject Reduction and Context Conversion results were all found to be easier to state and prove in a big-step form. The small-step instances of these operations, relations and results that we later need can all be derived as special cases of the big-step forms.

#### 3.1 Grammar

**Terms as a Nested Datatype.** The idea of representing terms as a nested datatype first appeared in Bellegarde and Hook [1]. It has been used several times for the study of syntactic properties and operations, such as substitution

and  $\beta$ -reduction; see for example Altenkirch and Reus [2] or Bird and Paterson [3]; the latter even deals with the simply-typed lambda calculus. However, this representation of syntax seems never before to have been used in the metatheory of a typing system where the typing judgements are given by a set of rules of deduction.

We wish to represent, in some type theory, the terms of some formal system whose syntax involves one or more binding constructor. Rather than defining one type whose objects are to represent these terms, we define a family of types  $\mathcal{T}_V$  for every type  $V$  in some universe. The objects of type  $\mathcal{T}_V$  represent those terms that can be formed using the objects of type  $V$  as free variables.

For example, suppose we wish to represent the terms of the untyped lambda calculus. Let us assume we have, for every type  $X$ , a type  $X_\perp$  consisting of a copy  $\uparrow x$  of each  $x : X$ , together with one extra object,  $\perp$ . (In Coq, this would be the type `option X` provided by the Coq library.) We can then represent the terms of the untyped lambda calculus by the inductive family  $\mathcal{T}_V$  whose constructors are as follows.

$$\frac{x : V}{\mathbf{var} \ x : \mathcal{T}_V} \quad \frac{M : \mathcal{T}_{V_\perp}}{\lambda M : \mathcal{T}_V} \quad \frac{M : \mathcal{T}_V \quad N : \mathcal{T}_V}{MN : \mathcal{T}_V}$$

The constructor  $\lambda$  takes a term  $M$  whose free variables are taken from  $V_\perp$ , binds the variable  $\perp$ , and returns a term  $\lambda M$  whose free variables are taken from  $V$ .

The definition of substitution takes the following form. We aim to define, for every term  $M : \mathcal{T}_U$  and every *substitution function*  $\sigma : U \rightarrow \mathcal{T}_V$ , the term  $M[\sigma] : \mathcal{T}_V$ , the result of substituting for *each* variable  $u : U$  the term  $\sigma(u) : \mathcal{T}_V$ . The definition that we would naturally write down is as follows.

$$\begin{aligned} x[\sigma] &\equiv \sigma(x) \\ (MN)[\sigma] &\equiv M[\sigma]N[\sigma] \\ (\lambda M)[\sigma] &\equiv \lambda M \left[ \begin{array}{l} \perp \mapsto \perp \\ \uparrow x \mapsto \sigma(x)[y \mapsto \uparrow y] \end{array} \right] \end{aligned}$$

(Here and henceforth, we are omitting the constructor `var`.)

It is possible to prove that this recursion terminates. However, this definition cannot be made directly in Coq, as it is not a definition by structural recursion. But we can define directly the special case where  $\sigma$  always returns a variable. We thus define the operation of *replacement*; given a term  $M : \mathcal{T}_U$  and a *renaming function*  $\rho : U \rightarrow V$ , we define the term  $M\{\rho\} : \mathcal{T}_V$ , the result of replacing each variable  $u : U$  with  $\rho(u) : V$ , thus:

$$\begin{aligned} x\{\rho\} &\equiv \rho(x) \\ (MN)\{\rho\} &\equiv M\{\rho\}N\{\rho\} \\ (\lambda M)\{\rho\} &\equiv \lambda M \left\{ \begin{array}{l} \perp \mapsto \perp \\ \uparrow x \mapsto \uparrow \rho(x) \end{array} \right\} \end{aligned}$$

Substitution can then be defined as follows: for  $\sigma : U \rightarrow \mathcal{T}_V$ ,

$$\begin{aligned} x[\sigma] &\equiv \sigma(x) \\ (MN)[\sigma] &\equiv M[\sigma]N[\sigma] \\ (\lambda M)[\sigma] &\equiv \lambda M \left[ \begin{array}{l} \perp \mapsto \perp \\ \uparrow x \mapsto \sigma(x)\{y \mapsto \uparrow y\} \end{array} \right] \end{aligned}$$

Altenkirch and Reus [2] show how the type operator  $\mathcal{T}$ , the constructor **var**, and the substitution operation  $[\ ]$  form a *Kleisli triple*, a concept closely related to that of a monad.

Bird and Paterson [3] give the monadic structure explicitly. The substitution operation can be split into a mapping  $\mathcal{T}_U \rightarrow \mathcal{T}_{\mathcal{T}_V}$ , followed by a *folding* operation  $\mathcal{T}_{\mathcal{T}_V} \rightarrow \mathcal{T}_V$ . The type operator  $\mathcal{T}$ , together with the constructor **var**, the replacement operation  $\{ \}$  and this folding operation, form a monad.

**Terms as an Indexed Family.** For our formalization, we modify this construction slightly. Rather than allowing any type  $V : \mathbf{Set}$  to be used as the type of variables, we only use the members of a family  $\mathcal{F}_n$  of finite types,  $\mathcal{F}_n$  having  $n$  distinct canonical members. We then define the type  $\mathcal{T}_n$  of terms that use the objects of  $\mathcal{F}_n$  as free variables. We shall refer to these as “**nat-indexed terms**”.

We define a family of finite types  $\mathcal{F}_n$  ( $n : \mathbb{N}$ ),  $\mathcal{F}_n$  having exactly  $n$  distinct canonical objects. Following our first maxim (see Section 2), we define  $\mathcal{F}$ , not as an inductive family of types, but as a recursive function thus:

$$\mathcal{F}_0 = \emptyset, \quad \mathcal{F}_{n+1} = (\mathcal{F}_n)_\perp .$$

Here,  $\emptyset$  (empty) is the empty type, and  $X_\perp$  (option  $X$ ) is a type consisting of a copy  $\uparrow x$  of each object  $x$  of  $X$  together with one extra object  $\perp$ . Both of these types are provided by the Coq library.

We now define the family of types  $\mathcal{T}_n$  (**term**  $n$ ) of terms using the objects of type  $\mathcal{F}_n$  as free variables, for  $n : \mathbb{N}$ :

$$\frac{x : \mathcal{F}_n}{x : \mathcal{T}_n} \quad \frac{s : \mathcal{S}}{s : \mathcal{T}_n} \quad \frac{M : \mathcal{T}_n \quad N : \mathcal{T}_n}{MN : \mathcal{T}_n} \quad \frac{A : \mathcal{T}_n \quad B : \mathcal{T}_{n+1}}{\Pi AB : \mathcal{T}_n} \quad \frac{A : \mathcal{T}_n \quad M : \mathcal{T}_{n+1}}{\lambda AM : \mathcal{T}_n}$$

We proceed to define the replacement and substitution operations, as discussed in the previous section. We can then prove the following various forms of the Substitution Lemma:

$$\begin{aligned} M\{\rho\}\{\rho'\} &\equiv M\{\rho' \circ \rho\} & M\{\rho\}[\sigma] &\equiv M[\sigma \circ \rho] \\ M[\sigma]\{\rho\} &\equiv M[x \mapsto \sigma(x)\{\rho\}] & M[\sigma][\sigma'] &\equiv M[x \mapsto \sigma(x)[\sigma']] \end{aligned}$$

**Reduction Relation.** We define the relation of *parallel one-step reduction*,  $\triangleright$ , which has type  $\Pi n : \mathbb{N}. \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathbf{Prop}$ , thus:

$$\frac{x : \mathcal{F}_n}{x \triangleright x} \quad \frac{s : \mathcal{S}}{s \triangleright s} \quad \frac{M \triangleright M' \quad N \triangleright N'}{MN \triangleright M'N'}$$

$$\frac{A \triangleright A' \quad B \triangleright B'}{\Pi AB \triangleright \Pi A'B'} \quad \frac{A \triangleright A' \quad M \triangleright M'}{\lambda AM \triangleright \lambda A'M'} \quad \frac{M \triangleright M' \quad N \triangleright N'}{(\lambda AM)N \triangleright M' \left[ \begin{array}{l} \perp \mapsto N' \\ \uparrow x \mapsto x \end{array} \right]}$$

Note, in particular, the way that substitution of  $N'$  for  $\perp$  is defined in terms of our big-step substitution in the final clause.

The relation of reduction,  $\rightarrow$ , is defined to be the transitive closure of  $\triangleright$ , and the relation of convertibility,  $\simeq$ , is defined to be the symmetric, transitive closure of  $\triangleright$ . We prove that  $\triangleright$  satisfies the diamond condition, and deduce the Church-Rosser Theorem.

**Contexts.** We now define the type  $\mathcal{C}_n$  of contexts with domain  $\mathcal{F}_n$ . As with variables, to make case analysis easier, we do not define this as an inductive family, but rather by recursion on  $n$  as follows:

$$\mathcal{C}_0 = \mathbf{1} \quad \mathcal{C}_{n+1} = \mathcal{C}_n \times \mathcal{T}_n$$

Here,  $\mathbf{1}$  (`unit`) is a type with a unique canonical element  $*$  (`tt`), and  $A \times B$  (`prod A B`) is the Cartesian product of  $A$  and  $B$ , both provided by the Coq library.

We can define the function `typeof`, with type  $\Pi n : \mathbb{N}. \mathcal{F}_n \rightarrow \mathcal{C}_n \rightarrow \mathcal{T}_n$ , which looks up the type of the variable  $x : \mathcal{F}_n$  in the context  $\Gamma : \mathcal{C}_n$ . We shall write  $\Gamma(x)$  for `typeof _ x Γ` in this paper. Thus, the situation that we would describe on paper by “ $x : A \in \Gamma$ ” is expressed in our formalization by  $\Gamma(x) \equiv A$ .

The `typeof` function is defined by recursion on  $n$  as follows (the case  $n = 0$  being vacuous):

$$\begin{aligned} \langle \Gamma, A \rangle(\perp) &\equiv A\{\uparrow\} \\ \langle \Gamma, A \rangle(\uparrow x) &\equiv \Gamma(x)\{\uparrow\} \end{aligned}$$

### 3.2 Typing Relation

We declare the axioms  $\mathcal{A} : \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathbf{Prop}$  and rules  $\mathcal{R} : \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathbf{Prop}$  of the arbitrary PTS with which we are working as parameters, and we are then finally able to defined the typing relation  $\vdash$  of the PTS. This relation has type

$$\Pi n : \mathbb{N}. \mathcal{C}_n \rightarrow \mathcal{T}_n \rightarrow \mathcal{T}_n \rightarrow \mathbf{Prop}$$

When given a context  $\Gamma : \mathcal{C}_n$  and terms  $M, A : \mathcal{T}_n$ , it returns the proposition “The judgement  $\Gamma \vdash M : A$  is derivable”. It is defined as an inductive relation, and its constructors are simply the rules of deduction of a PTS (see Figure 1).

**Subcontext Relation.** The definition of the subcontext relation is the first place where our formalization differs significantly from the informal development of the metatheory.

When we use named variables, the subcontext relation is defined as follows:



$$\begin{array}{l}
\text{axioms :} \quad \frac{}{* \vdash s : t} \quad (\mathcal{A} \ s \ t) \\
\\
\text{start :} \quad \frac{\Gamma \vdash A : s}{\Gamma, A \vdash \perp : A\{\uparrow\}} \\
\\
\text{weakening :} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, C \vdash A\{\uparrow\} : B\{\uparrow\}} \\
\\
\text{product :} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, A \vdash B : s_2}{\Gamma \vdash \Pi AB : s_3} \quad (\mathcal{R} \ s_1 \ s_2 \ s_3) \\
\\
\text{application :} \quad \frac{\Gamma \vdash M : \Pi AB \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B \left[ \begin{array}{l} \perp \mapsto N \\ \uparrow x \mapsto x \end{array} \right]} \\
\\
\text{abstraction :} \quad \frac{\Gamma, A \vdash M : B \quad \Gamma \vdash A : s_1 \quad \Gamma, A \vdash B : s_2}{\Gamma \vdash \lambda AM : \Pi AB} \quad (\mathcal{R} \ s_1 \ s_2 \ s_3) \\
\\
\text{conversion :} \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad (A \simeq B).
\end{array}$$

**Fig. 1.** Rules of Deduction of a Pure Type System

The context  $\Gamma$  is a subcontext of  $\Delta$  iff, for every variable  $x$  and type  $A$ , if  $x : A \in \Gamma$  then  $x : A \in \Delta$ .

We could think of a *function* giving, for each entry  $x : A$  in  $\Gamma$ , the position at which  $x : A$  occurs in  $\Delta$ . If  $\Gamma$  is of length  $m$  and  $\Delta$  of length  $n$ , then we can write the definition in the following form:

$\Gamma$  is a subcontext of  $\Delta$  iff there exists a function  $\rho : \{1, \dots, m\} \rightarrow \{1, \dots, n\}$  such that, if the  $i$ th entry of  $\Gamma$  is  $x : A$ , then the  $\rho(i)$ th entry of  $\Delta$  is  $x : A$ .

In our formalization, we cannot use the same definition, as the variables of  $\mathcal{F}_n$  come in a fixed order:  $\perp, \uparrow \perp, \uparrow \uparrow \perp, \dots$ . But we can still talk of functions mapping positions in one context to positions in another. In fact, we have met these functions before: they are the *renaming* functions  $\rho : \mathcal{F}_m \rightarrow \mathcal{F}_n$  used by the replacement operation.

We therefore define the relation: “ $\Gamma$  is a subcontext of  $\Delta$  *under* the renaming  $\rho$ ”. This means that, if we identify each variable  $x$  in  $\Gamma$  with the variable  $\rho(x)$  in  $\Delta$ , then the entry with subject  $x$  in  $\Gamma$  is the same as the entry with subject  $\rho(x)$  in  $\Delta$ . More precisely:

**Definition 1.** Let  $\Gamma : \mathcal{C}_m$ ,  $\Delta : \mathcal{C}_n$ , and  $\rho : \mathcal{F}_m \rightarrow \mathcal{F}_n$ .  $\Gamma$  is a subcontext of  $\Delta$  under  $\rho$ ,  $\Gamma \subseteq_\rho \Delta$ , iff  $(\forall x : \mathcal{F}_m) \Delta(\rho(x)) \equiv \Gamma(x)\{\rho\}$ .

Note that the relation we have defined allows contraction:  $\rho$  may map two different variables  $x$  and  $y$  to the same variable in  $\mathcal{F}_n$  (in which case  $x$  and  $y$  must have the same type in  $\Gamma$ ). If we wish to exclude contraction, we shall use the relation “ $\Gamma \subseteq_\rho \Delta \wedge \rho$  is injective”.

With this definition, the Weakening result becomes:

**Theorem 1 (Weakening).** *If  $\Gamma \subseteq_\rho \Delta$ ,  $\Gamma \vdash M : A$ , and  $\Delta$  is valid, then  $\Delta \vdash M\{\rho\} : A\{\rho\}$ .*

This is a big-step version of the Weakening property: the small-step version would be

$$\text{If } \Gamma, \Delta \vdash M : A \text{ and } \Gamma \vdash B : s \text{ then } \Gamma, x : B, \Delta \vdash M : A.$$

Apart from the definition of the subcontext relation, the big-step version of the Weakening Lemma is not particularly novel: it appears in many informal and formal developments, including McKinna and Pollack [6]. The form that the next result takes is probably more surprising:

**Substitution.** What form should the Substitution result take, given that we are using substitution mappings  $\sigma : \mathcal{F}_m \rightarrow \mathcal{T}_n$ ? A moment’s thought will show that, with named variables, the result would read as follows:

$$\text{If } x_1 : A_1, \dots, x_m : A_m \vdash M : B \text{ and}$$

$$\Gamma \vdash \sigma(x_1) : A_1[\sigma], \Gamma \vdash \sigma(x_2) : A_2[\sigma], \dots, \Gamma \vdash \sigma(x_m) : A_m[\sigma]$$

$$\text{then } \Gamma \vdash M[\sigma] : B[\sigma].$$

(For the case  $m = 0$ , we would need to add the hypothesis “ $\Gamma$  is valid”.)

It is then clear how the result should read with **nat**-indexed variables. Let us first introduce a definition to abbreviate the hypotheses.

Let us define the relation  $\Gamma \models \sigma :: \Delta$  ( $\sigma$  satisfies the context  $\Delta$  under  $\Gamma$ ) to mean

$$(\forall x : \mathcal{F}_m) \Gamma \vdash \sigma(x) : \Delta(x)[\sigma].$$

Then our big-step version of the Substitution property reads:

**Theorem 2 (Substitution).** *If  $\Delta \vdash M : B$ ,  $\Gamma \models \sigma :: \Delta$ , and  $\Gamma$  is valid, then  $\Gamma \vdash M[\sigma] : B[\sigma]$ .*

This is proven by induction on the derivation of  $\Delta \vdash M : B$ .

**Subject Reduction.** Subject Reduction similarly takes a big-step form. Subject Reduction is traditionally stated in the form:

$$\text{If } \Gamma \vdash M : A \text{ and } M \rightarrow N, \text{ then } \Gamma \vdash N : A.$$

In our formalization, the most convenient form in which to prove Subject Reduction is as follows. We extend the notion of parallel one-step reduction to contexts, by making the following definition:

**Definition 2.** Define the relation of parallel one-step reduction,  $\triangleright$ , on  $\mathcal{C}_n$  as follows:  $\Gamma \triangleright \Delta$  iff  $(\forall x : \mathcal{F}_n) \Gamma(x) \triangleright \Delta(x)$ .

We can now prove:

**Theorem 3 (Subject Reduction).** If  $\Gamma \vdash M : A$ ,  $\Gamma \triangleright \Delta$  and  $M \triangleright N$ , then  $\Delta \vdash N : A$ .

This is proven by induction on the derivation of  $\Gamma \vdash N : A$ . The usual form of Subject Reduction follows quite simply.

**Other Results.** The formalization also contains proofs of Context Conversion, the Generation lemmas, Type Validity (that if  $\Gamma \vdash M : A$  then either  $A$  is a sort or  $\Gamma \vdash A : s$  for some sort  $s$ ), Predicate Reduction and the Uniqueness of Types result for functional PTSs. Apart from Context Conversion, which takes the expected big-step form, these results take the same form, and the proofs follow the same lines, as the paper development.

### 3.3 Strings of Binders

The proof of the Strengthening Theorem in van Bentham Jutting's paper [5] is a technically complex proof, and a very good, tough test of any formalization of the theory of PTSs.

Perhaps surprisingly, the feature that makes the proof most difficult to formalize when using `nat`-indexed terms is the use of terms of the form  $\Lambda \Delta.M$  and  $\Pi \Delta.M$ , where  $\Delta$  is a *string* of abstractions

$$\Delta \equiv \langle x_1 : A_1, \dots, x_n : A_n \rangle$$

We shall need to build the type of strings in such a way that these operations  $\Lambda$  and  $\Pi$  can be defined. A closer look at the proofs in [5] reveals that we shall also need an operation for concatenating a context with a string; that is, given a context  $\Gamma$  and a string  $\Delta$ , producing a context  $\Gamma \hat{\Delta}$ . We must also be able to apply a substitution to a string.

Several different approaches to these definitions were tried before the ones described below were found. As we have discussed, it was found to be important to *avoid arithmetic within types* at all costs. This proved impossible to do without violating our first maxim, to prefer recursive definitions to inductive ones. Even then, it was difficult to find a set of definitions that avoids the need for addition, particularly within the type of the substitution operation. A subtle solution was eventually found, and is described below.

**Strings.** We define the inductive family of types `string`. If  $n \leq m$ , the type `string m n` is the type of all strings

$$\Delta \equiv \langle A_m, A_{m+1}, A_{m+2}, \dots, A_{n-1} \rangle$$

such that  $A_m : \mathcal{T}_m$ ,  $A_{m+1} : \mathcal{T}_{m+1}$ ,  $\dots$ ,  $A_{n-1} : \mathcal{T}_{n-1}$ . If  $n = m$ , the type has only one member (the empty string); and if  $n > m$ , `string m n` is empty.

$$\frac{m : \mathbb{N} \quad n : \mathbb{N}}{\mathbf{string} \ m \ n : \mathbf{Set}} \quad \frac{n : \mathbb{N}}{\langle \rangle : \mathbf{string} \ n \ n} \quad \frac{A : \mathcal{T}_m \quad \Delta : \mathbf{string} \ m + 1 \ n}{A :: \Delta : \mathbf{string} \ m \ n}$$

Using the standard technical trick described above in Section 2.1, we can prove that  $\langle \rangle$  is the only object of type  $\mathbf{string} \ n \ n$ , and every object  $\Delta$  in  $\mathbf{string} \ m \ n$  has the form  $A :: \Delta'$  if  $m < n$ .

We define the operations  $\Pi$  and  $\Lambda$  that operate on strings and terms, and the operation of *concatenation* that appends a string to a context. The types for these operations are as follows:

$$\frac{\Delta : \mathbf{string} \ m \ n \quad A : \mathcal{T}_n}{\Pi \Delta . A : \mathcal{T}_m} \quad \frac{\Delta : \mathbf{string} \ m \ n \quad A : \mathcal{T}_n}{\Lambda \Delta . A : \mathcal{T}_m} \quad \frac{\Gamma : \mathcal{C}_m \quad \Delta : \mathbf{string} \ m \ n}{\Gamma \hat{\ } \Delta : \mathcal{C}_n}$$

These operations can all be defined by recursion on the string  $\Delta$ .

**Substitution in Strings.** The definition of substitution on strings is very tricky. One natural suggestion would be to define an operation with type

$$\Pi m, n, p : \mathbb{N}. \mathbf{string} \ m \ n \rightarrow (\mathcal{F}_m \rightarrow \mathcal{T}_{m+p}) \rightarrow \mathbf{string} \ (m + p) \ (n + p).$$

However, as mentioned above, it proved necessary to find a definition that would not involve addition within any type.

After many such false starts, the following solution was arrived at. We define a relation on pairs of natural numbers, which we call *matching*:

$$\langle m, n \rangle \sim \langle p, q \rangle$$

(read: “ $\langle m, n \rangle$  matches  $\langle p, q \rangle$ ”). This relation is equivalent to

$$n \leq m \wedge m - n = p - q.$$

If  $\langle m, n \rangle \sim \langle p, q \rangle$ , then it is possible to apply a substitution  $\mathcal{F}_m \rightarrow \mathcal{T}_p$  to a string in  $\mathbf{string} \ m \ n$  to obtain a string of type  $\mathbf{string} \ p \ q$ .

(We actually place the type  $\langle m, n \rangle \sim \langle p, q \rangle$  in  $\mathbf{Set}$ , as we shall need to define substitution by recursion on the proof that  $\langle m, n \rangle \sim \langle p, q \rangle$ .)

**Definition 3.** Define the relation  $\langle m, n \rangle \sim \langle p, q \rangle$  inductively as follows:

$$\frac{m : \mathbb{N} \quad p : \mathbb{N}}{\langle m, m \rangle \sim \langle p, p \rangle} \quad \frac{\langle m + 1, n \rangle \sim \langle p + 1, q \rangle}{\langle m, n \rangle \sim \langle p, q \rangle}$$

We can now define the substitution operation on strings:

**Definition 4.** Suppose  $\langle m, n \rangle \sim \langle p, q \rangle$ . We define, for each  $\Delta : \mathbf{string} \ m \ n$  and  $\rho : \mathcal{F}_n \rightarrow \mathcal{T}_q$ , the string  $\Delta[\rho] : \mathbf{string} \ p \ q$ , by recursion on the proof of  $\langle m, n \rangle \sim \langle p, q \rangle$  as follows:

- The base case is  $\langle m, m \rangle \sim \langle p, p \rangle$ . For  $\Delta : \mathbf{string} \ m \ m$  and  $\sigma : \mathcal{F}_m \rightarrow \mathcal{T}_p$ ,

$$\Delta[\sigma] \equiv \langle \rangle : \mathbf{string} \ p \ p$$

- Suppose  $\langle m, n \rangle \sim \langle p, q \rangle$  was deduced from  $\langle m + 1, n \rangle \sim \langle p + 1, q \rangle$ . For  $\Delta : \text{string } m \ n$  and  $\sigma : \mathcal{F}_m \rightarrow \mathcal{T}_p$ ,

$$\Delta[\sigma] \equiv (\text{head } \Delta)[\sigma] :: (\text{tail } \Delta)[\sigma_{\perp}] : \text{string } p \ q$$

Now that these operations are in place, it is a straightforward, albeit lengthy, task to formalise van Bentham Jutting’s proof of Strengthening. We omit the details here; we refer the interested reader to [9]. We note in passing only that, as is by now to be expected, the Strengthening Theorem itself takes a big-step form:

**Theorem 4 (Strengthening).** *If  $\Gamma \subseteq_{\rho} \Delta$ ,  $\rho$  is injective,  $\Delta \vdash M : A$ , and  $\Gamma$  is valid, then  $\Gamma \vdash M : A$ .*

## 4 Systems with Judgemental Equality

The author’s original motivation for this work was to check a technically complex proof of a result of his in the theory of PTSs [10], in order to obtain a guarantee of its correctness. It is worth stating briefly how the formalization of the system with judgemental equality proceeds, as the metatheory of these systems have not often been formalized.

We use the same types  $\mathcal{T}_n$  of terms and  $\mathcal{C}_n$  of contexts that we have already constructed. The system with judgemental equality has two judgement forms:

$$\Gamma \vdash M : A \text{ and } \Gamma \vdash M = N : A .$$

We build these using a mutual inductive definition:

```

Inductive PTS' :
  forall n, context n -> term n -> term n -> Prop :=
  ...
with PTSeq :
  forall n, context n -> term n -> term n -> term n -> Prop :=
  ...
    
```

We define the notion of a *valid context* of type  $\mathcal{C}_n$ , by induction on  $n$ :

$$\begin{aligned} (\langle \rangle \text{ is valid}) &\equiv \top \\ (\langle \Gamma, A \rangle \text{ is valid}) &\equiv \exists s : \mathcal{S}. \Gamma \vdash A : s \end{aligned}$$

We also need the notion of *equality of contexts*: we define the proposition  $\Gamma = \Delta$  for  $\Gamma$  and  $\Delta$  both of the same type  $\mathcal{C}_n$ .

$$\begin{aligned} (\langle \rangle = \langle \rangle) &\equiv \top \\ (\langle \Gamma, A \rangle = \langle \Delta, B \rangle) &\equiv \Gamma = \Delta \wedge \exists s : \mathcal{S}. \Gamma \vdash A = B : s \end{aligned}$$

The definition of satisfaction is similar to the one we used in PTSs: for  $\Gamma : \mathcal{C}_n$ ,  $\Delta : \mathcal{C}_m$ , and  $\sigma : \mathcal{F}_m \rightarrow \mathcal{T}_n$ , we define

$$(\Gamma \models \sigma :: \Delta) \equiv \forall x : \mathcal{F}_m. \Gamma \vdash \sigma(x) : \Delta(x)[\sigma]$$

We also need the notion of two substitutions, each of which satisfies  $\Delta$  under  $\Gamma$ , being equal. For  $\Gamma : \mathcal{C}_n$ ,  $\Delta : \mathcal{C}_m$ , and  $\sigma, \sigma' : \mathcal{F}_m \rightarrow \mathcal{T}_n$ , we define

$$(\Gamma \models \sigma = \sigma' :: \Delta) \equiv \forall x : \mathcal{F}_m. \Gamma \vdash \sigma(x) = \sigma'(x) : \Delta(x)[\sigma] .$$

We then prove:

**Theorem 5.**

1. **Context Validity** If  $\Gamma \vdash J$ , then  $\Gamma$  is valid.
2. **Context Conversion** If  $\Gamma \vdash J$ ,  $\Gamma = \Delta$ , and  $\Delta$  is valid, then  $\Delta \vdash J$ .
3. **Substitution** If  $\Gamma \models \sigma :: \Delta$ ,  $\Delta \vdash J$ , and  $\Gamma$  is valid, then  $\Gamma \vdash J[\sigma]$ .
4. **Weakening** If  $\Gamma \vdash J$ ,  $\Gamma \subseteq_\rho \Delta$ , and  $\Delta$  is valid, then  $\Delta \vdash J\{\rho\}$ .
5. **Functionality** If  $\Gamma \models \sigma = \sigma' :: \Delta$ ,  $\Delta \vdash M : A$ , and  $\Gamma$  is valid, then  $\Gamma \vdash M[\sigma] = M[\sigma'] : A[\sigma]$ .

In each case,  $J$  stands for either of the judgement bodies  $M : A$  or  $M = N : A$ . In the formalization, each of these statements (except Functionality) is therefore two theorems. They are proven simultaneously — that is, we prove their conjunction by a simultaneous induction on the derivation of the premise.

We note that Weakening is not needed in the proof of Substitution. Weakening can either be proven by an induction on its premise, or as a special case of Substitution.

The Start, Generation and Type Validity lemmas can also be proven; these all take the expected form.

The form that the statement of Subject Reduction takes is:

$$\text{If } \Gamma \vdash M : A \text{ and } M \rightarrow N, \text{ then } \Gamma \vdash M = N : A.$$

This is a very difficult property to prove for systems with judgemental equality, either on paper or formally. Its proof can be seen as the main result in the paper [10]. The formalization of this proof is not yet complete.

## 5 Related Work

McKinna and Pollack [6] produced a large formalization of many results in the theory of PTSs, based on a representation of syntax that uses named variables. They use two separate types:  $V$  for the bound variables, and  $P$  for the free variables, or *parameters*. It was a concern of theirs not to gloss over such matters as  $\alpha$ -conversion, as is usually done in informal developments — to ‘take symbols seriously’, in their words. If one shares this concern, or one wishes to stay close to a particular implementation, then this is an excellent formalization to look at.

However, if one’s concern is solely to obtain a guarantee of the correctness of a metatheoretic result, this formalization has two principal disadvantages. Firstly, we are often dealing with operations renaming variables, or replacing a variable with a parameter or vice versa; we would prefer not to have to deal with these technicalities. Secondly, we have objects that do not correspond to any term

in the syntax — namely, those in which objects of type  $V$  occur free. We need frequently to check that every object of type  $V$  is bound in the terms with which we are dealing — that they are *closed*, in McKinna and Pollack’s terminology.

Barras produced a formalization in Coq of the metatheory of the Calculus of Constructions, , entitled “Coq in Coq” [7, 11], in the hope of certifying Coq itself — or, at least, certifying a proof checker that is as close to Coq as Gödel’s Theorem allows. As terms are represented internally in Coq with de Bruijn notation, so does Barras’s formalization, using Coq’s natural numbers for the free and bound variables.

Again, this formalization quickly becomes technically complex. It involves operations such as  $\uparrow_k^n$ , which raises every de Bruijn index after the  $k$ th by  $n$  places. We are frequently using lemmas about how these operations interact with each other, or with substitutions. Proofs often proceed by several case analyses involving comparisons of natural numbers, or arithmetical manipulation of the sub- and super-scripts of these operators.

Also worthy of mention is higher-order abstract syntax [12], which embeds the object theory one is studying within the type theory in which one is working, then takes advantage of the type theory’s binding and substitution operations. This did not seem suitable for our purposes; the author thought it important to maintain the separation between object theory and metatheory for this work.

Several other approaches to the representation of syntax have been developed more recently, including Gabbay and Pitts’ work [13], which develops variable binding as an operation in FM-set theory, and CINNI [14], a formal system which contains both named variable syntax and de Bruijn notation as subsystems. It would be very interesting to see a similar formal development of the metatheory of some system based on either of these.

## 6 Conclusion

Which formalization of a given piece of mathematics one prefers depends, of course, on what one intends to use the formalization for, and simply on personal taste. If one’s sole concern is to obtain a guarantee of the correctness of the proof of a metatheoretic result, then one has free choice of which representation of terms one uses. We have seen that, if one chooses this indexed family representation, then careful thought is needed over the form of definitions and theorems; but, once the correct forms have been found, the proofs are short, simple, direct and elegant.

The technicalities in our formalizations mostly involve the replacement and substitution functions, how they interact with each other and with reduction, conversion and so forth. These are quite natural objects to find in the metatheory of a type theory, so we feel this work has a more ‘type-theoretic’ flavour than is the case with many other formalizations.

One’s time is spent thinking about the most convenient form for definitions and theorems, rather than proving many technical lemmas. This suits the author’s preferences nicely; others may prefer an approach that, for example, lends

itself better to automation of the technical parts. Nevertheless, it shall hopefully be useful to see what shape formalized metatheory takes when the indexed family representation of terms is used.

## References

1. Bellegarde, F., Hook, J.: Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.* **23** (1994) 287–311
2. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: *CSL '99*. Volume 1683 of LNCS., Springer-Verlag (1999) 453–468
3. Bird, R.S., Paterson, R.: de Bruijn notation as a nested datatype. *J. Functional Programming* **9** (1999) 77–91
4. Barendregt, H.: Lambda calculi with types. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: *Handbook of Logic in Computer Science*. Volume II. Oxford University Press (1992)
5. van Benthem Jutting, L.S.: Typing in pure type systems. *Information and Computation* **105** (1993) 30–41
6. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. *J. Autom. Reasoning* **23** (1999) 373–409
7. Barras, B.: Auto-validation d'un système de preuves avec familles inductives. PhD thesis, Université Paris 7 (1999)
8. Letouzey, P.: Vectors. Message to Coq-Club mailing list (2004) <http://pauillac.inria.fr/pipermail/coq-club/2004/001265.html>.
9. Adams, R.: A formalization of the theory of PTSs in Coq. Available online at <http://www.cs.rhul.ac.uk/~robin/coqPTS/docu.dvi> (2005)
10. Adams, R.: Pure type systems with judgemental equality. (Accepted for publication in the *Journal of Functional Programming*)
11. Barras, B.: A formalization of the calculus of constructions. (Web page) <http://coq.inria.fr/contribs/coq-in-coq.html>
12. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, New York, NY, USA, ACM Press (1988) 199–208
13. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. *Formal Aspects of Computing* **13** (2002) 341–363
14. Stehr, M.O.: CINNI: A generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi. In: *Third International Workshop on Rewriting Logic and its Applications (WRLA'2000)*, Kanazawa, Japan, September 18 – 20, 2000. Volume 36 of *Electronic Notes in Theoretical Computer Science.*, Elsevier (2000)



# A Content Based Mathematical Search Engine: Whelp

Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen,  
Enrico Tassi, and Stefano Zacchiroli

Department of Computer Science, University of Bologna,  
Mura Anteo Zamboni, 7 — 40127 Bologna, Italy  
{`aspersi`, `fguidi`, `sacerdot`, `tassi`, `zacchiro`}@cs.unibo.it

**Abstract.** The prototype of a content based search engine for mathematical knowledge supporting a small set of queries requiring matching and/or typing operations is described. The prototype — called Whelp — exploits a metadata approach for indexing the information that looks far more flexible than traditional indexing techniques for structured expressions like substitution, discrimination, or context trees. The prototype has been instantiated to the standard library of the Coq proof assistant extended with many user contributions.

## 1 Introduction

The paper describes the prototype of a content based search engine for mathematical knowledge — called Whelp — developed inside the European Project IST-2001-33562 MoWGLI [4]. Whelp has been mostly tested to search notions inside the library of formal mathematical knowledge of the Coq proof assistant [8]. Due to its dimension (about 40,000 theorems), this library was adopted by MoWGLI as a main example of repository of structured mathematical information. However, Whelp — better, its filtering phase — only works on a small set of metadata automatically extracted from the structured sources, and is thus largely independent from the actual syntax (and semantics) of the information. Metadata also offer a higher flexibility with respect to more canonical indexing techniques such as discrimination trees [16], substitution trees [13] or context trees [12] since all these approaches are optimized for the single operation of (forward) matching, and are difficult to adapt or tune with additional constraints (such as global constraints on the signature of the term, just to make a simple but significant example).

Whelp is the final output of a three-year research work inside MoWGLI which consisted in exporting the Coq library into XML, defining a suitable set of metadata for indexing the information, implementing the actual indexing tools, and finally designing and developing the search engine. Whelp itself is the result of a complete architectural re-visitation of a first prototype described in [14], integrated with the efficient retrieval mechanisms described in [3] (further improved as described in Sect. 5.2), and integrated with syntactic facilities borrowed from

the disambiguating parser of [19]. Since the prototype version described in [14], also the Web interface has been completely rewritten and simplified, exploiting most of the publishing techniques developed for the hypertextual rendering of the Coq library (see <http://helm.cs.unibo.it/>) and described in Sect. 6.

The Whelp search engine is located at <http://helm.cs.unibo.it/whelp>.

## 2 Syntax

Whelp interacts with the user as a classical World Wide Web search engine, it expects single line queries and returns a list of results matching it. Whelp currently supports four different kinds of queries, addressing different user-requirements emerged in MoWGLI: MATCH, HINT, ELIM, and LOCATE (described in Sect. 5). The list is not meant to be exhaustive and is likely to grow in the future.

The most typical of these queries (MATCH and HINT) require the user to input a term of the Calculus of (co-)Inductive Constructions — CIC — (the underlying calculus of Coq), supporting different kinds of pattern based queries. Nevertheless, the concrete syntax we chose for writing the input term is not bound to any specific logical system: it has been designed to be as similar as possible to ordinary mathematics formulae, in their  $\text{\TeX}$  encoding (see Table 1).

**Table 1.** Whelp’s term syntax

<i>term</i>	::= <i>identifier</i>	
	<i>number</i>	
	Prop   Type   Set	sort
	?	placeholder
	<i>term term</i>	application
	<i>binder vars . term</i>	abstraction
	<i>term \to term</i>	arrow type
	( <i>term</i> )	grouping
	<i>term binop term</i>	binary operator
	<i>unop term</i>	unary operator
<i>binder</i>	::= \forall   \exists   \lambda	
<i>vars</i>	::= <i>names</i>	variables
	<i>names : term</i>	typed variables
<i>names</i>	::= <i>identifier</i>   <i>identifier names</i>	
<i>binop</i>	::= +   -   *   /   ^	arithmetic operators
	<   >   \leq   \geq   =   \neq	comparison operators
	\lor   \land	logical operators
<i>unop</i>	::= -	unary minus
	\not	logical negation

As a consequence of the generality of syntax, user provided terms do not usually have a unique direct mapping to CIC term, but must be suitably interpreted in order to solve *ambiguities*. Consider for example the following term

input: `\forall x. 1*x = x.` in order to find the corresponding CIC term we need to know the possible meanings of the number 1 and the symbol `*`.<sup>1</sup>

The typical processing of a user query (depicted in Fig. 1) is therefore a pipeline made of four distinct phases: *parsing* (canonical transformation from concrete textual syntax to Abstract Syntax Trees, ASTs for short), *disambiguation* (described in next section), *metadata extraction* (described in Sect. 4), and the actual *query* (described in Sect. 5).

It is worth observing that while the preliminary phases (disambiguation and metadata extraction) of Whelp are not logic independent, the query engine is.

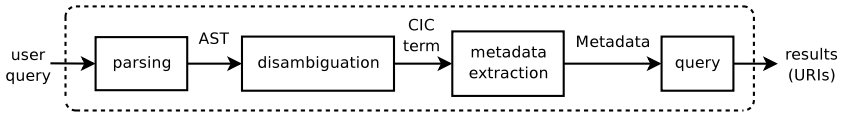


Fig. 1. Whelp’s processing

### 3 Disambiguation

The disambiguation phase builds CIC terms from ASTs of user inputs (also called *ambiguous terms*). Ambiguous terms may carry three different *sources of ambiguity*: unbound identifiers, literal numbers, and literal symbols. *Unbound identifiers* are sources of ambiguity since the same name could have been used to represent different objects. For example, three different theorems of the Coq library share the name *plus\_assoc* (locating them is an exercise for the interested reader. Hint: use Whelp’s LOCATE query).

*Numbers* are ambiguous since several different encodings of them could be provided in logical systems. In the Coq standard library for example we found naturals (in their unary encoding), positives (binary encoding), integers (signed positives), and reals. Finally, *symbols* (instances of the *binop* and *unop* syntactic categories of Table 1) are ambiguous as well: infix `+` for example is overloaded to represent addition over the four different kinds of numbers available in the Coq standard library. Note that given a term with more than one sources of ambiguity, not all possible disambiguation choices are valid: for example, given the input `1+1` we must choose an interpretation of `+` which is typable in CIC according to the chosen interpretation for `1`; choosing as `+` the addition over natural numbers and as `1` the real number 1 will lead to a type error.

A *disambiguation algorithm* takes as input an ambiguous term and return a fully determined CIC term. The *naive disambiguation algorithm* takes as input an ambiguous term  $t$  and proceeds as follows:

1. Create disambiguation domains  $\{D_i | i \in \text{Dom}(t)\}$ , where  $\text{Dom}(t)$  is the set of ambiguity sources of  $t$ . Each  $D_i$  is a set of CIC terms.

<sup>1</sup> Note that  $x$  is not undetermined, since it is a bound variable.

2. Let  $\Phi = \{\phi_i | i \in Dom(t), \phi_i \in D_i\}$  be an interpretation for  $t$ . Given  $t$  and an interpretation  $\Phi$ , a CIC term is fully determined. Iterate over all possible interpretations of  $t$  and type-check them, keep only typable interpretations (i.e. interpretations that determine typable terms).
3. Let  $n$  be the number of interpretations who survived step 2. If  $n = 0$  signal a type error. If  $n = 1$  we have found exactly one CIC term corresponding to  $t$ , returns it as output of the disambiguation phase. If  $n > 1$  let the user choose one of the  $n$  interpretations and returns the corresponding term.

The above algorithm is highly inefficient since the number of possible interpretations  $\Phi$  grows exponentially with the number of ambiguity sources. The actual algorithm used in Whelp is far more efficient being, in the average case, linear in the number of ambiguity sources.

The efficient algorithm can be applied if the logic can be extended with metavariables and a refiner can be implemented. This is the case for CIC and several other logics. *Metavariables* [17] are typed, non linear placeholders that can occur in terms;  $?_i$  usually denotes the  $i$ -th metavariable, while  $?$  denotes a freshly created metavariable. A *refiner* [15] is a function whose input is a term with placeholders and whose output is either a new term obtained instantiating some placeholder or  $\epsilon$ , meaning that no well typed instantiation could be found for the placeholders occurring in the term (type error).

The efficient algorithm starts with an interpretation  $\Phi_0 = \{\phi_i | \phi_i = ?, i \in Dom(t)\}$ , which associates a fresh metavariable to each source of ambiguity. Then it iterates refining the current CIC term (i.e. the term obtained interpreting  $t$

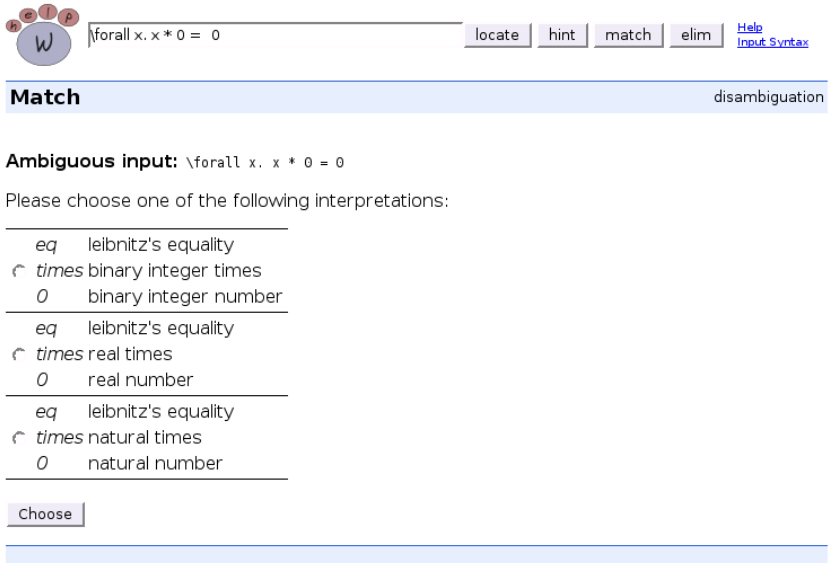


Fig. 2. Disambiguation: interpretation choice

with  $\Phi_i$ ). If the refinement succeeds the next interpretation  $\Phi_{i+1}$  will be created *making a choice*, that is replacing a placeholder with one of the possible choice from the corresponding disambiguation domain. The placeholder to be replaced is chosen following a preorder visit of the ambiguous term. If the refinement fails the current set of choices cannot lead to a well-typed term and backtracking is attempted. Once an unambiguous correct interpretation is found (i.e.  $\Phi_i$  does no longer contain any placeholder), backtracking is attempted anyway to find the other correct interpretations.

The intuition which explain why this algorithm is more efficient is that as soon as a term containing placeholders is not typable, no further instantiation of its placeholders could lead to a typable term. For example, during the disambiguation of user input `\forall x. x*0 = 0`, an interpretation  $\Phi_i$  is encountered which associates ? to the instance of 0 on the right, the real number 0 to the instance of 0 on the left, and the multiplication over natural numbers (`mult` for short) to `*`. The refiner will fail, since `mult` require a natural argument, and no further instantiation of the placeholder will be tried.

If, at the end of the disambiguation, more than one possible interpretations are possible, the user will be asked to choose the intended one (see Fig. 2).

Details of the disambiguation algorithm of Whelp can be found in [19], where an equivalent algorithm that avoids backtracking is also presented.

## 4 Metadata

We use a logic-independent metadata model for indexing mathematical notions. The model is essentially based on a single ternary relation  $Ref_p(s, t)$  stating that an object  $s$  refers an object  $t$  at a given position  $p$ . We use a minimal set of positions discriminating the hypotheses (H), from the conclusion (C) and the proof (P) of a theorem (respectively, the type of the input parameters, the type of the result, and the body of a definition). Moreover, in the hypothesis and in the conclusion we also distinguish the root position (MH and MC, respectively) from deeper positions (that, in a first order setting, essentially amounts to distinguish relational symbols from functional ones). Extending the set of positions we could improve the granularity and the precision of our indexing technique but so far, apart from a simple extension discussed below, we never felt this need.

*Example 1.* Consider the statement:

$$\forall m, n : nat. m \leq n \rightarrow m < (S n)$$

its metadata are described by the following table:

Symbol	Position
<code>nat</code>	MH
<code>≤</code>	MH
<code>&lt;</code>	MC
<code>S</code>	C

All occurrences of bound variables in position MH or MC are collapsed under a unique reserved name *Rel*, forgetting the actual variable name. The occurrences in other positions are not considered. See Sect. 5.3 for an example of use.

The accuracy of metadata for discriminating the statements of the library is remarkable. We computed (see [1]) that the average number of mathematical notions in the Coq library sharing the same metadata set is close to the actual number of *duplicates* (i.e. metadata almost precisely identify statements).

If more accuracy is needed, further filtering phases can be appended to the Whelp pipeline of Fig. 1 to prune out false matches. For instance, since the number of results of the query phase is usually small, very accurate yet slow filters can be exploited.

According to the type as proposition analogy, the metadata above may be also used to index the type of functions. For instance, functions from *nat* to *R* (reals) would be identified by the following metadata:

Symbol	Position
<i>nat</i>	MH
<i>R</i>	MC

in this case, however, the metadata model is a bit too rough, since for instance functions of type  $nat \rightarrow nat$ ,  $nat \rightarrow nat \rightarrow nat$ ,  $(nat \rightarrow nat) \rightarrow nat \rightarrow nat$  and so on would all share the following metadata set:

Symbol	Position
<i>nat</i>	MH
<i>nat</i>	MC

To improve this situation, we add an integer to MC (MH), expressing the number of parameters of the term (respectively, of the given hypothesis). We call *depth* this value since in the case of CIC is equal to the nesting depth of dependent products<sup>2</sup> along the spine relative to the given position. For instance, the three types above would now have the following metadata sets:

$nat \rightarrow nat$	$nat \rightarrow nat \rightarrow nat$	$(nat \rightarrow nat) \rightarrow nat \rightarrow nat$																						
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Symbol</th> <th>Position</th> </tr> </thead> <tbody> <tr> <td><i>nat</i></td> <td>MH(0)</td> </tr> <tr> <td><i>nat</i></td> <td>MC(1)</td> </tr> </tbody> </table>	Symbol	Position	<i>nat</i>	MH(0)	<i>nat</i>	MC(1)	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Symbol</th> <th>Position</th> </tr> </thead> <tbody> <tr> <td><i>nat</i></td> <td>MH(0)</td> </tr> <tr> <td><i>nat</i></td> <td>MC(2)</td> </tr> </tbody> </table>	Symbol	Position	<i>nat</i>	MH(0)	<i>nat</i>	MC(2)	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>Symbol</th> <th>Position</th> </tr> </thead> <tbody> <tr> <td><i>nat</i></td> <td>MH(1)</td> </tr> <tr> <td><i>nat</i></td> <td>H</td> </tr> <tr> <td><i>nat</i></td> <td>MH(0)</td> </tr> <tr> <td><i>nat</i></td> <td>MC(2)</td> </tr> </tbody> </table>	Symbol	Position	<i>nat</i>	MH(1)	<i>nat</i>	H	<i>nat</i>	MH(0)	<i>nat</i>	MC(2)
Symbol	Position																							
<i>nat</i>	MH(0)																							
<i>nat</i>	MC(1)																							
Symbol	Position																							
<i>nat</i>	MH(0)																							
<i>nat</i>	MC(2)																							
Symbol	Position																							
<i>nat</i>	MH(1)																							
<i>nat</i>	H																							
<i>nat</i>	MH(0)																							
<i>nat</i>	MC(2)																							

The depth is a technical improvement that is particularly important for retrieving functions from their types (we shall also see a use in the ELIM query,

<sup>2</sup> Recall that in type theory, the function space is just a degenerate case of dependent product.

Sect. 5.3), but is otherwise of minor relevance. In the following examples, we shall always list it among the metadata for the sake of completeness, but it may be usually neglected by the reader.

## 5 Whelp Queries

### 5.1 Match

Not all mathematical results have a canonical name or a set of keywords which could easily identify them. For this reason, it is extremely useful to be able to search the library by means of the explicit statement. More generally, exploiting the well-known types-as-formulae analogy of Curry-Howard, Whelp’s MATCH operation takes as input a type and returns a list of objects (definition or proofs) inhabiting it.

*Example 2.* Find a proof of the distributivity of times over plus on natural numbers. In order to retrieve those statements, Whelp need to be fed with the distributivity law as input:  $\forall \text{forall } x,y,z:\text{nat. } x * (y+z) = x*y + x*z$ . The MATCH query will return 4 results:

1. `cic:/Coq/Arith/Mult/mult_plus_distr_l.con`
2. `cic:/Coq/Arith/Mult/mult_plus_distr_r.con`
3. `cic:/Rocq/SUBST/comparith/mult_plus_distr_r.con`
4. `cic:/Sophia-Antipolis/HARDWARE/GENE/Arith_compl/mult_plus_distr2.con`

Each result locates a theorem in the Coq library that is organized in a hierarchical fashion. For example (4) identifies the `mult_plus_distr2` theorem in the `Arith_compl.v` file that is part of a contribution on hardware circuits developed at the university of Sophia-Antipolis.

(1), (3), and (4) have types which are  $\alpha$ -convertible with the user query; (2) is an interesting “false match” returned by Whelp having type  $\forall n,m,p \in \mathbb{N}. (n+m) * p = n * p + m * p$ , i.e. it is the symmetric version of the distributivity proposition we were looking for.

The match operation simply amounts to revert the indexing operation, looking for terms matching the metadata set computed from the input. For instance, the term  $\forall \text{forall } x,y,z:\text{nat. } x * (y+z) = x*y + x*z$  has the following metadata:

Symbol	Position
<i>nat</i>	MH(0)
=	MC(3)
<i>nat</i>	C
*	C
+	C

Note that *nat* occurs in conclusion as an hidden parameter of equality; the indexed term is the term after disambiguation, not the user input.

Searching for the distributivity law then amounts to look for a term  $s$  such that :

$$Ref_{MH(0)}(s, nat) \wedge Ref_{MC(3)}(s, =) \wedge Ref_C(s, nat) \wedge Ref_C(s, *) \wedge Ref_C(s, +)$$

In a relational database, this is a simple and efficient join operation.

*Example 3.* Suppose we are interested in looking for a definition of summation for series of natural numbers. The type of such an object is something of the kind  $(nat \rightarrow nat) \rightarrow nat \rightarrow nat \rightarrow nat$ , taking the series, two natural numbers expressing summation lower and upper bound, and giving back the resulting sum. Feeding Whelp's MATCH query with such a type does give back four results:

1. `cic:/Coq/Reals/Rfunctions/sum_nat.f.con`
2. `cic:/Sophia-Antipolis/Bertrand/Product/prod_nm.con`
3. `cic:/Sophia-Antipolis/Bertrand/Summation/sum_nm.con`
4. `cic:/Sophia-Antipolis/Rsa/Binomials/sum_nm.con`

Although we have a definition for summation in the standard library, namely `sum_nat_f`, its theory is very underdeveloped. Luckily we have a much more complete development for `sum_nm` in a contribution from Sophia, where:

$$sum\_nm\ n\ m\ f = \sum_{x=n}^{n+m} f(x)$$

Having discovered the name for summation, we may then inquire about the proofs of some of its properties; for instance, considering the semantics of `sum_nm`, we may wonder if the following statement is already in the library:

$$\forall m, n, c : nat. (sum\_nm\ n\ m\ \lambda x : nat. c) = (S\ m) * c$$

Matching the previous theorem actually succeed, returning the following:


`cic:/Sophia-Antipolis/Bertrand/Summation/sum_nm.c.con`.

**Matching Incomplete Patterns.** Whelp also support matching with partial patterns, i.e. patterns with placeholders denoted by `?`. The approach is essentially identical to the previous one: we compute all the constants  $A$  appearing in the pattern, and look for all terms referring at least the set of constants in  $A$ , at suitable positions.

Suppose for instance that you are interested in looking for all known facts about the computation of `sin` on given reals. You may just ask Whelp to MATCH `sin ? = ?`, that would result in the following list (plus a couple of spurious results due to the fact that Coq variables are not indexed, at present):

1. `cic:/Coq/Reals/Rtrigo/sin_2PI.con`
2. `cic:/Coq/Reals/Rtrigo/sin_PI.con`
3. `cic:/Coq/Reals/Rtrigo/sin_PI2.con`
4. `cic:/Coq/Reals/Rtrigo_calc/sin3PI4.con`





sin ? = ?    locate    hint    match    elim    [Help](#)  
[Input Syntax](#)

---

**Match** 17 results found

1. [cic:/Coq/Reals/Rtrigo/sin\\_2PI.con](#)  
**Object [sin\\_2PI](#):**  
 $(\sin 2*\pi)=0$
2. [cic:/Coq/Reals/Rtrigo/sin\\_PI.con](#)  
**Object [sin\\_PI](#):**  
 $(\sin \pi)=0$
3. [cic:/Coq/Reals/Rtrigo/sin\\_PI2.con](#)  
**Object [sin\\_PI2](#):**  
 $(\sin \pi/2)=1$
4. [cic:/Coq/Reals/Rtrigo\\_calc/sin3PI4.con](#)  
**Object [sin3PI4](#):**  
 $(\sin 3*\pi/(2*2))=1/\sqrt{2}$
5. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_2PI3.con](#)  
**Object [sin\\_2PI3](#):**  
 $(\sin 2*\pi/3)=\sqrt{3}/2$

**Fig. 3.** MATCH results

5. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_2PI3.con](#)
6. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_3PI2.con](#)
7. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_5PI4.con](#)
8. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_PI3.con](#)
9. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_PI3\\_cos\\_PI6.con](#)
10. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_PI4.con](#)
11. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_PI6.con](#)
12. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_PI6\\_cos\\_PI3.con](#)
13. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_cos5PI4.con](#)
14. [cic:/Coq/Reals/Rtrigo\\_calc/sin\\_cos\\_PI4.con](#)
15. [cic:/Coq/Reals/Rtrigo\\_def/sin\\_0.con](#)

Previous statements semantics is reasonably clear by their names; Whelp, however, also performs in-line expansion of the statements, and provides hyperlinks to the corresponding proofs (see Fig. 3).

## 5.2 Hint

In a process of backward construction of a proof, typical of proof assistants, one is often interested in knowing what theorems can be applied to derive the current goal. The HINT operation of Whelp is exactly meant to this purpose.

Formally, given a goal  $g$  and a theorem  $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$ , the problem consists in checking if there exists a substitution  $\theta$  such that:

$$t\theta = g \tag{1}$$

A necessary condition for (1), which provides a very efficient filtering of the solution space, is that the set of constants in  $t$  must be a subset of those in  $g$ . In terms of our metadata model, the problem consists to find all  $s$  such as

$$\{x|Ref(s, x)\} \subseteq A \quad (2)$$

where  $A$  is the set of constants in  $g$ . This is not a simple operation: the naive approach would require to iterate, for every possible source  $s$ , the computation of its forward references, i.e. of  $\{x|Ref(s, x)\}$ , followed by a set comparison with  $A$ .

The solution of [3] is based on the following remarks. Let us call  $Card(s)$  the cardinality of  $\{x|Ref(s, x)\}$ , which can be pre-computed for every  $s$ . Then, (2) holds if and only if there is a subset  $A'$  of  $A$  such that  $A' = \{x|Ref(s, x)\}$ , or equivalently:

$$A' \subseteq \{x|Ref(s, x)\} \wedge |A'| = Card(s)$$

and finally:

$$\bigwedge_{a \in A'} Ref(s, a) \wedge |A'| = Card(s)$$

The last one is a simple join that can be efficiently computed by any relational database. So the actual cost is essentially bounded by the computation of all subsets of  $A$ , and  $A$ , being the signature of a formula, is never very large (and often quite small).

The problem of matching against a large library of heterogeneous notions is however different and far more complex than in a traditional theorem proving setting, where one typically works with respect to a given theory with a fixed, and usually small signature. If e.g. we look for theorems whose conclusion matches some kind of equation like  $e_1 = e_2$  we shall eventually find in the library *a lot* of injectivity results relative to operators we are surely not interested in: in a library of 40,000 theorems like the one of Coq we would get back about 3,000 of such *silly matches*. Stated in other words, canonical indexing techniques specifically tailored on the matching problem such as discrimination trees [16], substitution trees [13] or context trees [12] are eventually doomed to fail in a mathematical knowledge management context where one cannot assume a preliminary knowledge on term signatures.

On the other side, the metadata approach is much more flexible, allowing a simple integration of matching operation with additional and different constraints. For instance in the version of *hint* described in [14] the problem of reducing the number of *silly matches* was solved by requiring at least a minimal intersection between the signatures of the two matching terms. However, this approach did sometimes rule out some interesting answers. In the current version the problem has been solved imposing further constraints of the full signature of the term (in particular on the hypothesis), essentially filtering out all solutions that would extend the signature of the goal. The actual implementation of this approach requires a more or less trivial extension to hypothesis of the methodology described in [3].

### 5.3 Elim

Most statements in the Coq knowledge base concern properties of functions and relations over algebraic types. Proofs of such statements are carried out by structural induction over these types. In particular, to prove a goal by induction over the type  $t$ , one needs to apply a lemma stating an induction principle over  $t$  (an *eliminator* of  $t$  [15]) to that goal. Since many different eliminators can be provided for the same type  $t$  (either automatically generated from  $t$ , or set up by the user), it is convenient to have a way of retrieving all the eliminators of a given type. The ELIM query of Whelp does this job.<sup>3</sup> To understand how it works, let's take the case of the algebraic type of the natural numbers: one feeds Whelp with the identifier *nat*, which denotes this type in the knowledge base, and expects to find at least the well-known induction principle *nat\_ind*:

$$\forall P : nat \rightarrow Prop.(P\ 0) \rightarrow (\forall n : nat.(P\ n) \rightarrow (P\ (S\ n))) \rightarrow \forall n : nat.(P\ n)$$

A fairly good approximation of this statement is based on the following observations: the first premise ( $nat \rightarrow Prop$ ) has an antecedent, a reference to *Prop* in its root and a reference to *nat*; the forth premise has no antecedents and a reference to *nat* in its root; the conclusion contains a bound variable in its root (i.e.  $P$ ). Notice that we choose not to approximate the major premises of *nat\_ind* (the second and the third) because they depend on the structure of *nat* and discriminate the different induction principles over this type.

Thus, a set of constraints approximating *nat\_ind* is the following (recall that *Rel* stands for an arbitrary bound variable):

Symbol	Position
<i>Prop</i>	MH(1)
<i>nat</i>	H
<i>nat</i>	MH(0)
<i>Rel</i>	MC

The ELIM query of Whelp simply generalizes this scheme substituting *nat* for a given type  $t$  and retrieving any statement  $c$  such that:

$$Ref_{MH(1)}(c, Prop) \wedge Ref_H(c, t) \wedge Ref_{MH(0)}(c, t) \wedge Ref_{MC}(c, Rel)$$

In the case of *nat*, ELIM returns 47 statements ordered by the frequency of their use in the library (as expected *nat\_ind* is the first one).

### 5.4 Locate

Whelp's LOCATE query implements a simple "lookup by name" for library notions. Once fed with an identifier  $i$ , LOCATE returns the list of all objects whose

<sup>3</sup> Of course it is possible to let the author state what are the elimination principles, storing this information as metadata accessible to Whelp. The technique we present consists in automatically guessing the intended usage of a theorem from its shape. Moreover, this is the only possible technique for legacy libraries that lack classification information.

name is  $i$ . Intuitively, the *name* of an object contained in library is the name chosen for it by its author.<sup>4</sup>

This list is obtained querying the specific relational metadata  $Name(c, i)$  that binds each unit of knowledge  $c$  to an identifier  $i$  (its *name*). Unix-shell-like wild-cards can be used to specify an incomplete identifier: all objects whose name matches the incomplete identifier are returned.

Even if not based on the metadata model described in Sect. 4, LOCATE turns out to be really useful to browse the library since quite often one remembers the name of an object, but not the corresponding contribution.

*Example 4.* By entering the name *gcd*, LOCATE returns four different versions of the “greatest common divisor”:

1. `cic:/Orsay/Maths/gcd/gcd.ind#xpointer(1/1)`
2. `cic:/Eindhoven/POCKLINGTON/gcd/gcd.con`
3. `cic:/Sophia-Antipolis/Bertrand/Gcd/gcd.con`
4. `cic:/Sophia-Antipolis/Rsa/Divides/gcd.con`

## 6 Web Interface

The result of Whelp is, for all queries, a list of URIs (unique identifiers) for notions in the library. This list is not particularly informative for the user, who would like to have hyperlinks or, even better, in-line expansion of the notions.

In the MoWGLI project we developed a service to render on the fly, via a complex chain of XSLT transformations, mathematical objects encoded in XML (those objects of the library of Coq that Whelp is indexing). XSLT is the standard presentational language for XML documents and an XSLT transformation (or *stylesheet*) is a purely functional program written in XSLT that describes a simple transformation between two XML formats.

The service can also render *views* (misleadingly called “theories” in [2]), that is an arbitrary, structured collection of mathematical objects, suitably assembled (by an author or some mechanical tool) for presentational purposes. In a view, definitions, theorems, and so on may be intermixed with explanatory text or figures, and statements are expanded without proofs: a link to the corresponding proof objects allows the user to inspect proofs, if desired.

Providing Whelp with an appealing user interface for presenting the answers (see Fig. 3) has been as simple as making it generate a view and pipelining Whelp with UWOBO,<sup>5</sup> the component of our architecture that implements the rendering service.

UWOBO is a stylesheet manager implemented in OCaml<sup>6</sup> and based on LibXSLT, whose main functionality is the application of a list of stylesheets (each one with the respective list of parameters) to a document. The stylesheets

<sup>4</sup> In the current implementation object names correspond to the last fragment of object URIs, without extension.

<sup>5</sup> <http://helm.cs.unibo.it/software/uwobo/>

<sup>6</sup> <http://caml.inria.fr/>

are pre-compiled to improve performance. Both stylesheets and the document are identified using HTTP URLs and can reside on any host. UWOBO is both a Web server and a Web client, accepting processing requests and asking for the document to be processed. Whelp is a Web server, accepting queries as processing requests and returning views to the client.

The Whelp interface is thus simply organized as a HTTP pipeline (see Fig. 4).

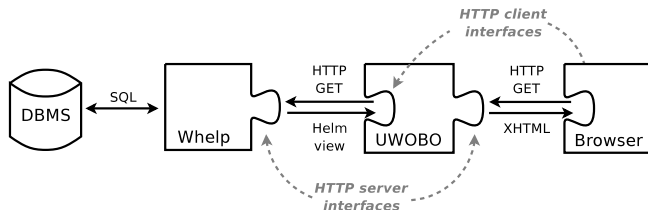


Fig. 4. Whelp's HTTP pipeline

## 7 Conclusions

Whelp is the Web searching helper developed at the University of Bologna as a part of the European Project IST-2001-33562 MoWGLI. HELP is also the acronym of the four operations currently supported by the system: Hint, Elim, Locate and Pattern-matching.

Much work remains to be done, spanning from relatively simple technical improvements, to more complex architectural re-visitations concerning the indexing technique and the design and implementation of the queries.

Among the main technical improvements which we plan to support in a near future there are:

1. the possibility to confine the search to sub-libraries, and in particular to the standard library alone (this is easy due to the paths of names);
2. skipping the annoying dialog phase with the user during disambiguation for the choice of the intended interpretation, replacing it with a direct investigation of all possibilities;
3. interactive support for advanced queries, allowing the user to directly manipulate the metadata constraints (very powerful, if properly used).

The current indexing politics has some evident limitations, resulting in unexpected results of queries.

The most annoying problem is due to the current management of Coq variables. Roughly, in Coq, *variables* are meant for *declarations*, while *constants* are meant for *definitions*. The current XML-exportation module of Coq [18] does not discharge section variables, replacing this operation with an explicit substitution mechanism; in particular, variables may be instantiated and their status look more similar to local variables than to constants. For this reason, variables have not been indexed; that currently looks as a mistake.

A second problem is due to coercions. The lack of an explicit mechanism for composition of coercions tends to clutter the terms with long chains of coercions, which in case of big algebraic developments as e.g. C-Corn [9], can easily reach about ten elements. The fact that an Object  $r$  refers a coercion  $c$  contains very little information, especially if coercions typically come in a row, as in Coq. In the future, we plan to skip coercions during indexing.

Notice that the two previous problems, both logic dependent, only affect metadata extraction and not the metadata model that is logic independent.

The final set of improvements concerns the queries. A major issue, for all kinds of content based operations, is to take care, at some extent, of delta reduction. For instance, in Coq, the elementary order relations over natural numbers are defined in terms of the *less or equal* relation, that is a suitable inductive type. Every query concerning such relations could be thus reduced to a similar one about the *less or equal* relation by delta reduction. Even more appealing it looks the possibility to extend the queries up to equational rewriting of the input (via equations available in the library).<sup>7</sup>

Similarly, the HINT operation, could and should be improved by suitably tuning the current politics for computing the *intended* signature of the search space (for instance, “closing” it by delta reduction or rewriting, adding constructors of inductive types, and so on).

Different kind of queries could be designed as well. An obvious generalization of HINT is a AUTO, automatically attempting to solve a goal by repeated applications of theorems of the library (a deeper exploration of the search space could be also useful for a better rating of the HINT results).

A more interesting example of content-based query that exploits the higher order nature of the input syntax is to look for all mathematical objects providing examples, or instances, of a given notion. For instance we may define the following property, asserting the commutativity of a generic function  $f$

$$is\_commutative := \lambda A : Set. \lambda f : A \rightarrow A \rightarrow A. \forall x, y : A. (f x y) = (f y x)$$

Then, an INSTANCE query should be able to retrieve from the library all commutative operations which have been defined.<sup>8</sup>

To conclude, we remark that the only two components of Whelp that are dependent on CIC, the logic of the Coq system, are the disambiguator for the user input and the metadata extractor. Moreover, the algorithm used for disambiguation depends only on the existence of a refiner for mathematical formulae extended with placeholders and the metadata model is logic independent. Thus Whelp can be easily modified to index and search other mathematical libraries provided that the statements of the theorems can be easily parsed (to extract the metadata) and that there exists a service to render the results of the queries.

<sup>7</sup> The possibility of considering search up to isomorphism (see [10, 11]) looks instead less interesting, because our indexing policy is an interesting surrogate that works very well in practice while being much simpler than search up to isomorphisms.

<sup>8</sup> MATCH is in fact a particular case of INSTANCE where the initial sequence of lambda abstractions is empty.

In particular, the Mizar development team has just released version 7.3.01 of the Mizar proof assistant that provides both a native XML format and XSLT stylesheets to render the proofs. Thus it is now possible to instantiate Whelp to work on the library of Mizar, soon making possible a direct comparison on the field between Whelp and MML Query, the new search engine for Mizar articles described in [6, 5]. Another interesting comparison is with the approach described in [7], which has been tested on the Mizar library.

## References

1. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, I. Schena. *The Science of Equality: some statistical considerations on the Coq library*. Mathematical Knowledge Management Symposium, 25-29 November 2003, Heriot-Watt University, Edinburgh, Scotland.
2. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, I. Schena. *Mathematical Knowledge Management in HELM*. Annals of Mathematics and Artificial Intelligence, 38(1): 27–46; May 2003.
3. A. Asperti, M. Selmi. *Efficient Retrieval of Mathematical Statements*. In Proceeding of the Third International Conference on Mathematical Knowledge Management, MKM 2004. Bialowieza, Poland. LNCS 3119.
4. A. Asperti, B. Wegner. *An Approach to Machine-Understandable Representation of the Mathematical Information in Digital Documents*. In: Fengshai Bai and Bernd Wegner (eds.): Electronic Information and Communication in Mathematics, LNCS vol. 2730, pp. 14–23, 2003
5. G. Bancerek, P. Rudnicki. *Information Retrieval in MML*. In A. Asperti, B. Buchberger, J. Davenport (eds), Proceedings of the Second International Conference on Mathematical Knowledge Management, MKM 2003. LNCS, 2594.
6. G. Bancerek, J. Urban. *Integrated Semantic Browsing of the Mizar Mathematical Repository*. In: A. Asperti, G. Bancerek, A. Trybulec (eds.), Proceeding of the Third International Conference on Mathematical Knowledge Management, Springer LNCS 3119.
7. P. Cairns. *Informalising Formal Mathematics: Searching the Mizar Library with Latent Semantics*. In Proceeding of the Third International Conference on Mathematical Knowledge Management, MKM 2004. Bialowieza, Poland. LNCS 3119.
8. The Coq proof-assistant, <http://coq.inria.fr>
9. L. Cruz-Filipe, H. Geuvers, F. Wiedijk, “C-CoRN, the Constructive Coq Repository at Nijmegen”. In: A. Asperti, G. Bancerek, A. Trybulec (eds.), Proceeding of the Third International Conference on Mathematical Knowledge Management, Springer LNCS 3119, 88-103, 2004.
10. D. Delahaye, R. Di Cosmo. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In Proceedings of TYPES 99, Lökeberg. Springer-Verlag LNCS, 1999.
11. R. Di Cosmo. *Isomorphisms of Types: from Lambda Calculus to Information Retrieval and Language Design*, Birkhauser, 1995, ISBN-0-8176-3763-X.
12. H. Ganzinger, R. Nieuwehuis, P. Nivela. *Fast Term Indexing with Coded Context Trees*. *Journal of Automated Reasoning*. To appear.
13. P. Graf. Substitution Tree Indexing. Proceedings of the 6th RTA Conference, Springer-Verlag LNCS 914, pp. 117-131, Kaiserslautern, Germany, April 4-7, 1995.

14. F. Guidi, C. Sacerdoti Coen. *Querying Distributed Digital Libraries of Mathematics*. In Proceedings of Calculemus 2003, 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning. Aracne Editrice.
15. C. McBride. *Dependently Typed Functional Programs and their Proofs*. Ph.D. thesis, University of Edinburgh, 1999.
16. W. McCune. *Experiments with discrimination tree indexing and path indexing for term retrieval*. Journal of Automated Reasoning, 9(2):147-167, October 1992.
17. C. Munoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. Ph.D. thesis, INRIA, 1997.
18. C. Sacerdoti Coen. *From Proof-Assistants to Distributed Libraries of Mathematics: Tips and Pitfalls*. In Proc. Mathematical Knowledge Management 2003, Lecture Notes in Computer Science, Vol. 2594, pp. 30–44, Springer-Verlag.
19. C. Sacerdoti Coen, S. Zacchiroli. *Efficient Ambiguous Parsing of Mathematical Formulae*. In Proceedings of the Third International Conference on Mathematical Knowledge Management, MKM 2004. LNCS, 3119.



# A Machine-Checked Formalization of the Random Oracle Model

Gilles Barthe and Sabrina Tarento

INRIA Sophia-Antipolis, France

{Gilles.Barthe, Sabrina.Tarento}@sophia.inria.fr

**Abstract.** Most approaches to the formal analysis of cryptography protocols make the perfect cryptographic assumption, which entails for example that there is no way to obtain knowledge about the plaintext pertaining to a ciphertext without knowing the key. Ideally, one would prefer to abandon the perfect cryptography hypothesis and reason about the computational cost of breaking a cryptographic scheme by achieving such goals as gaining information about the plaintext pertaining to a ciphertext without knowing the key. Such a view is permitted by non-standard computational models such as the Generic Model and the Random Oracle Model. Using the proof assistant Coq, we provide a machine-checked account of the Generic Model and the Random Oracle Model. We exploit this framework to prove the security of the ElGamal cryptosystem against adaptive chosen ciphertexts attacks.

## 1 Introduction

Cryptographic mechanisms provide a fundamental mechanism to ensure security, and are used pervasively in numerous application domains, including distributed systems and web services. However, designing secure cryptographic mechanisms is extremely difficult to achieve [1]. Therefore, there is an increasing trend to study provable security of cryptographic schemes, whereby one provides a clear specification of the security requirements, and establish with complexity-theoretic arguments that the proposed scheme meets the requirements [18]. Typically, the security of the scheme is established by showing that the attacker has a negligible advantage, i.e. that its chance of succeeding in launching an attack that exploits its capabilities is not significantly higher than its chance of breaking the scheme by brute force. While provable cryptography has become an important tool, it is not unusual to see attacks against cryptographic schemes that were deemed sound using methods from provable security; in most cases, such attacks will exploit an hidden assumption, e.g. that some event occurs with negligible probability.

The objective of our work, initiated in [4], is to machine-check results from provable cryptography. In [4], we use the proof assistant Coq [7] to establish the security of cryptographic schemes, using the Generic Model or GM for short [16, 11], which provides a non-standard computational model for reasoning about the probability and computational cost of breaking a cryptographic

scheme. Our work demonstrates that the benefits of machine-checking results from provable cryptography are two-fold: firstly, we are able to give a precise description of the models that underlie proofs in provable cryptography. Secondly, we are able to provide accurate statements about the security of cryptographic schemes, and to highlight hidden assumptions or approximations of the attacker’s advantage in published proofs. However, the formalization of [4] only focuses on non-interactive attacks where the attacker tries to break a cryptographic scheme without any interaction with oracles that perform cryptographic operations. This is an important restriction since in most practical scenarios the attacker is able to interact with oracles that provide useful information for launching an attack. Different forms of oracles include hash oracles, which allow the attacker to hash a message, decryption oracles or decryptors for short, which allow the attacker to retrieve the plaintext from (correct) ciphertexts, and signature oracles or signers for short, which allow the attacker to sign messages.

The main contribution of this paper is to extend our security proofs to an interactive setting, building on a combination of the GM and of the Random Oracle Model or ROM for short [5, 9] that assumes the hash function to be collision resistant (i.e. that collisions of random functions have negligibly small probability). As an application of our results, we prove the security of signed ElGamal encryption against strong adaptive chosen ciphertext attacks. Following [4], the key insight in our formalization is a distinction between the symbolic execution of an attack that specifies the behavior of the attacker, and the concrete execution of the attack that can lead the attacker to gain information about the secrets.

In the case of the Generic Model, the attacker tries to gain knowledge about secrets by trying to find so-called collisions, which establish a correlation between two different outputs produced during the (concrete) execution of the attack. In this setting, the distinction between the symbolic level and concrete level takes the following form:

- at the symbolic level, secrets are treated symbolically and the execution of the attack outputs polynomials  $p_1 \dots p_t$  whose indeterminates are the secrets used in the cryptographic scheme. At this level, the attacker constructs polynomials that will be used at the concrete level for gaining some information about secrets;
- at the concrete level, secrets are interpreted and the attacker checks whether collisions occur, i.e. the secrets are a root of some polynomial  $p_i - p_j$  (with  $i \neq j$ ) where  $p_i$  and  $p_j$  are taken from the polynomials  $p_1 \dots p_t$  that were constructed at the symbolic level.

In the setting of the Random Oracle Model, we must also account for interactions with oracles. We do so with the same methodology, i.e. we consider symbolic outputs that are built performing symbolic hash computations, and concrete outputs where the symbolic results of hash computations are interpreted.

*Contents of the Paper.* The remainder of the paper is organized as follows. Section 2 provides an account of the Generic Model and of the Random Oracle

Models and of their application to ElGamal. Section 3 discusses our formalization of discrete probabilities and polynomials, which are required to prove our main results. Section 4 reviews our formalization of GM. Section 5 deals with interactive attacks using a hash oracle and a decryptor, and shows an application of our results to ElGamal. We conclude in Section 6.

## 2 A Primer on Cryptography

### 2.1 Public-Key Cryptography

In public key cryptosystems, each participant gets a pair of keys, a public key and a private key. The public key is published, while the private key is kept secret. All communications involve only public keys, and no private key is ever transmitted or shared. The only requirement is that public keys to be associated with their users in a trusted (authenticated) manner (for instance, in a trusted directory). Anyone can send a confidential message by just using public information, but the message can only be decrypted with the right private key, which is in the sole possession of the intended recipient. Furthermore, public-key cryptography can be used not only for privacy (encryption), but also for authentication (digital signatures) and other various techniques [13].

In a public key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. The typical defense against this is to make the problem of deriving the private key from the public key as difficult as possible. For example, ElGamal cryptosystem assume the intractability of the Decisional Diffie Hellman problem, or DDH-problem [8] i.e., given  $g^x [p]$  and  $g^y [p]$ , it is hard to tell the difference between  $g^{xy} [p]$  and  $g^r [p]$  where  $r$  is random and  $p$  is a prime number.

The Diffie-Hellman key exchange algorithm is usually described as an active exchange of keys by two parties  $A$  and  $B$ , who have a (publicly known) prime number  $p$  and a generator  $g$ :

- party  $A$  selects a random number  $x$ , and transmits  $g^x [p]$  to  $B$ , symbolically  $A \longrightarrow B : g^x [p]$ ;
- party  $B$  selects a random number  $y$ , and transmits  $g^y [p]$  to  $A$ , symbolically  $B \longrightarrow A : g^y [p]$ ;
- both parties communicate using  $g^{xy} [p]$  as their session key.

ElGamal [10] can be considered as a special case of the Diffie-Hellman key exchange algorithm. In ElGamal, to send a message to a party whose public key is  $g^y [p]$ , we send our own public key,  $g^x [p]$ , and in addition the message is enciphered by multiplying it by  $g^{xy} [p]$  i.e., an ElGamal ciphertext has the form  $(g^y [p], mg^{xy} [p])$  for a plaintext  $m$ ; the multiplication also being modulo  $p$ .

To sign an ElGamal ciphertext, we add a Schnorr signature to the ciphertext  $(g^y [p], mg^{xy} [p])$ : pick random  $s$ , compute  $c = H(g^s, g^y, mg^{xy})$  where  $H$  is a random function chosen at random over all functions of that type with uniform

probability distribution, and compute  $z = s + cy$ , then  $(g^y, mg^{xy}, c, z)$  is the signed ciphertext.

In this paper, we prove the security of ElGamal encryption against strong adaptive chosen ciphertext attacks CCA, as described e.g. by C.Rackoff and D.Simon [12]. CCA security means that indistinguishability against an adversary that has access to a decryption oracle which it can freely use except for the target ciphertext.

## 2.2 The Generic Model

The generic model, or GM for short, was introduced by Shoup [16], building upon Nechaev [11], and can be used to provide an overall guarantee that a cryptographic scheme is not flawed [14, 15, 18]. For example, GM is useful for establishing the complexity of the discrete logarithm or the decisional Diffie-Hellman problem, which we describe below.

The GM focuses on generic attacks, i.e. attacks that do not exploit any specific weakness in the underlying mathematical structures, which in the case of GM is a cyclic group  $G$  of prime order  $q$ . More concretely, the GM focuses on attacks that work for all cyclic groups, and that are independent of the encoding of group elements; in practice, this is achieved by leaving the group  $G$  unspecified. Furthermore, the GM constrains the behavior of the attacker so that he cannot access oracles, and can only gain information about the secret through testing group equalities (a.k.a. collisions). In order to test group equalities, the attacker performs repeatedly modular exponentiations of the program inputs, using coefficients that are chosen randomly and with uniform distribution over the probability space  $\mathbb{Z}_q$ .

More precisely, a generic attacker  $\mathcal{A}$  over  $G$  is given by its list of secrets, say  $s_1, \dots, s_k \in \mathbb{Z}_q$ , its list of inputs, say  $g^{l_1}, \dots, g^{l_{t'}}$   $\in \mathbb{Z}_q$ , which depends upon secrets, and a run, which is a sequence of  $t$  multivariate exponentiation (mex) steps. For the latter, the attacker selects arbitrarily, and independently of the secrets, the coefficients  $a_{i,1}, \dots, a_{i,t'} \in \mathbb{Z}_q$  and computes for  $t' < i \leq t$  the group elements  $f_i = \text{mex}(a_{i,1}, \dots, a_{i,t'}, (g^{l_1}, \dots, g^{l_{t'}})) = \prod_{j=1}^{t'} g^{l_j a_{i,j}}$ , where  $f_j = g^{l_j}$  for  $1 \leq j \leq t'$ . The output of the run is the list  $f_1, \dots, f_t$ , from which the attacker will test for collisions, i.e. equalities  $f_j = f_{j'}$  i.e.,  $f_j - f_{j'} = 0$  with  $1 \leq j < j' \leq t$ .

Considering  $s_1, \dots, s_k$  as formal variables over  $\mathbb{Z}_q$ ,  $f_j - f_{j'}$  is a polynomial in  $\mathbb{Z}_q[s_1, \dots, s_k]$ . The random  $s_1, \dots, s_k$  are statistically independent of the coefficients  $a_{j,1}, \dots, a_{j,t'}$  and  $a_{j',1}, \dots, a_{j',t'}$ . The attacker can obtain information about the secrets by solving the equation  $f_j - f_{j'}$ .

The objective of the GM model is to establish upper bounds for the probability of a generic attacker to be successful. To this end, the GM assumes that a generic attacker  $\mathcal{A}$  is successful if it finds a non-trivial collision, i.e. a collision that reveals information about secrets (those collisions which do not reveal information are called trivial, and are defined as collisions that hold with probability 1, i.e. for all choices of secret data), or if not, if it guesses the secrets at random. Rather than considering the probability of an attacker to be successful, it is convenient to consider its advantage, which is the probability to be successful with respect to

an attacker who would try to guess secrets at random. Indeed, modeling explicitly the probability of finding secrets requires an implicit assumption about what the attacker wants to find, e.g. that he is only interested in one specific secret or in all secrets. Focusing on the attacker advantage is more general, because we do not need to specify whether the attacker is interested in finding parts or all of the secrets.

Note that the GM also makes the implicit assumption that the advantage of the attacker is reduced to the probability of finding non-trivial collisions. This assumption incurs a loss of precision in the bounds one gives (since finding a non-trivial collision may not be sufficient to reveal all secrets); however, it allows to show that the advantage of the attacker is negligible for a sufficiently large order  $q$  of the group  $G$  and a reasonable number of steps  $t$  of the run.

### 2.3 The Random Oracle Model

Interactive generic algorithms are an extension of generic algorithms in which the attacker is able to interact with oracles through interactive steps. Such interactive algorithms can be modeled using the Random Oracle Model, or ROM for short, that was introduced by Bellare and Rogaway [5] but its idea originates from earlier work by Fiat and Shamir [9].

For the purpose of our work, we do not need to develop a general framework for interactions; instead we focus on two typical oracles with whom the attacker can interact: queries to hash functions and decryptors. These forms of interaction are used in particular in the signed ElGamal encryption protocol.

To sign a message, we do an interaction with a hash oracle i.e., a hash function  $H : G \rightarrow M \rightarrow \mathbb{Z}_q$  where  $M$  is the set of messages. Cryptographic hash functions are used in various contexts, for example, to compute the message digest when making a digital signature. A hash function compresses the bits of a message to a fixed-size hash value in a way that distributes the possible messages evenly among the possible hash values. A cryptographic hash function does this in a way that makes it extremely difficult to come up with a message that would hash to a particular hash value. The ROM assumes a random hash function and is a stronger assumption than assuming the hash function to be collision resistant; the fundamental assumption of ROM is that the hash function  $H : G \rightarrow M \rightarrow \mathbb{Z}_q$  is chosen at random with uniform probability distribution over all functions of that type. Note that interactions provide the algorithm with values, and that, in this setting, mex-steps perform computations of the form  $f_i = \prod_{1 \leq j \leq t'} g^{l_j a_{i,j}}$ , where for  $1 \leq j \leq t'$ ,  $g^{l_j}$  is an input of the algorithm, and where  $a_{i,1}, \dots, a_{i,t'}$  are arbitrary but may depend on values that the algorithm received through interactions; for  $1 \leq i \leq t''$ ,  $f_i$  is a group output of the algorithm so we assume it to do  $t''$  mex-steps. Like in the non interactive case, we consider that the interactive generic adversary takes a list of secrets  $s_1, \dots, s_k$  and a list of inputs  $g^{l_1}, \dots, g^{l_{t'}}$ .

*Example 1.* Let  $x \in \mathbb{Z}_q$  and  $h = g^x$  be the private and public keys for encryption,  $m \in G$  the message to be encrypted. For encryption, pick random  $r \in \mathbb{Z}_q$ ,

$(g^r, mh^r)$  is the ElGamal ciphertext. To add Schnorr signatures, pick random  $s \in \mathbb{Z}_q$ , compute  $c = H(g^s, g^r, mh^r)$  and  $z = s + cr$ , then  $(g^r, mh^r, c, z)$  is the signed ciphertext. A decryptor  $Dec$  takes a claimed ciphertext  $(\bar{h}, \bar{f}, c, z)$  and computes

$$F = (if H(g^z \bar{h}^{-c}, \bar{h}, \bar{f}) = c \text{ then } \bar{h}^x \text{ else } ?)$$

where  $?$  is a random value, and then returns  $\frac{\bar{f}}{F}$  which is the original message, if  $(\bar{h}, \bar{f}, c, z)$  is a valid ciphertext.

A decryptor should not decrypt the target ciphertext because if the attacker sends to the decryptor the target ciphertext, the equality  $H(g^z \bar{h}^{-c}, \bar{h}, \bar{f}) = c$  is always verified and so the attacker obtains immediately the original message.

As in the non-interactive model, an attacker is a generic algorithm that seeks to gain knowledge about secrets through testing equalities between the group elements it outputs, possibly through interactions. However, the attacker has now access to oracles for computing hash values and for decryption. Note that each operation performed by the attacker, i.e. reading an input, performing an interaction, or taking a mex-step, counts as a step in the run. However, as in the non-interactive case, testing equality is free. The adversary's advantage is the probability that the adversary finds non-trivial collisions among computed group elements plus the probability that the adversary obtains information on secrets through an interaction with the decryptor.

In the remaining of this subsection, we explain more precisely how the attacker can get information about the secrets by an interaction with the decryptor. Each interaction with the decryptor yields a polynomial and we can obtain information on the secrets if we find the zero of this polynomial.

Recall that a Schnorr signature on a message  $m$  is a triple  $(m, c, z) \in M \times \mathbb{Z}_q^2$  such that  $H(g^z h^{-c}, m) = c$  and let  $(f_i, f_j, c, z)$  be the claimed ciphertext that  $\mathcal{A}$  transmits to the decryptor. In the ROM, the equation  $c = H(g^z h^{-c}, f_i, f_j)$  required for a valid signature, necessitates that  $\mathcal{A}$  selects  $c$  from the given hash values  $H(f_\sigma, f_i, f_j)$  for given group elements  $f_\sigma, f_i, f_j$ . The attacker gets  $c = H(g^z h^{-c}, f_i, f_j)$  from the hash oracle and must compute  $z$  so that  $g^z h^{-c} = f_\sigma$ , i.e., it must compute  $z = \log_g(f_\sigma f_j^c)$ . The computed  $z$  i.e., the element  $z$  used for an interaction with the decryptor (recall that a decryptor takes as input a quadruple  $(f_i, f_j, c, z)$ ), does not depend on the secrets  $s_1, \dots, s_k$  whereas  $z' = \log_g(f_\sigma f_j^c) = \log_g f_\sigma + c \log_g f_j$ , which denotes the value required for a signature, may depend on it.

The group steps of the iterative generic algorithm refer to the given group elements  $l_1, \dots, l_{t'}$ . The adversary computes  $f_i := \prod_{1 \leq j \leq t'} g^{l_j a_{i,j}}$  for  $i = 1, \dots, t$  using exponents  $a_{i,1}, \dots, a_{i,t'} \in \mathbb{Z}_q$  that arbitrarily depend on values that the algorithm received through interactions but not on the secrets  $s_1, \dots, s_k$ . Hence  $z'$  is of the form:

$$\begin{aligned} z' &= \log_g(f_\sigma f_j^c) \\ &= \langle a_\sigma + ca_j, (l_1, \dots, l_{t'}) \rangle \end{aligned} \tag{1}$$

where  $\langle \cdot, \cdot \rangle$  is a scalar product i.e.,  $\langle (a_1, \dots, a_n), (c_1, \dots, c_n) \rangle = \sum_{j=1}^n a_j c_j$ .

Considering  $s_1, \dots, s_k$  as formal variables over  $\mathbb{Z}_q$ ,  $z'$  is a polynomial in  $\mathbb{Z}_q[s_1, \dots, s_k]$  (as the inputs  $l_1, \dots, l_{t'}$  are polynomials in  $\mathbb{Z}_q[s_1, \dots, s_k]$ ). The random  $c, s_1, \dots, s_k$  are statistically independent of the coefficients  $a_{\sigma,1}, \dots, a_{\sigma,t'}$  and  $a_{j,1}, \dots, a_{j,t'}$ .  $\mathcal{A}$  can obtain information about the secrets by solving the equation  $z' = z$  i.e., the polynomial  $z'$  must be equal to the computed group element  $z$ , so we must have  $z' - z = 0$ . Let us notice that the value required for a signature i.e.,  $z'$  depends on the secrets, so we note  $z'(s_1, \dots, s_k)$  instead  $z'$  to make the difference with the value computed by the algorithm i.e.,  $z$  which is a constant in  $\mathbb{Z}_q[s_1, \dots, s_k]$ . The equation  $z' - z = 0$  is seen as a polynomial equality  $z'(s_1, \dots, s_k) - z \equiv 0$  for the secrets  $s_1, \dots, s_k$ . Each interaction with the decryptor provides a polynomial  $z'(s_1, \dots, s_k) - z$ , thus after  $l$  interactions with the decryptor, we have a list of  $l$  polynomials  $z'(s_1, \dots, s_k) - z$ , so we can obtain informations about the secrets if we can find a zero of a polynomial that belongs to this list. In fact, an interaction with the decryptor succeeds if the equation  $z'(s_1, \dots, s_k) - z = 0$  holds; and by applying Schwartz lemma (see Section 3.3) to the polynomial  $z'(s_1, \dots, s_k) - z$ , we get a bound to the probability of finding the secrets  $s_1, \dots, s_k$ . To conclude, we eliminate interactions with the decryptor by computing extractor for each interaction with the decryptor.

## 2.4 Applications of GM+ROM

We consider the application to the signed ElGamal encryption; let the attacker be given the generator  $g$ , the public key  $h = g^x$ , distinct messages  $m_0, m_1$ , a target signed ciphertext  $cip_b = (g^r, m_b g^{rx}, c, z)$  corresponding to  $m_b$  for a random bit  $b \in \{0, 1\}$  and oracles for the hash function  $H$  and for decryption. Then an interactive generic adversary using  $t$  generic steps including  $t''$  operations on group elements and  $l$  interactions with the decryptor, can not predict  $b$  with a better probability than  $\frac{1}{2} + \frac{2\binom{t}{2}}{q-2\binom{t}{2}}$ . The probability space consists of the random  $x, b, H$  and the key of the encipherer  $r$ .

To prove this, we use Schwartz lemma (see Section 3.3), the bound of the probability of finding collisions among the computed group elements i.e., non trivial collisions occur with no better probability than  $\frac{2\binom{t''}{2}}{q-2\binom{t''}{2}}$  if we compute  $t''$  group elements. For this bound, the probability space refers to the random  $b, r, x$ ; and we use the bound of the probability of finding information on the plaintext by having  $l$  interactions with the decryptor:  $\frac{3l}{q}$ .

## 3 Remarks on the Formalization of Algebra

This section provides a brief discussion on some issues with the formalization of the mathematical concepts that underlie the generic and random oracle models. We focus on two particularly important issues, namely the formalization of discrete probabilities and of multivariate polynomials.

### 3.1 Sets and Algebra

Standard presentations of the generic model involve a cyclic group  $G$  of order  $q$ , and formalizing the generic model directly in Coq would therefore require that we use a formalization of groups. Although several such formalizations are already provided by the Coq contributions, we find it convenient to exploit the canonical isomorphism between the group  $G$  and the ring  $\mathbb{Z}_q$  (assuming that  $g$  is a generator of the group, the function  $x \mapsto g^x$  is an isomorphism from  $\mathbb{Z}_q$  to  $G$ ) and work directly with  $\mathbb{Z}_q$  instead of  $G$ . Thus, instead of considering an element  $g^a$  of the group, we will always consider the exponent  $a$ , and instead of considering multivariate exponentiation  $mex : \mathbb{Z}_q^d \rightarrow G^d \rightarrow G$  as done in the generic model  $(a_1, \dots, a_d), (g_1, \dots, g_d) \mapsto \prod_{i=1}^d g_i^{a_i}$  we use the function  $mex : \mathbb{Z}_q^d \rightarrow \mathbb{Z}_q^d \rightarrow \mathbb{Z}_q$  defined as  $(a_1, \dots, a_d), (s_1, \dots, s_d) \mapsto \sum_{j=1}^d a_j s_j$  where it is assumed that  $g_i = g^{s_i}$ .

Dispensing with the formalization of groups does not allow us to dispense with the formalization of sets, which are represented using setoids [3, 2]. For the sake of readability, we avoid in as much as possible mentioning setoids in our presentation, although they are pervasive in our formalizations.

### 3.2 Probabilities

There are several possible formalizations of probabilities in Coq. Our choice of formalization was influenced by two important factors. The first factor is a simplifying factor, namely for our purposes we only need to consider discrete probabilities, i.e. probabilities over finite sets. The second factor is a complicating factor, namely for our purposes we need to consider probabilities over setoids.

On this basis, we have found convenient to define probabilities w.r.t. an arbitrary type  $V$  and a finite subset  $E$  of  $V$ , given as a (non-repeating and non-empty)  $V$ -list; intuitively,  $E$  is meant to contain exactly one representative from each equivalence class generated by the setoid underlying equality. Then the probability of an event  $A$ , i.e. of a predicate over  $V$ , is defined as the ratio between the number of elements in  $E$  for which  $A$  holds and the total number of elements in  $E$ , i.e.

**Definition** `Event := V → Prop`

**Definition** `PrE(A:Event) := length (filter E A) / (length E)`.

One can check that  $Pr_E$  satisfies the properties of a probability measure, and define derived notions such as conditional probabilities. In the sequel,  $E$  will often be omitted to adopt the notation  $Pr(A)$ .

### 3.3 Polynomials

There have been several formalizations of polynomials in Coq. Our choice of formalization was guided by the proof of Schwartz Lemma (see below), which requires to view a polynomial in  $n + 1$  variables as a polynomial in  $n$  or 1 variables. Our current formalization (which is different from the formalization



used in [4] and in our opinion more elegant) uses the Horner representation of polynomials for polynomials in one variable, and define polynomials in  $n + 1$  variables recursively from polynomials in  $n$  variables. Formally, we consider a ring  $R$  with underlying carrier  $C$  and define  $R[X]$  as the inductive type:

**Inductive**  $R[X]$  : **Type** :=  
 | Pc :  $C \rightarrow R[X]$   
 | P<sub>X</sub> :  $R[X] \rightarrow C \rightarrow R[X]$ .

where  $Pc$  is the constructor for constant polynomials and  $P_X \quad \alpha \quad d = \alpha * X + d$ . Once monovariate polynomials have been formalized, polynomials in  $n + 1$  variables are built recursively from polynomials in  $n$ -variables, using the canonical isomorphism between  $R[X_1, \dots, X_{n+1}]$  and  $(R[X_1, \dots, X_n])[X_{n+1}]$ .

Then one can define equality  $\equiv$  between polynomials using an appropriate inductive relation, and endow  $R[X]$  and  $R[X_1, \dots, X_{n+1}]$  with a ring structure under the usual operations of polynomial addition, subtraction, multiplication, etc. Using this formalization, we have proved the following lemma, which provides an upper bound on the probability of a vector to be a root of a polynomial of degree  $d$  over  $\mathbb{Z}_q[X_1, \dots, X_n]$ , and which is the key result that enables security proofs in the Generic Model.

**Lemma 1 (Schwartz Lemma).**

$$\forall (p : \mathbb{Z}_q[X_1, \dots, X_n], \quad q \neq 0 \rightarrow p \neq 0 \rightarrow Pr_{x_1, \dots, x_n \in \mathbb{Z}_q^n} (p(x_1, \dots, x_n) = 0) \leq (\text{degree } p) / q.$$

The lemma requires that  $q$  is not null and that  $p$  is not identically null, and establishes that the probability that an element  $x \in \mathbb{Z}_q^n$  is a zero of a polynomial  $p$  is smaller than the degree of the polynomial divided by  $q$ . The proof proceeds by induction on the number  $n$  of variables. Note that we slightly abuse notation and write  $Pr_{x_1, \dots, x_n \in \mathbb{Z}_q^n} (p(x_1, \dots, x_n) = 0)$  for  $Pr_{Var \rightarrow \mathbb{Z}_q} (\lambda f : Var \rightarrow \mathbb{Z}_q. (p \ f) = 0)$ , where  $Var$  is the finite set with inhabitants  $X_1, \dots, X_n$ .

We now state corollaries of Schwartz lemma that are used in Section 5. The first corollary follows rather directly from Schwartz lemma, while the second corollary is proved by induction.

**Lemma 2.**  $\forall (d : \mathbb{N}) (p_1, \dots, p_n : \mathbb{Z}_q[X_1, \dots, X_n], \quad q \neq 0 \rightarrow (\forall 1 \leq j \leq n, (\text{degree } p_j) \leq d \wedge p_j \neq 0) \rightarrow Pr_{x_1, \dots, x_n \in \mathbb{Z}_q^n} (p_n(x_1, \dots, x_n) = 0 \mid \forall j < n, p_j(x_1, \dots, x_n) \neq 0) \leq d / (q - nd)$ .

**Lemma 3.**  $\forall (d : \mathbb{N}) (p_1, \dots, p_n : \mathbb{Z}_q[X_1, \dots, X_n], \quad q \neq 0 \rightarrow (\forall 1 \leq j \leq n, (\text{degree } p_j) \leq d \wedge p_j \neq 0) \rightarrow nd < q \rightarrow Pr_{x_1, \dots, x_n \in \mathbb{Z}_q^n} (p_1(x_1, \dots, x_n) = 0 \vee \dots \vee p_n(x_1, \dots, x_n) = 0) \leq nd / (q - nd)$ .

## 4 A Review of the Formalization of the Generic Model

The main difficulty in formalizing generic algorithms is to pinpoint the notion of secret. As mentioned in the introduction, we take advantage of the expressiveness of our framework and introduce an abstract type *Sec* of secrets together

```

1 Parameter Sec:Set.
2 Parameter input:list  $\mathbb{Z}_q[Sec]$ .
3
4 Inductive GA:Type:=
5   nostep:GA
6   | step:GA  $\rightarrow$  (list  $\mathbb{Z}_q$ )  $\rightarrow$  GA.
7
8 Fixpoint SymbOutput( $\mathcal{A}$ :GA):(list  $\mathbb{Z}_q[Sec]$ ):=
9   match  $\mathcal{A}$  with nostep  $\Rightarrow$  nil
10  | (step  $\mathcal{A}'$  e)  $\Rightarrow$  (mex e input)::(SymbOutput  $\mathcal{A}'$ )
11  end.
12
13 Definition ConcrOutput( $\mathcal{A}$ :GA)( $\sigma$ :Sec  $\rightarrow$   $\mathbb{Z}_q$ ):list  $\mathbb{Z}_q$ :=
14  map  $\lambda x:\mathbb{Z}_q[Sec].[[x]]_\sigma$  (SymbOutput  $\mathcal{A}$ ).
15
16 Definition CO( $\mathcal{A}$ :GA)( $\sigma$ :Sec  $\rightarrow$   $\mathbb{Z}_q$ ):=
17   $\forall e e':\mathbb{Z}_q[Sec], e \in$  (SymbOutput  $\mathcal{A}$ )  $\wedge$ 
18   $e' \in$  (SymbOutput  $\mathcal{A}$ )  $\wedge e - e' \neq 0 \wedge [[e - e']]_\sigma = 0$ .

```

**Fig. 1.** Formalization of the GM

with an interpretation function from  $Sec$  to  $\mathbb{Z}_q$ . In order to fix terminology, we refer to elements of  $Sec$  as symbolic secrets and to their interpretation in  $\mathbb{Z}_q$  as (concrete values of) secrets. Then, we define generic algorithms which describe the behavior of the attacker at an abstract level; being defined at an abstract level, the behavior of the attacker is independent of the concrete values of the secret, as required by the GM.

Generic algorithms may be executed symbolically, producing symbolic outputs in the form of polynomials. Further, abstract runs are interpreted into concrete runs; the latter correspond to executing an attack and may or not be successful, depending on whether a non-trivial collision is found.

The formal definition of a generic algorithm is given in Figure 1. In order to model the notion of secrets, we introduce a type  $Sec$  of formal secret parameters (see line 1) and model inputs as a list of non-repeating polynomial expressions over secrets (see line 2); note that inputs are determined by the cryptographic system under consideration, and are known to the attacker. In the above we implicitly assume that the set  $Sec$  is modeled as a finite type with  $k$  secrets  $s_1, \dots, s_k$  and we use  $\mathbb{Z}_q[Sec]$  as a shorthand for  $\mathbb{Z}_q[s_1, \dots, s_k]$ .

Then, we consider generic algorithms (see line 4) in which the attacker selects arbitrarily and independently of the secrets a list of coefficients  $a_{i,1}, \dots, a_{i,t'} \in \mathbb{Z}_q$ . Generic algorithms can be executed to produce symbolic outputs (see line 8), and concrete outputs (see line 13) are obtained from the symbolic outputs by using the extension of an interpretation function  $\sigma$  from polynomial expressions to elements in  $\mathbb{Z}_q$ , more precisely,  $[[\ ]]_\sigma : \mathbb{Z}_q[Sec] \rightarrow \mathbb{Z}_q$  returns the evaluation of a polynomial in  $\mathbb{Z}_q[Sec]$  by using an interpretation function  $\sigma$ . An interpretation function  $\sigma : Sec \rightarrow \mathbb{Z}_q$  maps formal secret parameters to actual secrets in  $\mathbb{Z}_q$ .

Now we can define a non-trivial collision (see line 16) as a pair of polynomials  $e$  and  $e'$  (found in `(SymbOutput r)`) that are non identically equal and such that the interpretation of the polynomial  $e-e'$  under  $\sigma$  is 0. By considering only polynomials non identically equal, we eliminate trivial collisions.

In order to give an upper bound for the probability of finding non-trivial collisions, one relies on Schwartz Lemma (see Section 3.3). In the sequel, we let  $d$  be the maximal degree of the inputs i.e., the polynomials  $l_j$  for  $1 \leq j \leq t'$ , let  $t$  be the number of steps  $\mathcal{A}$  performs.

**Proposition 1.**  $\forall \mathcal{A} : \mathcal{GA}, Advantage(\mathcal{A}) = Pr(\mathcal{CO}(\mathcal{A})) \leq \frac{\binom{t}{2}d}{q - \binom{t}{2}d}$

In the non-interactive setting, we consider that the advantage of the attacker is bounded by the probability of finding non-trivial collisions. Such an over-approximation is quite coarse since we consider the attacker to be successful whenever he gains some informations about the interpretation function  $\sigma$ . In principle, one could try to be more precise and estimate the probability of the attacker to find the function  $\sigma$  (i.e. its value for all inputs). However, we only want to show that the advantage is negligible if the number of steps performed by the attacker is reasonable, and hence the over-approximation is justified.

In [4], we instantiate the proposition to specific cryptographic schemes.

## 5 Formalization of the Random Oracle Model

### 5.1 Formalization

The main difficulty in formalizing interactive algorithms is to capture the idea of random hash function. Following the idea of the generic model, we consider a symbolic representation of the interactions with the hash oracle by introducing a type *Val* of random variables that will represent the results of the interactions with the hash oracle. In addition, we define an interpretation function from *Val* to  $\mathbb{Z}_q$ . In order to fix terminology, we will refer to elements of *Val* as symbolic hash values and to their interpretation as hash results.

The formal definition of interactive generic algorithms is given in Figure3. As explained above, we introduce a type *Val* of symbolic hash results and a type *Sec* of symbolic secrets (see line 1). Then, we model inputs as a non-repeating list of polynomial expressions over secrets (see line 2).

In order to model ciphertexts, we introduce a type of symbolic group elements *SymbG*, defined as the type of  $\mathbb{Z}_q$ -lists (see line 4). Symbolic group elements are intuitively assumed to have a length that matches the length of the list of inputs  $(l_1, \dots, l_{t'})$ , and the list  $(a_1, \dots, a_{t'})$  represents the polynomial  $\sum_{j=1}^{t'} a_j l_j$ . In other words, symbolic group elements correspond to linear combinations of the inputs. Then we define symbolic ciphertexts as pairs of symbolic group elements, by analogy with ElGamal ciphertexts that have the form  $(g^r, mg^{rx})$  where  $g^r$  and  $mg^{rx}$  are group elements (see line 5). Finally, symbolic hash queries are defined as triples of the form  $(g, m, v)$  where  $g$  is a group element,  $m$  is a message, and

$v$  is the symbolic hash result, by analogy with hash queries that have the form  $H(a, (b, d))$ .

Interactive generic algorithms are defined inductively (see line 8) and may consist of an empty step, or a mexstep i.e., a computation of group elements using the function *mex*, or an hashstep i.e., a query to the hash oracle, or a decstep i.e., an interaction with the decryptor. A few words are in order to explain the type of the constructor *decstep*: first of all, observe that in analogy with ElGamal decryptor that takes a claimed ciphertext  $(\bar{h}, \bar{f}, c, z)$ , our formalization considers that an interaction with the decryptor requires a symbolic hash query  $I = (f_j, \bar{h}, \bar{f}, c)$  and an element  $z$  of  $\mathbb{Z}_q$ .

Interactive algorithms have two kinds of outputs:

- symbolic hash outputs (see line 14) are just a list of elements of type *SymbH*. Then we can derive the list of concrete hash outputs (see line 22) by applying an interpretation function  $\tau$ ;
- symbolic group outputs (see line 25) are polynomials constructed as linear combinations of inputs in the same way of non-interactive generic algorithms. Concrete group outputs (see line 33) are obtained from the symbolic outputs by using the extension of an interpretation function  $\sigma$  from secrets to elements in  $\mathbb{Z}_q$ .

In an interactive generic algorithm, the attacker might gain knowledge about secrets either through collisions, or through interactions with the decryptor. Thus its advantage will be bounded by the probability of finding a collision plus the probability of performing a successful interaction. In the latter case, we show that the attacker can only obtain information if the interpretation function is solution to a polynomial equation derived from the equality tested by the decryptor, i.e.  $c = H(g^z \bar{h}^{-c}, \bar{h}, \bar{f})$ , where  $(\bar{h}, \bar{f}, c, z)$  is the claimed ciphertext received by the decryptor. Note that we do not need to formalize the result returned by the decryptor, which is random in case the claimed ciphertext does not verify the above equality, since we are only interested in the probability to learn information about secrets.

To eliminate interactions with the decryptor, we formalize an extractor (see Figure 2). Let us remember that we can obtain information on the secrets by an interaction with the decryptor if  $z'(s_1, \dots, s_k) - z = 0$  holds, where  $z'(s_1, \dots, s_k)$  is the value required for a signature and  $z$  is the computed group element (see the explications in the section 2.3). More precisely,  $\mathcal{A}$  gets the hash value  $c = H(g^z h^{-c}, f_i, f_j)$  from the hash oracle and must compute  $z'$  so that  $g^{z'} h^{-c} = f_\sigma$ , i.e., it must compute  $z' = \log_g(f_\sigma f_j^c)$  (see line 1). For each *decstep*,  $\mathcal{A}$  can obtain informations about the secrets  $s_1, \dots, s_k$  by finding a zero of the polynomial  $z'(s_1, \dots, s_k) - z$  (see line 7). After  $l$  interactions with the decryptor, we have a list of  $l$  polynomials  $z'(s_1, \dots, s_k) - z$  (see line 9). So we can find informations about the secrets (see line 16) if there exists a polynomial  $p$  in the list of  $l$  polynomials  $z'(s_1, \dots, s_k) - z$  such that its interpretation under the interpretation function  $\sigma$  is 0.

Let us notice that in our model, as we do not formalize the result of the decryptor but we consider an extractor that tries to find informations on secrets

```

1 Definition z' (h:SymbH) (τ:Val → $\mathbb{Z}_q$ ): $\mathbb{Z}_q$ [Sec]:=
2 let h:=(fσ, fi, fj, c) in
3 let loggfσ :=(mk_pol fσ input) in
4 let loggfi :=(mk_pol fi input) in
5 loggfσ +(τ c)* loggfi .
6
7 Definition Extract(h:SymbH) (z: $\mathbb{Z}_q$ ) (τ:Val → $\mathbb{Z}_q$ ): $\mathbb{Z}_q$ [Sec]:=(z' h τ)-z.
8
9 Fixpoint list_Extract(A:IGA) (τ:Val → $\mathbb{Z}_q$ ):list  $\mathbb{Z}_q$ [Sec]:=
10 match A with nostep ⇒ nil
11     | mexstep A' _ ⇒ list_Extract A' τ
12     | hashstep A' _ ⇒ list_Extract A' τ
13     | decstep A' h z ⇒(Extract h z τ)::(list_Extract A' τ)
14 end.
15
16 Definition Extractor(A:IGA) (τ:Val → $\mathbb{Z}_q$ ) (x:Sec → $\mathbb{Z}_q$ ):Prop:=
17 ∀ p: $\mathbb{Z}_q$ [Sec], p ∈ (list_Extract A τ) ∧ [|p|]σ≡0.

```

**Fig. 2.** Formalization of an Extractor

by finding the zeros of the polynomial  $z'(s_1, \dots, s_k) - z \equiv 0$ , if the attacker sends the target ciphertext to the decryptor, in fact this leads to a trivial polynomial equality  $z'(s_1, \dots, s_k) \equiv z$  and so we do not obtain informations on secrets.

## 5.2 Properties of Interactive Generic Algorithm

In this section, we prove the security of cryptographic protocols like ElGamal against a strong adaptive chosen ciphertexts attack by giving an upper bound of the probability for an interactive adversary to find information about secrets.

We consider an interactive generic algorithm  $\mathcal{A}$  with inputs polynomials  $l_1, \dots, l_{t'}$  with maximal degree  $d$ . Furthermore we assume that  $\mathcal{A}$  performs  $t$  generic steps including  $t''$  mex-steps and  $l$  interactions with the decryptor.

**Proposition 2.**  $\forall \mathcal{A} : IGA, Pr(\mathcal{CO}(\mathcal{A})) \leq \frac{\binom{t''}{2}d}{q - \binom{t''}{2}d}$

*Proof.* All outputs are of the form  $p_i = \sum_{1 \leq j \leq t'} a_{i,j} l_j(s_1, \dots, s_k)$ , where  $p_i$  is a polynomial of degree  $d$ . Hence there exists a collision  $f_i = f_{i'}$  iff  $(s_1, \dots, s_k)$  is a root of  $p_i - p_{i'}$ . There are  $\binom{t''}{2}$  equalities of the form  $f_i = f_{i'}$  to test, hence  $\binom{t''}{2}$  polynomials of the form  $p_i - p_{i'}$ , each of which is not identical to 0 (as there are non-trivial collisions), and has degree  $\leq d$ . So we can apply Lemma 3 (the extension of Schwartz Lemma) to deduce the expected result.

**Proposition 3.**  $\forall \mathcal{A} : IGA, Pr(Extractor(\mathcal{A})) \leq \frac{(d+1)l}{q}$

*Proof.* The proof is by induction of the interactive generic algorithm  $\mathcal{A}$ . The only interesting case is when the algorithm interacts with the decryptor. In this

case, the interaction is successful iff  $\tau$  is a solution of the extracted polynomial, which is of degree  $d$ .

In the interactive setting, we consider that the advantage of the attacker is bounded by the probability of finding non-trivial collisions plus the probability of finding a zero of a polynomial resulting on an interaction with the decryptor.

**Proposition 4.**  $\forall \mathcal{A} : IGA, Advantage(\mathcal{A}) = Pr(\mathcal{CO}(\mathcal{A})) + Pr(Extractor(\mathcal{A}))$   

$$\leq \frac{\binom{t''}{2}^d}{q - \binom{t''}{2}^d} + \frac{(d+1)l}{q}$$

### 5.3 Application to Signed ElGamal Encryption

We can instantiate the propositions to specific cryptographic schemes. For example, we prove the security of signed ElGamal encryption against a strong adaptive chosen ciphertexts attack. We consider ElGamal protocol, be given in input: the generator  $g$ , the public key  $h$ , distinct messages  $m_0, m_1$ , a target ciphertext  $cip_b$  corresponding to  $m_b$  for a random bit  $b \in \{0, 1\}$  i.e., the concrete inputs are the list  $(g, g^x, g^r, m_b g^{rx})$ , so the formal inputs are the logarithm of the elements of the list of concrete inputs, i.e the list  $(1, x, r, \log_g m_b + rx)$ . Therefore, the secrets are  $b \in \{0, 1\}$  and  $r, x \in \mathbb{Z}_q$ . By consequent, the maximal degree of the inputs is  $d := 2$ . In this example, the advantage of the attacker is bounded by  $\frac{2\binom{t}{2}}{q-2\binom{t}{2}}$  because  $Pr(\mathcal{CO}(\mathcal{A})) \leq \frac{2\binom{t''}{2}}{q-2\binom{t''}{2}}$  and  $Pr(Extr(\mathcal{A})) \leq \frac{3l}{q}$ .

### 5.4 Remarks on the Formalization

The results of this paper have been formalized and proved in Coq. The formalization is available from:

<http://www-sop.inria.everest/proofs/acces/>

The formalizations, proofs and applications represent 17, 515 lines of code which are split as follows:

- discrete probabilities: 4, 077 lines of code. This includes the lots of useful lemmas e.g. for rewriting probabilities;
- polynomials: 4, 196 lines of code. This includes the representation of polynomials and the proof that they form a ring;
- basic libraries: 4, 545 lines of code. This includes libraries for lists over type,  $\mathbb{Z}_q$ , etc.
- GM+ROM: 6, 253 lines of code. This includes the formalization of GM and ROM, a proof of Schwartz lemma, its extensions, and its applications to GM and ROM.

Our development is modular, and can be instantiated to other cryptosystems based on cyclic groups. To prove the security of such systems, we just have to give the list of inputs, a generator and a set of secrets, as illustrated by the instantiation of our results to ElGamal.

```

1 Parameter Sec Val:Set.
2 Parameter input:list  $\mathbb{Z}_q[Sec]$ .
3
4 Definition SymbG:=list  $\mathbb{Z}_q$ .
5 Definition SymbM:=SymbG * SymbG.
6 Definition SymbH:=SymbG * SymbM * Val.
7
8 Inductive IGA : Type :=
9   | nostep : IGA
10  | mexstep : IGA → (list  $\mathbb{Z}_q$ ) → IGA
11  | hashstep : IGA → SymbH → IGA
12  | decstep : IGA → SymbH  $\rightarrow$   $\mathbb{Z}_q$  → IGA.
13
14 Fixpoint SymbHashOutput( $\mathcal{A}$ :IGA):(list SymbH):=
15   match  $\mathcal{A}$  with
16   | nostep  $\Rightarrow$  nil
17   | mexstep  $\mathcal{A}'$  e  $\Rightarrow$  SymbHashOutput  $\mathcal{A}'$ 
18   | hashstep  $\mathcal{A}'$  h  $\Rightarrow$  h::(SymbHashOutput  $\mathcal{A}'$ )
19   | decstep  $\mathcal{A}'$  _  $\Rightarrow$  SymbHashOutput  $\mathcal{A}'$ 
20 end.
21
22 Definition ConcrHashOutput( $\mathcal{A}$ : IGA)( $\tau$ :Val  $\rightarrow$   $\mathbb{Z}_q$ ):(list  $\mathbb{Z}_q$ ):=
23 map  $\lambda(x,y,z,t).\tau$  t (SymbHashOutput  $\mathcal{A}$ ).
24
25 Fixpoint SymbMexOutput( $\mathcal{A}$ :IGA): list  $\mathbb{Z}_q[Sec]$ : =
26   match  $\mathcal{A}$  with
27   | nostep  $\Rightarrow$  nil _
28   | mexstep  $\mathcal{A}'$  e  $\Rightarrow$  (mex e input)::(SymbMexOutput  $\mathcal{A}'$ )
29   | hashstep  $\mathcal{A}'$  _  $\Rightarrow$  SymbMexOutput  $\mathcal{A}'$ 
30   | decstep  $\mathcal{A}'$  _  $\Rightarrow$  SymbMexOutput  $\mathcal{A}'$ 
31 end.
32
33 Definition ConcrMexOutput( $\mathcal{A}$ : IGA)( $\sigma$ :Sec  $\rightarrow$   $\mathbb{Z}_q$ ):(list  $\mathbb{Z}_q$ ):=
34 map  $\lambda x:\mathbb{Z}_q[Sec].[x]_{\sigma}$  (SymbMexOutput  $\mathcal{A}$ ).

```

Fig. 3. Formalization of ROM

## 6 Conclusion

Provable cryptography aims at establishing rigorous security proofs for cryptographic schemes and appeals to involved complexity-theoretic arguments to show that the advantage of an attacker (over an attacker that proceeds by brute force) is negligible. Whereas provable cryptography provides an overall guarantee of the correctness of cryptographic schemes, it is not unusual for security proofs to contain glitches or to rely on hidden assumptions which open the way for attacks. In this perspective, it is very important to provide machine-checked proofs of the main results in provable cryptography. Dually, formalizing provable

cryptography is interesting from the perspective of machine-checked mathematics because it relies on many mathematical theories of interest, including discrete probabilities, polynomials, and linear algebra.

In this paper, we have extended our previous machine-checked account of the GM to include the ROM and to establish security bounds for interactive algorithms. In another related work [19], we provide a machine-checked treatment of signature forgery attacks, as reported in [14]. These results generalize previous work, and give more rigorous bounds than those present in the literature. Nevertheless, machine-checked proofs of provable cryptography has barely been scratched, and much work remains to be done. For example, we would like to exploit our formalization to prove the security of realistic protocols, following e.g. [6, 17], and eventually perhaps to provide a machine-checked account of a formalism that integrates the computational view of cryptography, and provable cryptography.

*Acknowledgments.* We are grateful to the anonymous referees for their constructive and detailed comments.

## References

1. M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *Transactions on Software Engineering*, 22(1):6–15, January 1996.
2. H. Barendregt and H. Geuvers. Proof assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 18, pages 1149–1238. Elsevier Publishing, 2001.
3. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13:261–293, March 2003.
4. G. Barthe, J. Cederquist, and S. Tarento. A Machine-Checked Formalization of the Generic Model and the Random Oracle Model. In D. Basin and M. Rusinowitch, editors, *Proceedings of IJCAR’04*, volume 3097 of *Lecture Notes in Computer Science*, pages 385–399, 2004.
5. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM Press, November 1993.
6. D. Brown. Generic Groups, Collision Resistance, and ECDSA, 2002. Available from <http://eprint.iacr.org/2002/026/>.
7. Coq Development Team. *The Coq Proof Assistant User’s Guide. Version 8.0*, January 2004.
8. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
9. A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Proc. CRYPTO’86*, volume 286 of *Lecture Notes in Computer Science*, pages 186–194. Springer-Verlag, 1986.
10. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
11. V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.



12. Charles Rackoff and Daniel R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 433–444, London, UK, 1992. Springer-Verlag.
13. Bruce Schneier. *Applied Cryptography (Second Edition)*. John Wiley & Sons, 1996.
14. C.-P. Schnorr. Security of Blind Discrete Log Signatures against Interactive Attacks. In S. Qing, T. Okamoto, and J. Zhou, editors, *Proceedings of ICICS'01*, volume 2229 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2001.
15. C.-P. Schnorr and M. Jakobsson. Security of Signed ElGamal Encryption. In T. Okamoto, editor, *Proceedings of ASIACRYPT'00*, volume 1976 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2000.
16. V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *Proceedings of EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1997.
17. N. Smart. The Exact Security of ECIES in the Generic Group Model. In B. Honary, editor, *Cryptography and Coding*, pages 73–84. Springer-Verlag, 2001.
18. J. Stern. Why provable security matters? In E. Biham, editor, *Proceedings of EUROCRYPT'03*, volume 2656 of *Lecture Notes in Computer Science*, pages 449–461. Springer-Verlag, 2003.
19. S. Tarento. Machine-checked security proofs of cryptographic signature schemes. In *Proceedings of ESORICS'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

# Extracting a Normalization Algorithm in Isabelle/HOL

Stefan Berghofer

Technische Universität München,  
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany  
<http://www.in.tum.de/~berghofe/>

**Abstract.** We present a formalization of a constructive proof of weak normalization for the simply-typed  $\lambda$ -calculus in the theorem prover Isabelle/HOL, and show how a program can be extracted from it. Unlike many other proofs of weak normalization based on Tait’s strong computability predicates, which require a logic supporting strong eliminations and can give rise to dependent types in the extracted program, our formalization requires only relatively simple proof principles. Thus, the program obtained from this proof is typable in simply-typed higher-order logic as implemented in Isabelle/HOL, and a proof of its correctness can automatically be derived within the system.

## 1 Introduction

The past years have seen an increasing interest in machine-assisted formalizations of the metatheory of programming languages. As a recent example, consider the POPLMARK Challenge by Pierce et al. [3], which has as a goal the formalization of the basic metatheory of System  $F_{<}$  in a theorem prover, including the definition of an evaluation relation and a proof of type safety. One important aspect of this challenge is *executability*. In particular, it should be possible to generate an executable function computing the normal form of a term from the formalized definition of the evaluation relation, which allows “real” language implementations to be *tested* against the formalization.

A particularly elegant way of obtaining executable code from a formalization is the *extraction* of programs from proofs. In this article, we focus on a proof of *weak normalization* for the simply-typed  $\lambda$ -calculus ( $\lambda_{\rightarrow}$ ) using the proof assistant Isabelle/HOL [16]. Weak normalization means that each well-typed term can be reduced to a term in normal form, whereas *strong normalization* means that each reduction sequence starting from a well-typed term is guaranteed to terminate. Although weak normalization follows from strong normalization as a corollary, proofs of weak normalization are interesting in their own right. If done *constructively*, a proof of weak normalization contains a particular reduction algorithm, which can be uncovered using *program extraction*.

Admittedly, the idea of extracting normalization algorithms from proofs is not completely new. It already dates back to the work of Berger [8], who describes an experiment in extracting a program from a strong normalization proof, which was

formalized using the Minlog theorem prover [7]. Like many other normalization proofs to be found in the literature, Berger’s proof is based on so-called *strong computability predicates* originally introduced by Tait [18]. A term  $t$  is called *strongly computable* if

- $t$  is of base type and  $t$  is *strongly normalizing*, or
- $t$  is of type  $\sigma \Rightarrow \tau$ , and for all *strongly computable* terms  $s$  of type  $\sigma$ , the term  $t s$  of type  $\tau$  is *strongly computable*

Formalizing this notion of strong computability in a theorem prover poses a number of problems. Due to the *negative* occurrence of “*strongly computable*” in the second clause, the above characterization of strong computability is not admissible as an *inductive definition*. A way out of this problem is to define strong computability by recursion on the type of the term  $t$ . Unfortunately, the definition of predicates by recursion on datatypes, which is sometimes called *strong elimination*, exceeds the expressive power of theorem provers such as Minlog. For this reason, Berger does not completely formalize his proof inside the theorem prover, but performs the induction on types “on the meta level”. Strictly speaking, this proof therefore does not yield a single normalization function, but a whole family of functions, which then have to be put together manually. Although Isabelle/HOL is powerful enough to define predicates such as strong computability by recursion, this is still problematic for program extraction. Since predicates in a proof become types in the extracted program, predicates defined by recursion on datatypes give rise to programs using dependent types. Such programs can neither be expressed inside Isabelle/HOL, nor can they easily be translated to functional programming languages such as ML.

As an alternative to Tait-style normalization proofs, Matthes and Joachimski [12] have given a very elegant paper proof of weak normalization for the simply-typed  $\lambda$ -calculus using only relatively simple proof principles. Instead of strong computability predicates, their proof uses a simple inductive characterization of  $\beta$ -normal terms, which turns out to be quite well suited for the purpose of program extraction.

In this article, we present a complete formalization of Matthes’ and Joachimski’s weak normalization proof in the theorem prover Isabelle/HOL. We show how a provably correct program can automatically be extracted from this proof using Isabelle’s framework for program extraction [9, 10].

Similar machine-checked formalizations have been carried out by Altenkirch [1, 2], who proved strong normalization for System F using the LEGO proof assistant [13], as well as Barras and Werner [5, 4] who proved decidability of type checking for the Calculus of Constructions and extracted a type checker from this proof using the Coq [6] theorem prover. A formalization of substantial parts of the metatheory of *Pure Type Systems*, also using the LEGO proof assistant, has been done by Pollack [17]. Coquand [11] has formalized a normalization function for the simply-typed  $\lambda$ -calculus using the proof editor ALF.

The rest of this article is structured as follows: in §2, we give the basic definitions of untyped  $\lambda$ -calculus. On top of this calculus, a system of simple types is introduced in §3. In §4 a definition of normal forms of  $\lambda$ -terms is given, which

serves as a basis for the proof of the normalization theorem presented in §5. An analysis of the program extracted from this proof is contained in §6.

## 2 Basic Definitions

We start by introducing basic concepts such as terms and substitutions. The following definitions are due to Nipkow [14], who used them as a basis for a proof of the Church-Rosser property for  $\beta$ -reduction. They are reproduced here in order to make the exposition self-contained.  $\lambda$ -terms are modelled by the datatype  $dB$  using *de Bruijn indices*, which are encoded by natural numbers.

**datatype**  $dB = Var\ nat \mid App\ dB\ dB \mid Abs\ dB$

We use  $t \circ u$  as an infix notation for  $App\ t\ u$ . When substituting a term for a variable inside an abstraction, the indices of all free variables in the term have to be incremented. This is taken care of by the *lift* function

**consts**  $lift :: dB \Rightarrow nat \Rightarrow dB$

**primrec**

$$\begin{aligned} (Var\ i) \uparrow k &= (if\ i < k\ then\ Var\ i\ else\ Var\ (i + 1)) \\ (s \circ t) \uparrow k &= s \uparrow k \circ t \uparrow k \\ (Abs\ s) \uparrow k &= Abs\ (s \uparrow (k + 1)) \end{aligned}$$

where  $t \uparrow k$  is an infix notation for  $lift\ t\ k$ . Using  $op\ \uparrow$ , we can now define the substitution  $t[s/k]$  of a term  $s$  for a variable  $k$  in a term  $t$  as follows:

**consts**  $subst :: dB \Rightarrow dB \Rightarrow nat \Rightarrow dB$

**primrec**

$$\begin{aligned} (Var\ i)[s/k] &= (if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ s\ else\ Var\ i) \\ (t \circ u)[s/k] &= t[s/k] \circ u[s/k] \\ (Abs\ t)[s/k] &= Abs\ (t[s \uparrow 0 / k+1]) \end{aligned}$$

Since the substitution function will be used to specify  $\beta$ -reduction, it actually does not only substitute the term  $u$  for the variable  $i$ , but also decrements the indices of all other free variables by 1, to compensate for the disappearance of abstractions during  $\beta$ -reduction. The definition of  $\beta$ -reduction, which is denoted by  $s \rightarrow_\beta t$ , is as usual:

**consts**  $beta :: (dB \times dB)\ set$

**inductive**  $beta$

**intros**

$$\begin{aligned} beta: Abs\ s \circ t &\rightarrow_\beta s[t/0] \\ appL: s \rightarrow_\beta t &\Longrightarrow s \circ u \rightarrow_\beta t \circ u \\ appR: s \rightarrow_\beta t &\Longrightarrow u \circ s \rightarrow_\beta u \circ t \\ abs: s \rightarrow_\beta t &\Longrightarrow Abs\ s \rightarrow_\beta Abs\ t \end{aligned}$$

We also use  $\rightarrow_\beta^*$  to denote the transitive closure of  $\rightarrow_\beta$ . The following congruence rules for  $\rightarrow_\beta^*$  are occasionally useful in proofs:

**lemma** *rtrancl-beta-Abs*:  $s \rightarrow_{\beta}^* s' \implies \text{Abs } s \rightarrow_{\beta}^* \text{Abs } s'$

**lemma** *rtrancl-beta-AppL*:  $s \rightarrow_{\beta}^* s' \implies s \circ t \rightarrow_{\beta}^* s' \circ t$

**lemma** *rtrancl-beta-AppR*:  $t \rightarrow_{\beta}^* t' \implies s \circ t \rightarrow_{\beta}^* s \circ t'$

**lemma** *rtrancl-beta-App*:  $s \rightarrow_{\beta}^* s' \implies t \rightarrow_{\beta}^* t' \implies s \circ t \rightarrow_{\beta}^* s' \circ t'$

We will also need the following theorems, asserting that  $\rightarrow_{\beta}$  and  $\rightarrow_{\beta}^*$  are *compatible* with lifting and substitution. The first two of these properties are called *substitutivity* in [12].

**theorem** *subst-preserves-beta*:  $r \rightarrow_{\beta} s \implies (\bigwedge t i. r[t/i] \rightarrow_{\beta} s[t/i])$

**theorem** *subst-preserves-beta'*:  $r \rightarrow_{\beta}^* s \implies r[t/i] \rightarrow_{\beta}^* s[t/i]$

**theorem** *lift-preserves-beta*:  $r \rightarrow_{\beta} s \implies (\bigwedge i. r \uparrow i \rightarrow_{\beta} s \uparrow i)$

**theorem** *lift-preserves-beta'*:  $r \rightarrow_{\beta}^* s \implies r \uparrow i \rightarrow_{\beta}^* s \uparrow i$

**theorem** *subst-preserves-beta2*:  $\bigwedge r s i. r \rightarrow_{\beta} s \implies t[r/i] \rightarrow_{\beta}^* t[s/i]$

**theorem** *subst-preserves-beta2'*:  $r \rightarrow_{\beta}^* s \implies t[r/i] \rightarrow_{\beta}^* t[s/i]$

In addition to the usual *binary* application operator  $s \circ t$ , it is often convenient to also have an *n*-ary application operator  $t \circ^{\circ} ts$  for applying a term  $t$  to a list of terms  $ts$ . To this end, we introduce the abbreviation

**translations**  $t \circ^{\circ} ts \equiv \text{foldl } (op \circ) t ts$

The following equations, describing how lifting and substitution operate on such *n*-ary applications, are easily established by induction on the list  $ts$ :

**lemma** *lift-map*:  $\bigwedge t. (t \circ^{\circ} ts) \uparrow i = t \uparrow i \circ^{\circ} \text{map } (\lambda t. t \uparrow i) ts$

**lemma** *subst-map*:  $\bigwedge t. (t \circ^{\circ} ts)[u/i] = t[u/i] \circ^{\circ} \text{map } (\lambda t. t[u/i]) ts$

### 3 Typed Lambda Terms

In this section, we introduce the type system for simply-typed  $\lambda$ -calculus. The typing judgement usually depends on some environment (or context), assigning types to the free variables occurring in a term. Since variables are encoded using de Bruijn indices, it seems convenient to model environments as functions from natural numbers to types. In order to insert a type  $T$  into an environment  $e$  at a given position  $i$ , we define the function

**constdefs**

*shift* ::  $(nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$

$e\langle i:T \rangle \equiv \lambda j. \text{if } j < i \text{ then } e\ j \text{ else if } j = i \text{ then } T \text{ else } e\ (j - 1)$

where  $e\langle i:T \rangle$  is syntactic sugar for *shift*  $e\ i\ T$ . The types of variables with indices less than  $i$  are left untouched, whereas the types of variables with indices greater than  $i$  are shifted one position up. Instead of working directly with the above definition, we will use the following characteristic theorems for *shift*.

**lemma** *shift-eq*:  $i = j \implies (e\langle i:T \rangle) j = T$

**lemma** *shift-gt*:  $j < i \implies (e\langle i:T \rangle) j = e\ j$

**lemma** *shift-lt*:  $i < j \implies (e\langle i:T \rangle) j = e\ (j - 1)$

**lemma** *shift-commute*:  $e\langle i:U \rangle\langle 0:T \rangle = e\langle 0:T \rangle\langle \text{Suc } i:U \rangle$

Note that the above definition is actually a bit more polymorphic than necessary. We now come to the definition of types. In simply-typed  $\lambda$ -calculus, a type can either be an *atomic* type or a *function* type:

**datatype**  $type = Atom\ nat \mid Fun\ type\ type$

In the sequel, we use  $T \Rightarrow U$  as an infix notation for  $Fun\ T\ U$ . In analogy to the concept of an  $n$ -ary application, it is also useful to have an  $n$ -ary function type operator, which is characterized as follows:

**translations**  $Ts \Rightarrow T \Leftarrow foldr\ Fun\ Ts\ T$

Intuitively,  $Ts \Rightarrow T$  denotes the type of a function whose arguments have the types contained in the list  $Ts$  and whose result type is  $T$ . The definition of the typing judgement  $e \vdash t : T$  is rather straightforward:

**inductive typing**

**intros**

$VarT: e\ x = T \Longrightarrow e \vdash Var\ x : T$

$AbsT: e\langle\theta:T\rangle \vdash t : U \Longrightarrow e \vdash Abs\ t : (T \Rightarrow U)$

$AppT: e \vdash s : T \Rightarrow U \Longrightarrow e \vdash t : T \Longrightarrow e \vdash (s \circ t) : U$

In the typing rule for abstractions, the argument type  $T$  of the function is inserted at position  $\theta$  in the environment  $e$  when checking the type of the body  $t$ . The above typing judgement naturally extends to lists of terms. We write  $e \vdash ts : Ts$  to mean that the terms  $ts$  have types  $Ts$ . Formally, this extension of the typing judgement to lists of terms is defined as follows:

**primrec**

$(e \vdash [] : Ts) = (Ts = [])$

$(e \vdash (t \# ts) : Ts) =$

$(case\ Ts\ of\ [] \Rightarrow False \mid T \# Ts \Rightarrow e \vdash t : T \wedge e \vdash ts : Ts)$

Using the above typing judgement for lists of terms, we can prove the following elimination and introduction rules for types of  $n$ -ary applications:

**lemma** *list-app-typeE*:  $e \vdash t \circ\circ\ ts : T \Longrightarrow$

$(\bigwedge Ts. e \vdash t : Ts \Rightarrow T \Longrightarrow e \vdash ts : Ts \Longrightarrow P) \Longrightarrow P$

**lemma** *list-app-typeI*:  $\bigwedge t\ T\ Ts. e \vdash t : Ts \Rightarrow T \Longrightarrow$

$e \vdash ts : Ts \Longrightarrow e \vdash t \circ\circ\ ts : T$

Before we come to the *subject reduction* theorem, which is the main result of this section, we need several additional results about lifting and substitution. The first two of these lemmas state that lifting preserves the type of a term:

**lemma** *lift-type*:  $e \vdash t : T \Longrightarrow (\bigwedge i\ U. e\langle i:U\rangle \vdash t \uparrow i : T)$

**lemma** *lift-types*:  $\bigwedge Ts. e \vdash ts : Ts \Longrightarrow e\langle i:U\rangle \vdash (map\ (\lambda t. t \uparrow i)\ ts) : Ts$

The first lemma is easily proved by induction on the typing derivation, whereas the second one, which is just a generalization of the first lemma to lists of terms, can be proved by induction on the list  $ts$  using the first result. The other two lemmas state that well-typed substitution preserves the type of terms:

**lemma** *subst-lemma*:  $e \vdash t : T \implies$

$$(\bigwedge e' i U u. e' \vdash u : U \implies e = e'\langle i:U \rangle \implies e' \vdash t[u/i] : T)$$

**lemma** *subst-lemma*:  $\bigwedge Ts. e \vdash u : T \implies$

$$e\langle i:T \rangle \vdash ts : Ts \implies e \vdash (\text{map } (\lambda t. t[u/i]) \text{ } ts) : Ts$$

Again, the proof of the first lemma is by induction on the typing derivation, while the second one is proved by induction on  $ts$ . We are now ready to prove the *subject reduction* property, i.e. that  $\rightarrow_\beta$  preserves the type of a term:

**lemma** *subject-reduction*:  $e \vdash t : T \implies (\bigwedge t'. t \rightarrow_\beta t' \implies e \vdash t' : T)$

The proof is by induction on the typing derivation, where the cases for variables and abstractions are fairly trivial. The case dealing with applications  $s \circ t$  can be proved using elimination on  $s \circ t \rightarrow_\beta t'$  followed by an application of *subst-lemma*. This theorem easily extends to the transitive closure  $\rightarrow_{\beta^*}$  of  $\rightarrow_\beta$ :

**theorem** *subject-reduction'*:  $t \rightarrow_{\beta^*} t' \implies e \vdash t : T \implies e \vdash t' : T$

## 4 Terms in Normal Form

The definition which is central to the proof of weak normalization is, of course, that of a term in *normal form*. Intuitively, a term is in normal form, if it is either a variable applied to a list of terms in normal form, or an abstraction whose body is a term in normal form.

**consts** *NF* :: *dB set*

**inductive** *NF*

**intros**

$$\text{App}N: \text{listall } (\lambda t. t \in NF) \text{ } ts \implies \text{Var } x \circ \circ \text{ } ts \in NF$$

$$\text{Abs}N: t \in NF \implies \text{Abs } t \in NF$$

In the above definition, we write *listall P xs* to denote that a predicate  $P$  holds for all elements in the list  $xs$ . We conclude this section by stating some properties of *NF*, which will be of particular importance for the main proof presented in the next section. As a trivial consequence of the above definition of normal forms, a term consisting of just a variable is in normal form.

**lemma** *Var-NF*:  $\text{Var } n \in NF$

By substituting a variable  $i$  for a variable  $j$  in a normal term  $t$ , we obtain a term which is still in normal form:

**lemma** *subst-Var-NF*:  $t \in NF \implies (\bigwedge i j. t[\text{Var } i/j] \in NF)$

The above lemma is easily proved by induction on the derivation of  $t \in NF$ . If  $t$  is in normal form, the term  $t \circ \text{Var } i$  possesses a normal form, too:

**lemma** *app-Var-NF*:  $t \in NF \implies \exists t'. t \circ \text{Var } i \rightarrow_{\beta^*} t' \wedge t' \in NF$

Again, this result can be proved by induction on the derivation of  $t \in NF$ , using the previous lemma *subst-Var-NF* in the abstraction case. Finally, lifting a normal term  $t$  again yields a normal term:

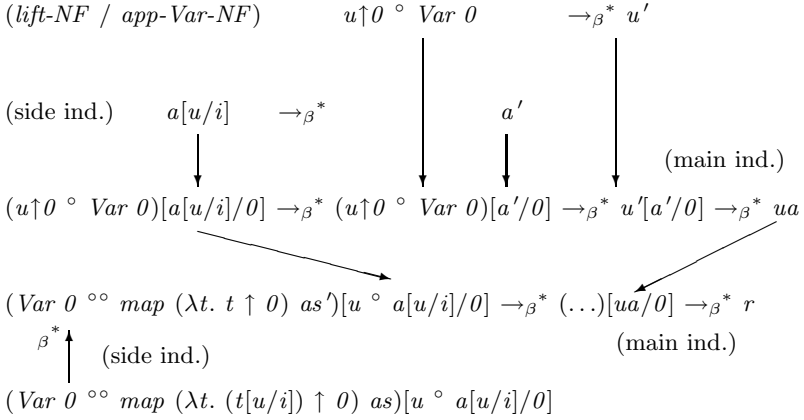


Fig. 1. Overview of proof for application case

**lemma** *lift-NF*:  $t \in NF \implies (\bigwedge i. t \uparrow i \in NF)$

As usual, the proof is by induction on the derivation of  $t \in NF$ .

## 5 Main Theorems

We are now just one step away from our main result, the weak normalization theorem. Actually, the main difficulty is to prove a central lemma, from which weak normalization then follows by a relatively simple argument. The essence of this lemma can be summarized by the slogan “*well-typed substitution preserves the existence of normal forms*”. More formally, if we have a well-typed term  $t$  in normal form containing a variable  $i$  of type  $U$ , then the term  $t[u/i]$  obtained by substituting a term  $u$  in normal form of type  $U$  for the variable  $i$  can be reduced to a normal form  $t'$ . The proof of this statement, which we will now discuss in detail, is by main induction on the type  $U$ , followed by a side induction on the derivation of  $t \in NF$ . An interesting point to note is that the main induction hypothesis is used only in one case of the proof, whereas all the other cases are proved using the side induction hypothesis. In the following, we will give the proof of the lemma in the form of a commented script in the *Isar* proof language due to Markus Wenzel [19, 15].

**lemma** *subst-type-NF*:

$$\begin{aligned} \bigwedge t \in T \ u \ i. \ t \in NF \implies e \langle i : U \rangle \vdash t : T \implies u \in NF \implies e \vdash u : U \implies \\ \exists t'. \ t[u/i] \rightarrow_{\beta^*} t' \wedge t' \in NF \end{aligned}$$

We start the proof by performing induction on the type  $U$ .

**proof** (*induct*  $U$ )

**fix**  $T \ t$

We proceed by side induction on the derivation of  $t \in NF$ :



**assume**  $t \in NF$   
**thus**  $\bigwedge e T' u i. e\langle i:T \rangle \vdash t : T' \implies$   
 $u \in NF \implies e \vdash u : T \implies \exists t'. t[u/i] \rightarrow_{\beta^*} t' \wedge t' \in NF$   
**proof** *induct*  
**fix**  $e T' u i$  **assume**  $uNF: u \in NF$  **and**  $uT: e \vdash u : T$   
 $\{$   
**case**  $(AppN\ ts\ x\ e\ T'\ u\ i)$   
**assume**  $e\langle i:T \rangle \vdash Var\ x \circ\circ\ ts : T'$   
**then obtain**  $Us$   
**where**  $varT: e\langle i:T \rangle \vdash Var\ x : Us \Rightarrow T'$   
**and**  $argsT: e\langle i:T \rangle \Vdash ts : Us$   
**by**  $(rule\ var\ app\ typesE)$

In the application case, we have to distinguish whether or not the variable  $x$  in the head of the term coincides with the variable  $i$  to be substituted.

**from** *nat-eq-dec* **show**  $\exists t'. (Var\ x \circ\circ\ ts)[u/i] \rightarrow_{\beta^*} t' \wedge t' \in NF$   
**proof**  
**assume**  $eq: x = i$   
**show** *?thesis*

In this case, we do a case analysis on the argument list  $ts$ . If the argument list is empty, the claim follows trivially.

**proof**  $(cases\ ts)$   
**case** *Nil*  
**with** *eq* **have**  $(Var\ x \circ\circ\ [])[u/i] \rightarrow_{\beta^*} u$  **by** *simp*  
**with** *Nil* **and** *uNF* **show** *?thesis* **by** *simp rules*  
**next**  
**case**  $(Cons\ a\ as)$   
 $\dots$

The most difficult case of the proof is the one where the argument list is nonempty, i.e. the term on which the substitution is performed has the form  $Var\ x \circ\circ (a \# as)$ . The overall structure of the argument for this case is shown in Figure 1. Substitution and normalization will be performed in several steps. As a first step, we apply substitution and normalization to the tail  $as$  of the argument list. To this end, we prove the following intermediate statement:

**from** *AppN* **and** *Cons* **have** *listall ?SI as*  
**by** *simp (rules dest: listall-conj2)*  
**with** *lift-preserves-beta'* *lift-NF* *uNF* *uT* *argsT'*  
**have**  $\exists as'. \forall j. Var\ j \circ\circ\ map\ (\lambda t. (t[u/i]) \uparrow 0)\ as \rightarrow_{\beta^*}$   
 $Var\ j \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as' \wedge$   
 $Var\ j \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as' \in NF$  **by**  $(rule\ norm-list)$   
**then obtain**  $as'$  **where**  
 $asred: Var\ 0 \circ\circ\ map\ (\lambda t. (t[u/i]) \uparrow 0)\ as \rightarrow_{\beta^*}$   
 $Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as'$   
**and**  $asNF: Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as' \in NF$  **by** *rules*

The desired normal forms are guaranteed to exist due to the side induction hypothesis *listall ?SI as*. Since we later on want to substitute another term for the head variable  $Var\ 0$ , we also have to lift the argument terms, in order to avoid that they are affected by the substitution. In other words, the head variable has to be *new*. The above statement is proved by “reverse induction” on *as*, i.e. elements are appended to the right of the list in the induction step. From the computational point of view, the existentially quantified variable  $as'$  acts as a kind of *accumulator* for the normalized terms. The proof relies on the fact that  $\rightarrow_{\beta^*}$  is a congruence wrt. application and that  $\rightarrow_{\beta^*}$  is compatible with lifting.

By using the side induction hypothesis one more time, we can also apply substitution and normalization to the head  $a$  of the argument list.

**from** *AppN* **and** *Cons* **have** *?SI a* **by** *simp*  
**with** *argT* **and** *uNF* **and** *uT* **have**  $\exists a'. a[u/i] \rightarrow_{\beta^*} a' \wedge a' \in NF$   
**by** *rules*  
**then obtain**  $a'$  **where** *ared*:  $a[u/i] \rightarrow_{\beta^*} a'$  **and** *aNF*:  $a' \in NF$   
**by** *rules*

In order to show that the application of  $u$  to  $a[u/i]$  has a normal form, too, we first note that the term  $u$  applied to a new variable again has a normal form. Since the argument type  $T''$  of  $u$  is smaller than the type  $T = T'' \Rightarrow Ts \Rightarrow T'$  of  $i$ , we can use the main induction hypothesis, together with the previous result and compatibility of  $\rightarrow_{\beta^*}$  with substitution, to show that also  $u \circ a[u/i]$  has a normal form.

**from** *uNF* **have**  $u \uparrow 0 \in NF$  **by** (*rule lift-NF*)  
**hence**  $\exists u'. u \uparrow 0 \circ Var\ 0 \rightarrow_{\beta^*} u' \wedge u' \in NF$  **by** (*rule app-Var-NF*)  
**then obtain**  $u'$  **where** *ured*:  $u \uparrow 0 \circ Var\ 0 \rightarrow_{\beta^*} u'$   
**and** *u'NF*:  $u' \in NF$  **by** *rules*  
**from** *T* **and** *u'NF* **have**  $\exists ua. u'[a'/0] \rightarrow_{\beta^*} ua \wedge ua \in NF$   
**proof** (*rule MII*)  
 ...  
**qed**  
**then obtain**  $ua$  **where** *uared*:  $u'[a'/0] \rightarrow_{\beta^*} ua$   
**and** *uaNF*:  $ua \in NF$  **by** *rules*  
**from** *ared* **have**  $(u \uparrow 0 \circ Var\ 0)[a[u/i]/0] \rightarrow_{\beta^*} (u \uparrow 0 \circ Var\ 0)[a'/0]$   
**by** (*rule subst-preserves-beta2'*)  
**also from** *ured* **have**  $(u \uparrow 0 \circ Var\ 0)[a'/0] \rightarrow_{\beta^*} u'[a'/0]$   
**by** (*rule subst-preserves-beta'*)  
**also note** *uared*  
**finally have**  $(u \uparrow 0 \circ Var\ 0)[a[u/i]/0] \rightarrow_{\beta^*} ua$  .  
**hence** *uared'*:  $u \circ a[u/i] \rightarrow_{\beta^*} ua$  **by** *simp*

Finally, since the type  $Ts \Rightarrow T'$  of  $u \circ a[u/i]$  is also smaller than  $T = T'' \Rightarrow Ts \Rightarrow T'$ , we may again use the main induction hypothesis, together with the previous result, the above intermediate statement concerning the application of substitution and normalization to the argument list *as*, as well as compatibility of  $\rightarrow_{\beta^*}$  with substitution, to show that also  $u \circ a[u/i] \circ \circ map\ (\lambda t. t[u/i])\ as$  has a normal form.

**from**  $T$  **have**

$\exists r. (Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as')[ua/0] \rightarrow_{\beta^*} r \wedge r \in NF$

**proof** (rule *MI2*)

...

**qed**

**then obtain**  $r$

**where**  $rred: (Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as')[ua/0] \rightarrow_{\beta^*} r$

**and**  $rnf: r \in NF$  **by rules**

**from**  $asred$  **have**

$(Var\ 0 \circ\circ\ map\ (\lambda t. (t[u/i]) \uparrow 0)\ as)[u \circ a[u/i]/0] \rightarrow_{\beta^*}$

$(Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as')[u \circ a[u/i]/0]$

**by** (rule *subst-preserves-beta'*)

**also from**  $uared'$  **have**

$(Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as)[u \circ a[u/i]/0] \rightarrow_{\beta^*}$

$(Var\ 0 \circ\circ\ map\ (\lambda t. t \uparrow 0)\ as')[ua/0]$  **by** (rule *subst-preserves-beta2'*)

**also note**  $rred$

**finally have**

$(Var\ 0 \circ\circ\ map\ (\lambda t. (t[u/i]) \uparrow 0)\ as)[u \circ a[u/i]/0] \rightarrow_{\beta^*} r .$

**with**  $rnf$  *Cons eq* **show** *?thesis*

**by** (*simp add: map-compose [symmetric] o-def*) **rules**

**qed**

This concludes the proof for the case where  $x = i$ .

**next**

**assume**  $neq: x \neq i$

...

**qed**

The proof for this case is much easier than the previous one, although it is not completely trivial. As in the previous case, the side induction hypothesis has to be applied to all terms in the argument list  $ts$ . Again, the fact that  $\rightarrow_{\beta^*}$  is a congruence wrt. application is required. This time, no lifting is involved, but the head variable may be decremented as a side effect of substitution (see §2) if  $i < x$ . This concludes the proof for the application case. The abstraction case follows by an easy application of the side induction hypothesis, using the fact that  $\rightarrow_{\beta^*}$  is a congruence wrt. abstraction.

**next**

**case** (*AbsN r e- T'- u- i-*)

**assume**  $absT: e\langle i:T \rangle \vdash Abs\ r : T'$

**then obtain**  $R\ S$  **where**  $e\langle 0:R \rangle \langle Suc\ i:T \rangle \vdash r : S$

**by** (rule *abs-typeE*) *simp*

**moreover have**  $u \uparrow 0 \in NF$  **by** (rule *lift-NF*)

**moreover have**  $e\langle 0:R \rangle \vdash u \uparrow 0 : T$  **by** (rule *lift-type*)

**ultimately have**  $\exists t'. r[u \uparrow 0/Suc\ i] \rightarrow_{\beta^*} t' \wedge t' \in NF$  **by** (rule *AbsN*)

**thus**  $\exists t'. Abs\ r[u/i] \rightarrow_{\beta^*} t' \wedge t' \in NF$

```

    by simp (rules intro: rtrancl-beta-Abs NF.AbsN)
  }
qed
qed

```

Before we can embark on the proof of the main theorem of this section, stating that each well-typed  $\lambda$ -term has a normal form, there is another problem to solve. In the proof of the central lemma *subst-type-NF*, all the required typing information was easy to reconstruct even without inspecting the typing derivation, since the terms supplied as an input to the algorithm underlying the proof

```

theorem type-NF: assumes  $T: e \vdash_R t : T$ 
  shows  $\exists t'. t \rightarrow_{\beta^*} t' \wedge t' \in NF$  using  $T$ 
proof induct
  case VarRT
  show ?case by (rules intro: Var-NF)
next
  case AbsRT
  thus ?case by (rules intro: rtrancl-beta-Abs AbsN)
next
  case (AppRT  $T U e s t$ )
  from AppRT obtain  $s' t'$  where
     $sred: s \rightarrow_{\beta^*} s'$  and  $sNF: s' \in NF$ 
    and  $tred: t \rightarrow_{\beta^*} t'$  and  $tNF: t' \in NF$  by rules
  have  $\exists u. (Var\ 0 \circ t' \uparrow 0)[s'/0] \rightarrow_{\beta^*} u \wedge u \in NF$ 
  proof (rule subst-type-NF)
  have  $t' \uparrow 0 \in NF$  by (rule lift-NF)
  hence  $listall\ (\lambda t. t \in NF)\ [t' \uparrow 0]$  by (rule listall-cons) (rule listall-nil)
  hence  $Var\ 0 \circ\circ [t' \uparrow 0] \in NF$  by (rule AppN)
  thus  $Var\ 0 \circ t' \uparrow 0 \in NF$  by simp
  show  $e\langle 0:T \Rightarrow U \rangle \vdash Var\ 0 \circ t' \uparrow 0 : U$ 
  proof (rule AppT)
  show  $e\langle 0:T \Rightarrow U \rangle \vdash Var\ 0 : T \Rightarrow U$ 
    by (rule VarT) simp
  from tred have  $e \vdash t' : T$ 
    by (rule subject-reduction') (rule rtyping-imp-typing)
  thus  $e\langle 0:T \Rightarrow U \rangle \vdash t' \uparrow 0 : T$ 
    by (rule lift-type)
  qed
  from sred show  $e \vdash s' : T \Rightarrow U$ 
    by (rule subject-reduction') (rule rtyping-imp-typing)
  qed
  then obtain  $u$  where  $ured: s' \circ t' \rightarrow_{\beta^*} u$  and  $unf: u \in NF$  by simp rules
  from sred tred have  $s \circ t \rightarrow_{\beta^*} s' \circ t'$  by (rule rtrancl-beta-App)
  hence  $s \circ t \rightarrow_{\beta^*} u$  using ured by (rule rtrancl-trans)
  with unf show ?case by rules
qed

```

Fig. 2. Proof of main theorem

were already in normal form. This is no longer the case for the main theorem, of course, since its very purpose is the normalization of terms. As these terms do not contain any typing information themselves, this information has to be obtained from the typing derivation. In order to be able to formalize the main theorem, we therefore define a computationally relevant copy  $e \vdash_R t : T$  of the typing judgement  $e \vdash t : T$ , where the subscript  $R$  stands for *Relevant*. The introduction rules characterizing this judgement are the same as for the original one. In order to plug the previous lemma into the proof of the main theorem, we will need the following rule, stating that the computationally relevant typing judgement implies the computationally irrelevant one:

**lemma** *rtyping-imp-typing*:  $e \vdash_R t : T \implies e \vdash t : T$

This rule is easily proved by induction on  $e \vdash_R t : T$ . Note that the other direction would be provable as well, although, from the program extraction point of view, this would not make much sense, since there cannot be a program corresponding to the proof of a computationally relevant statement by induction on a computationally irrelevant statement. Note that instead of a computationally relevant typing judgement, we could as well have used a ‘‘Church-style’’ encoding of  $\lambda$ -terms, where variables in abstractions are decorated with types.

We are now ready to prove weak normalization, which will be done by induction on the typing derivation  $e \vdash_R t : T$ . All cases except for the application case are trivial. In order to normalize a term of the form  $s \circ t$ , we first use the induction hypothesis to compute the normal forms  $s'$  and  $t'$  of  $s$  and  $t$ , respectively. To show that also  $s' \circ t'$  has a normal form, we first note that the application of a new variable to the term  $t'$  is in normal form, so the term obtained by substituting the term  $s'$  for this variable has a normal form according to lemma *subst-type-NF*. By transitivity, we can then put together the reduction sequences found in this way, to yield a normal form of  $s \circ t$ . The whole proof is shown in Figure 2.

## 6 Extracted Programs

We conclude this case study with an analysis of the programs extracted from the proofs presented in the previous section. The programs corresponding to the proof of the main theorem *type-NF*, as well as the central lemma *subst-type-NF*, which performs substitution and normalization, is shown in Figure 3. The outer structure of the function *subst-type-NF* consists of two nested recursion combinators *type-induct-P* and *NFT-rec* corresponding to induction on types and the derivation of normal forms, respectively. The datatype representing the computational content of the inductive definition of normal forms is

$$\begin{aligned} \text{datatype } NFT &= \\ & \quad \text{Dummy} \\ & \quad | \text{AppN } (dB \text{ list}) \text{ nat } (nat \Rightarrow NFT) \\ & \quad | \text{AbsN } dB \text{ NFT} \end{aligned}$$

The universal quantifier over list indices in the definition of the predicate *listall*, which is used in the first introduction rule of *NF* shown in §4, gives rise to the

function type  $\text{nat} \Rightarrow \text{NFT}$  in the list of argument types for the constructor  $\text{AppN}$  in the above datatype definition. Since the constructors  $\text{AppN}$  and  $\text{AbsN}$  both refer to the type  $\text{NFT}$  to be defined recursively, another *Dummy* constructor is required in order to ensure non-emptiness of the datatype. This datatype comes with a so-called *realizability predicate*  $\text{NFR}$ , which establishes a connection between terms in normal form and elements of the above datatype. This predicate is inductively defined by the rules

$$\begin{aligned} \forall i < \text{length } ts. (nf\ i, ts\ !\ i) \in \text{NFR} &\Longrightarrow (\text{AppN } ts\ x\ nf\ s, \text{Var } x\ \circ\circ\ ts) \in \text{NFR} \\ (nf, t) \in \text{NFR} &\Longrightarrow (\text{AbsN } t\ nf, \text{Abs } t) \in \text{NFR} \end{aligned}$$

which are derived in a canonical way from the introduction rules of the predicate  $\text{NF}$ . Intuitively, we can think of  $(nf, t) \in \text{NFR}$  to mean that  $nf$  is a witness of the fact that the term  $t$  is in normal form. Note that  $(nf, t) \in \text{NFR} \Longrightarrow t \in \text{NF}$ , which is easily proved by induction on the derivation of  $(nf, t) \in \text{NFR}$ .

$$\begin{aligned} \text{subst-type-NF} &\equiv \\ \lambda x\ xa\ xb\ xc\ xd\ xe\ H\ Ha. & \\ \text{type-induct-P } xc & \\ (\lambda x\ H2\ H2a\ xa\ xb\ xc\ xd\ xe\ H. & \\ \text{NFT-rec arbitrary} & \\ (\lambda ts\ xa\ xaa\ r\ xb\ xc\ xd\ xe\ H. & \\ \text{var-app-typesE-P } (xb(xe:x))\ xa\ ts & \\ (\lambda Us. \text{case nat-eq-dec } xa\ xe\ \text{of} & \\ \text{Left} \Rightarrow & \\ \text{case } ts\ \text{of } [] \Rightarrow (xd, H) & \\ | a\ \# \text{ list} \Rightarrow & \\ \text{case } Us\ \text{of } [] \Rightarrow \text{arbitrary} & \\ | T''\ \# Ts \Rightarrow & \\ \text{let } (x, y) = & \\ \text{norm-list } (\lambda t. t\ \uparrow\ 0)\ xd\ xb\ xe\ \text{list } Ts & \\ (\lambda t. \text{lift-NF } 0)\ H & \\ (\text{listall-conj2-P-Q } (\lambda i. (xaa\ (Suc\ i), r\ (Suc\ i)))) & \\ (xa, ya) = \text{snd } (xaa\ 0, r\ 0)\ xb\ T''\ xd\ xe\ H; & \\ (xd, yb) = \text{app-Var-NF } 0\ (\text{lift-NF } 0)\ H; & \\ (xa, ya) = H2\ T''\ (Ts \Rightarrow xc)\ xd\ xb\ (Ts \Rightarrow xc)\ xa\ 0\ yb\ ya & \\ \text{in } H2a\ T''\ (Ts \Rightarrow xc)\ (\text{Var } 0\ \circ\circ\ \text{map } (\lambda t. t\ \uparrow\ 0))\ xb\ xc\ xa & \\ 0\ (y\ 0)\ ya & \\ | \text{Right} \Rightarrow & \\ \text{let } (x, y) = & \\ \text{norm-list } (\lambda t. t)\ xd\ xb\ xe\ ts\ Us\ (\lambda x\ H. H)\ H & \\ (\text{listall-conj2-P-Q } (\lambda z. (xaa\ z, r\ z))) & \\ \text{in case nat-le-dec } xe\ xa\ \text{of} & \\ \text{Left} \Rightarrow (\text{Var } (xa - Suc\ 0)\ \circ\circ\ x, y\ (xa - Suc\ 0)) & \\ | \text{Right} \Rightarrow (\text{Var } xa\ \circ\circ\ x, y\ xa))) & \\ (\lambda t\ x\ r\ xa\ xb\ xc\ xd\ H. & \\ \text{abs-typeE-P } xb & \\ (\lambda U\ V. \text{let } (x, y) = r\ (xa\ \langle 0:U \rangle)\ V\ (xc\ \uparrow\ 0)\ (Suc\ xd)\ (\text{lift-NF } 0)\ H) & \\ \text{in } (\text{Abs } x, \text{AbsN } x\ y))) & \\ H\ xb\ xc\ xd\ xe) & \\ x\ xa\ xd\ xe\ xb\ H\ Ha & \end{aligned}$$

$$\begin{aligned} \text{type-NF} &\equiv \\ \lambda H. \text{rtypingT-rec } (\lambda e\ x\ T. (\text{Var } x, \text{Var-NF } x)) & \\ (\lambda e\ T\ t\ U\ x\ r. \text{let } (x, y) = r\ \text{in } (\text{Abs } x, \text{AbsN } x\ y)) & \\ (\lambda e\ s\ T\ U\ t\ x\ xa\ r\ ra. & \\ \text{let } (x, y) = r; (xa, ya) = ra & \\ \text{in subst-type-NF } (\text{Var } 0\ \circ\circ\ xa\ \uparrow\ 0)\ e\ 0\ (T \Rightarrow U)\ U\ x & \\ (\text{AppN } [xa\ \uparrow\ 0]\ 0\ (\text{listall-cons-P } (\text{lift-NF } 0)\ ya)\ \text{listall-nil-P}))\ y) & \\ H & \end{aligned}$$

**Fig. 3.** Programs extracted from main theorem and *subst-type-NF*

The recursion combinator *NFT-rec* occurring in the program shown in Figure 3 has three functions as arguments, corresponding to the constructors of the above datatype. The first function corresponds to the *Dummy* constructor. Since this constructor may never occur, we may supply an *arbitrary* function as an argument, which, when generating executable code, may be implemented by a function raising an exception on invocation. The second function corresponds to the application case of the proof. It contains a case distinction (using function *nat-eq-dec*) on whether the variable *xa* coincides with the variable *xe*<sup>1</sup>. The first case (labelled with *Left*), which is the more difficult one, contains another case distinction on the structure of the argument list. The second case (labelled with *Right*) is the easier one. It contains another case distinction (using function *nat-le-dec*) on whether *xe* < *xa*. In the “*Left*” case, the variable in the head of the term is decremented, whereas it remains unchanged in the “*Right*” case. In both the case for *xa* = *xe* and *xa* ≠ *xe* the function *norm-list* is used to apply the normalization function to a list of terms. The last lines of the program shown in Figure 3 contain the relatively trivial program corresponding to the proof for the abstraction case. The correctness theorem corresponding to the program *subst-type-NF* is

$$\begin{aligned} \bigwedge x. (x, t) \in NFR &\implies \\ e \langle i : U \rangle \vdash t : T &\implies \\ (\bigwedge xa. (xa, u) \in NFR &\implies \\ e \vdash u : U &\implies \\ t[u/i] \rightarrow_{\beta^*} fst (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa) \wedge \\ (snd (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa), \\ fst (subst\text{-}type\text{-}NF\ t\ e\ i\ U\ T\ u\ x\ xa)) \\ \in NFR) \end{aligned}$$

The function *type-NF* is defined by recursion on the datatype

$$\begin{aligned} \text{datatype } rtypingT = & \\ & VarRT (nat \Rightarrow type) nat type \\ & | AbsRT (nat \Rightarrow type) type dB type rtypingT \\ & | AppRT (nat \Rightarrow type) dB type type dB rtypingT rtypingT \end{aligned}$$

representing the computational content of the typing derivation. The correctness statement for the main function *type-NF* is

$$\begin{aligned} \bigwedge ty. (ty, e, t, T) \in rtypingR &\implies \\ t \rightarrow_{\beta^*} fst (type\text{-}NF\ ty) \wedge (snd (type\text{-}NF\ ty), fst (type\text{-}NF\ ty)) &\in NFR \end{aligned}$$

where *rtypingR* is the realizability predicate corresponding to the computationally relevant version of the typing judgement. In analogy to *NFR*, we can think of  $(ty, e, t, T) \in rtypingR$  to mean that *ty* is a witness of the fact that *t* has type *T* in environment *e*. The reduction relation  $\rightarrow_{\beta^*}$  has been chosen to have no computational content, since we are only interested in the normal form of a term, and not the actual *reduction sequence* leading to it.

<sup>1</sup> Due to the numerous transformations, which are performed on the proof before extraction, variable names in the extracted programs may often differ from those in the original Isar proof document.

Compared to the programs which are extracted “manually” by Matthes and Joachimski [12–§2.3], the automatically extracted programs presented in this section may look a bit more complicated and harder to read. This is due to the fact that, although (according to the proof) the main recursion in the program given by Matthes and Joachimski should be over types, no type information is mentioned in the extracted program at all. Moreover, the program looks as if it were defined by recursion over terms, whereas, strictly speaking, it should involve recursion over the datatype *NFT* representing the computational content of the inductive characterization of normal forms.

In order to test the performance of the extracted normalization function, we compile it to ML and use it to compute multiplication on *Church numerals*:

$$\begin{aligned} 2 &=_{def} (\lambda f. \lambda x. f (f x)) \\ \star &=_{def} (\lambda m. \lambda n. \lambda f. m (n f)) \end{aligned}$$

The time required for computing the normal form of  $x \star 2$  on a Pentium III with 1 GHz is as follows:

$x$	2	4	8	16	32	64	128	256	512
runtime [s]	0.002	0.016	0.012	0.036	0.096	0.430	2.615	27.786	364.193

## 7 Conclusion

In this article, we have presented a fully-formalized, readable proof of a fairly complex result for simply-typed  $\lambda$ -calculus. All the definitions, theorems and proofs shown in this article have been directly generated from the Isabelle proof scripts. The whole formalization is quite compact. It consists of three main modules, one containing basic results about untyped  $\lambda$ -calculus, in particular the definition and properties of  $\beta$ -reduction, another one containing results about the type system, and one containing properties of normal forms, as well as proofs of the main theorems. Each of these modules takes up about 400 lines of Isabelle code. Using Isabelle’s built-in code generator, it is possible to generate executable ML code from the extracted normalization function presented in §6, which performs reasonably well on medium-size  $\lambda$ -terms.

## References

1. T. Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.
2. T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference (TLCA '93)*, volume 664 of *LNCS*, pages 13–28. Springer-Verlag, 1993.
3. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics: TPHOLS 2005*, *LNCS*. Springer-Verlag, 2005.



4. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.
5. B. Barras and B. Werner. Coq in Coq. To appear in Journal of Automated Reasoning.
6. B. Barras et al. The Coq proof assistant reference manual – version 7.2. Technical Report 0255, INRIA, February 2002.
7. H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, pages 41–71. Kluwer Academic Publishers, Dordrecht, 1998.
8. U. Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference (TLCA '93)*, pages 91–106, Berlin, Germany, 1993. Springer-Verlag.
9. S. Berghofer. Program Extraction in simply-typed Higher Order Logic. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*. Springer-Verlag, 2003.
10. S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Institut für Informatik, TU München, 2003.
11. C. Coquand. From semantics to rules: A machine assisted analysis. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*, volume 832 of *LNCS*, pages 91–105. Springer-Verlag, 1993.
12. F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed  $\lambda$ -calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003.
13. Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
14. T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.
15. T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer-Verlag, 2003.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
17. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
18. W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.
19. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002. <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.

# A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis

Yves Bertot<sup>1</sup>, Benjamin Grégoire<sup>2</sup>, and Xavier Leroy<sup>3</sup>

<sup>1</sup> Projet MARELLE, INRIA Sophia-Antipolis, France  
Yves.Bertot@sophia.inria.fr

<sup>2</sup> Projet EVEREST, INRIA Sophia-Antipolis, France  
Benjamin.Gregoire@sophia.inria.fr

<sup>3</sup> Projet CRISTAL, INRIA Rocquencourt, France  
Xavier.Leroy@inria.fr

**Abstract.** This paper reports on the correctness proof of compiler optimizations based on data-flow analysis. We formulate the optimizations and analyses as instances of a general framework for data-flow analyses and transformations, and prove that the optimizations preserve the behavior of the compiled programs. This development is a part of a larger effort of certifying an optimizing compiler by proving semantic equivalence between source and compiled code.

## 1 Introduction

Can you trust your compiler? It is generally taken for granted that compilers do not introduce bugs in the programs they transform from source code to executable code: users expect that the generated executable code behaves as prescribed by the semantics of the source code. However, compilers are complex programs that perform delicate transformations over complicated data structures; this is doubly true for optimizing compilers that exploit results of static analyses to eliminate inefficiencies in the code. Thus, bugs in compilers can (and do) happen, causing wrong executable programs to be generated from correct source programs.

For low-assurance software that is validated only by testing, compiler bugs are not a major problem: what is tested is the executable code generated by the compiler, therefore compiler bugs should be detected along with program bugs if the testing is sufficiently exhaustive. This is no longer true for critical, high-assurance software that must be validated using formal methods (program proof, model checking, etc): here, what is certified using formal methods is the source code, and compiler bugs can invalidate the guarantees obtained by this certification. The critical software industry is aware of this issue and uses a variety of techniques to alleviate it, such as compiling with optimizations turned off and conducting manual code reviews of the generated assembly code. These techniques do not fully address the issue, and are costly in terms of development time and program performance.

It is therefore high time to apply formal methods to the compiler itself in order to gain assurance that it preserves the behaviour of the source programs.

Many different approaches have been proposed and investigated, including proof-carrying code [6], translation validation [7], credible compilation [8], and type-preserving compilers [5]. In the ConCert project (Compiler Certification), we investigate the feasibility of performing program proof over the compiler itself: using the Coq proof assistant, we write a moderately-optimizing compiler from a C-like imperative language to PowerPC assembly code and we try to prove a semantic preservation theorem of the form

For all correct source programs  $S$ , if the compiler terminates without errors and produces executable code  $C$ , then  $C$  has the same behaviour (up to observational equivalence) as  $S$ .

An original aspect of ConCert is that most of the compiler is written directly in the Coq specification language, in a purely functional style. The executable compiler is obtained by automatic extraction of Caml code from this specification.

In this paper, we report on the correctness proof of one part of the compiler: optimizations based on dataflow analyses performed over an intermediate code representation called RTL (Register Transfer Language). Section 2 gives an overview of the structure of the compiler. Section 3 defines the RTL language and its semantics. Section 4 develops a generic framework for dataflow analyses and transformations. Section 5 instantiates this framework for two particular optimizations: constant propagation and common subexpression elimination. We discuss related work in section 6 and conclude in section 7.

The work outlined in this paper integrates in a more ambitious project that is still in progress and is therefore subject to further modifications. A snapshot of this work can be found on the project site<sup>1</sup>.

## 2 General Scheme of the Compiler

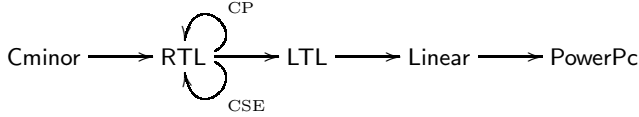
The compiler that we consider is, classically, composed of a sequence of code transformations. Some transformations are non-optimizing translations from one language to another, lower-level language. Others are optimizations that rewrite the code to eliminate inefficiencies, while staying within the same intermediate language. Following common practice, the transformations are logically grouped into three phases:

1. *The front-end*: non-optimizing translations from the source language to the RTL intermediate language (three-address code operating on pseudo-registers and with control represented by a control-flow graph).
2. *The optimizer*: dataflow analyses and optimizations performed on the RTL form.
3. *The back-end*: translations from RTL to PowerPC assembly code.

In the following sections, we concentrate on phase 2: RTL and optimizations. To provide more context, we give now a quick overview of the front-end and back-end (phases 1 and 3).

---

<sup>1</sup> <http://www-sop.inria.fr/lemme/concert/>



**Fig. 1.** Scheme of the compiler

The source language of the front-end is `Cminor`: a simple, C-like imperative language featuring expressions, statements and procedures. Control is expressed using structured constructs (conditionals, loops, blocks). We do not expect programmers to write `Cminor` code directly; rather, `Cminor` is an appropriate target language for translating higher-level languages such as the `goto`-less fragment of C. The translation from `Cminor` to RTL consists in decomposing expressions into machine-level instructions, assigning (temporary) pseudo-registers to `Cminor` local variables and intermediate results of expressions, and building the control-flow graph corresponding to `Cminor` structured control constructs.

The back-end transformations start with register allocation, which maps pseudo-registers (an infinity of which is available in RTL code) to actual hardware registers (in finite number) and slots in the stack-allocated activation record of the function. We use Appel and George’s register allocator based on coloring of an interference graph. The result of this pass is LTL (Location Transfer Language) code, similar in structure to RTL but using hardware registers and stack slots instead of pseudo-registers. Next, we linearize the control-flow graph, introducing explicit `goto` instructions where necessary (Linear code). Finally, PowerPC assembly code is emitted from the Linear code.

## 3 RTL and Its Semantics

### 3.1 Syntax

The structure of the RTL language is outlined in figure 2. A full program is composed of a set of function definitions  $P_F$  represented by a vector of pairs binding a function name and a function declaration, one of these function names should be declared as the main function  $P_{\text{main}}$ . It also contains a set of global variable declarations represented by a vector of pairs binding global variable names and their corresponding memory size.

Each function declaration indicates various parameters for this function, like the arguments (`args`), or the size of the stack memory that is required for local usage (`space`). Important ingredients are the function code (`C`) and the entry point (`ep`). The record also contains “certificates”: the field (`in`) contains a proof that the entry point is in the code, while the field (`WC`) contains a proof that the code is a well-connected graph.

The code itself is given as a partial function from a type  $s$  of program locations to a type  $I$  of instructions. To encode this partial function, we view it as the pair of a total function and a domain predicate. Each instruction combines an instruction description and a vector of potential successors.

global variable	$x$
program point	$c \mid s$
comparison	$\text{cmp} ::= = \mid \neq \mid < \mid \leq \mid > \mid \geq$
condition	$\text{cnd} ::= \text{Ccomp cmp} \mid \text{Ccompu cmp} \mid \text{Ccomp\_imm cmp } i$ $\mid \text{Ccompu\_imm cmp } i \mid \text{Ccompf cmp} \mid \text{Cnotcompf cmp}$ $\mid \text{Ccompptr cmp} \mid \text{Cmaskzero } i \mid \text{Cmasknotzero } i$
operation	$\text{op} ::= \text{Omove} \mid \text{Oconstint } i \mid \text{Oconstfloat } f$ $\mid \text{Oaddrglobal } x \mid \text{Oaddrstack } i$ $\mid \textit{arithmetic operations}$ $\mid \textit{pointer arithmetic}$ $\mid \textit{specific operations}$ $\mid \text{Ocomp cnd}$
memory chunk	$\text{mc} ::= \text{int8} \mid \text{uint8} \mid \text{int16} \mid \text{uint16} \mid \text{int32} \mid \text{float32} \mid \text{float64} \mid \text{addr}$
addressing mode	$\text{am} ::= \text{Aindexed} \mid \text{Aindexed\_imm } i \mid \text{Abased } x \ i \mid \text{Abased\_imm } x \ i$ $\mid \text{Ainstack } i$
instruction description	$d ::= \text{Inop} \mid \text{lop op } \bar{r} \ r$ $\mid \text{lload mc am } \bar{r} \ r \mid \text{lstore mc am } \bar{r} \ r$ $\mid \text{lcall } r \ \bar{r} \ r \mid \text{lcall\_imm } x \ \bar{r} \ r$ $\mid \text{lcond cnd } \bar{r} \mid \text{lswitch } r \ \bar{i}$ $\mid \text{lreturn } r$
instruction code	$I ::= d * \bar{s}$ $C ::= s \mapsto I$
Function declaration	$F ::= \{ \text{space} : \text{size}; \quad \text{args} : \bar{r};$ $\quad \text{C} : C; \quad \text{ep} : c;$ $\quad \text{in} : \text{ep} \in C;$ $\quad \text{WC} : \forall c \ s, \ s \in \text{Succ}(c) \Rightarrow s \in C \}$
Program declaration	$P ::= \{P_F : \overline{(x, F)}; \ P_{\text{main}} : x; \ P_{\text{vars}} : \overline{(x, \text{size})}\}$

Fig. 2. RTL syntax

Instruction descriptions really specify what each instruction does: memory access, call to other functions, conditional and switch, or return instructions. A “no-operation” instruction is also provided: this instruction is useful to replace discarded instructions without changing the graph shape during optimizations.

Instruction descriptions do not contain the successor of instructions, this information is given at the upper level  $I$  by the list of successors. This organization makes it easier to describe some analyses on the code in an abstract way. For most instruction descriptions, exactly one successor is required, except for the `lreturn` instruction, which takes no successor, the `lcond` instruction, which takes two successors and for the `lswitch` instruction which takes one successor for each defined label ( $\bar{i}$ ) and one for the default branch.

The syntactic categories of comparison (`cmp`), condition (`cnd`) and operation (`op`) are used to give a precise meaning to the various instructions. Note that

the language makes a distinction between several kinds of numbers: integers, unsigned integers, floating point numbers, and pointers. This distinction appears in the existence of several kinds of comparison and arithmetic operations. Operations of each kind are distinguished by the suffix `u`, `f`, or `ptr`, or no suffix for plain integers. The arithmetic on pointers makes it possible to compare two pointers, to subtract them (this return a integer), and to add an integer to a pointer. The way pointers are handled in the language is also related to the way memory is handled in the store and load instructions.

Memory access instructions (`lload` and `lstore`) take a memory chunk (`mc`) parameter that indicates the type of data that is read or written in memory. It also indicates how data is to be completed: when reading an 8-bit integer into a 32-bit register, it is necessary to choose the value of the remaining 24 bits. If the integer is considered unsigned, then the extra bits are all zeros, but if the data is signed, then the most significant bit actually is a sign bit and one may use complement to 2 representation. This way of handling numbers is common to most processors with 32-bit integer registers and 64-bit floating point registers.

A second parameter to load and store operations is the addressing mode. This addressing mode always takes two arguments: an address and an offset. The addressing modes given here are especially meaningful for the Power-PC micro-processor.

The language contains two instructions for function calls, `lcall_imm` represents the usual function call where the function name is known at compile-time, whereas `lcall` represents a function call where the function is given as a location in a register<sup>2</sup>. This capability is especially useful if we want to use Cminor or RTL as an intermediate language for the compilation of functional programming languages, where the possibility to manipulate and call function pointers is instrumental.

### 3.2 Values and Memory Model

To describe the way memory is handled in our language, we use a model that abstracts away from some of the true details of implementation. This abstraction has two advantages. First, it preserves some freedom of choice in the design of the compiler (especially with respect to stack operations); second it makes the correctness proofs easier. This abstraction also has drawbacks. The main drawback is that some programs that could be compiled without compiling errors and run without run-time error have no meaning according to our semantics.

The first point of view that one may have on memory access is that the memory may be viewed as a gigantic array of cells that all have the same size. Accessing a memory location relies on knowing this location's address, which can be manipulated just as any other integer. The stack also appears in the memory, and the address of objects on the stack can also be observed by some programs. If this memory model was used, then this would leave no freedom to the compiler designer: the semantics would prescribe detailed information such

---

<sup>2</sup> As in C, when a pointer to a function is taken as argument.

as the amount of memory that is allocated on the stack for a function call, because this information can be inferred by a cleverly written C program.

In practice, this point of view is too detailed, and the actual value of addresses may change from one run of the same program to another, for instance because the operating system interferes with the way memory is used by each program. The compiler should also be given some freedom in the way compiled programs are to use the stack, for instance in choosing which data is stored in the micro-processor's registers and which data is stored in the stack.

The abstract memory model that we have designed relies on the following feature: memory is always allocated as “blocks” and every address actually is the pair of a block address and an offset within this block. The fact that we use an offset makes it possible to have limited notions of pointer arithmetic: two addresses within the same block can be compared, their distance can be computed (as an integer), and so on. On the other hand, it makes no sense to compare two addresses coming from two different blocks, this leaves some freedom to the compiler to choose how much memory is used on the stack between two allocated chunks of memory. For extensions of the language, this also makes it possible to have memory management systems that store data around the blocks they allocate.

From the semantic point of view, this approach is enforced by the following features. First, our RTL language has a specific type of memory address for which direct conversion to integer is not possible, but we can add an integer to an address (we enforce the same restriction in our formal semantics of *Cminor*). Second, any memory access operation that refers to a block of a given size with an offset that is larger than this size has no semantics.

All the syntactic structures used for values and memory are described in Figure 3.

Our memory model relies on block allocation. Some of the blocks are long-lived and stored in the main memory. Other blocks are short lived and correspond to local memory used in C functions (the memory allocated on a stack at each function call is often referred to as a *frame*). These blocks are “allocated” when the function is called and “deallocated” when the function returns the control to its caller. It is relevant to make sure that different blocks used for different call instances of functions are really understood as different blocks. To achieve this level of abstraction, we consider that there is an infinity of blocks with different status. Memory is then considered as a mapping from an infinite abstract type of *head pointers* to a collections of block.

Memory modifications correspond to changing the values associated to these head pointers. In the beginning, most head pointers are mapped to a **NotYetAllocated** block. When a function is called and a block of memory is allocated for this function, a head pointer is updated and becomes mapped to a vector of elementary memory cells (corresponding to bytes). When the block is released, the head pointer is updated again and becomes mapped to a **Freed** block. Our semantics description of the language gives no meaning to load and store oper-

head pointer	$p$
memory address	$a ::= (p, n)$
value	$v ::= n \mid f \mid a \mid \perp$
basic block	$b ::= \perp \mid c \mid a$
block	$B ::= C \mid \bar{b} \mid \text{Freed} \mid \text{NotYetAllocated}$
memory	$m ::= p \mapsto B$
load	$\text{load} : m \rightarrow \text{mc} \rightarrow a \rightarrow v$
	$\text{load}_F : m \rightarrow a \rightarrow F$
store	$\text{store} : m \rightarrow \text{mc} \rightarrow a \rightarrow v \rightarrow m_\perp$
	$\text{store}_F : m \rightarrow \text{mc} \rightarrow a \rightarrow v \rightarrow m_\perp$

**Fig. 3.** Value and memory

ation in `NotYetAllocated` and `Freed` blocks. In this way, we are able to account for the limited life-span of some memory blocks.

To ensure the clean separation between memory addresses and integer values, we only specify that a memory address requires four bytes for storage<sup>3</sup>. We allow writing bytes into a location where an address was previously stored, but this has the side effect of storing an “unknown” value in all the other bytes of the address. We use the symbol  $\perp$  to denote these unknown values.

Blocks may also contain some code. This feature makes it possible to pass function pointers around and to implement function calls where the function is given by pointer. The level of abstraction is simple to state here: there is no way to know what size a piece of code is taking and only the head pointer of a block containing code may be used in a function call. Of course, reading an integer or an address from a block that contains code simply returns the unknown value  $\perp$ .

### 3.3 Semantics

Figure 4 outlines the semantics of RTL. Some parts are given by inductively defined relations, while other parts are given by functions. Important data for these relations and functions is collected in a structure named a *state*. The component named  $S_r$  is a function that gives the value of all registers. The component  $S_{\text{stk}}$  indicates the current position of the stack top; this position can be used as a head-pointer for access in the memory allocated for the current function call instance. The component  $S_{\text{mem}}$  contains the description of the whole memory state. The component  $S_{\text{id}}$  indicates the values associated to all the global variables (identifiers).

The semantics of individual instructions is described as an inductive relation using a judgment of the form  $(S, I) \mapsto (S', s)$ , where  $S$  is the input state,  $I$  is the instruction to execute,  $S'$  is the result state, and  $s$  is the location of the next instruction. In Figure 4, we include a few of the inference rules for this judgment. Each rule contains a few consistency checks: for instance, instructions expecting a single successor should not receive several potential successors, the

<sup>3</sup> This will allow to use integers for encoding pointers.



state	$S ::= \{S_r : r \rightarrow v; S_{\text{stk}} : p; S_{\text{mem}} : m; S_{\text{id}} : x \rightarrow v\}$
condition	$\text{eval}_{\text{cnd}} : S \rightarrow \text{cnd} \rightarrow \bar{r} \rightarrow v$
operation	$\text{eval}_{\text{op}} : S \rightarrow \text{op} \rightarrow \bar{r} \rightarrow v$
addressing mode	$\text{eval}_{\text{am}} : S \rightarrow \text{am} \rightarrow \bar{r} \rightarrow a_{\perp}$

---

Evaluation of one instruction (selected rules)

---

$$\frac{}{(S, (\text{Inop}, [s])) \mapsto (S, s)} \quad \frac{\text{eval}_{\text{op}}(S, \text{op}, \bar{r}) = v \quad \text{CheckType}(r_d, v)}{(S, (\text{lop op } \bar{r} r_d, [s])) \mapsto (S[r_d := v], s)}$$

$$\frac{S_{\text{id}}(x) = a \quad \text{load}_F(S_{\text{mem}}, a) = f \quad \text{InitFun}(f, S, \bar{r}) = S_1 \quad (S_1, f_{\text{ep}}) \xrightarrow{f} (S_2, v) \quad \text{Return}(S, S_2) = S_3 \quad \text{CheckType}(r_d, v)}{(S, (\text{lcall\_imm } x \bar{r} r_d, [s])) \mapsto (S_3[r_d := v], s)}$$

---

Evaluation of several instructions

---

$$\frac{f_C(c_1) = I \quad (S_1, I) \mapsto (S_2, c_2) \quad (S_2, c_2) \xrightarrow{f} (S_3, v)}{(S_1, c_1) \xrightarrow{f} (S_3, v)} \quad \frac{f_C(c) = (\text{lreturn } r, \emptyset)}{(S, c) \xrightarrow{f} (S, S.(r))}$$

---

Evaluation of a program

---

$$\frac{\text{InitProg}(p) = S \quad p_F(p_{\text{main}}) = f \quad \text{InitFun}(f, S, \emptyset) = S_1 \quad (S_1, f_{\text{ep}}) \xrightarrow{f} (S_2, v) \quad \text{Return}(S, S_2) = S_3}{p \xrightarrow{P} (S_3, v)}$$

**Fig. 4.** Semantics of RTL

registers providing the input for an arithmetic operation should contain values of the right type, and the register for the output value should also have the right type.

To describe the execution of several instructions, we use a judgment of the form  $(S, c) \xrightarrow{f} (S', v)$ , where  $f$  is a function,  $c$  is the location of an instruction in the code graph of  $f$ ,  $S'$  is the result state when the function execution terminates and  $v$  is the return value. So while the judgment  $(S, I) \mapsto (S', s)$  is given in a style close to *small step* operational semantics, the judgment  $(S, c) \xrightarrow{f} (S', v)$  describes a complete execution as we are accustomed to see in *big step* operational semantics or natural semantics. The two styles actually intertwine in the rule that describes the execution of the `lcall_imm` instruction: to describe the behavior of the single `lcall_imm` instruction, this rule requires that the code of the callee is executed completely to its end.

The rule for the `lcall_imm` instruction first states that the function code must be fetched from memory, then that some initialization must be performed on the state. This initialization consists of allocating a new memory block for the data that is local to the function and moving the  $S_{\text{stk}}$  field. Then, ver-

ifications must be performed: the registers that are supposed to contain the inputs to the called function must contain data of the right type. The registers that are used internally to receive the input arguments must be initialized. Then the function code is simply executed from its entry point, until it reaches a `lreturn` statement, which marks the end of the function execution. An auxiliary function, called `Return` then modifies the state again, restoring the  $S_{\text{stk}}$  field to its initial value and freeing the memory block that had been allocated.

Evaluating a program is straightforward. Some initialization is required for the global variables and the main function of the program is executed to its end.

When proving the correctness of optimizations, we intuitively want to prove that an optimized function performs the same computations as the initial function. A naive approach would be to express that the resulting state after executing the optimized function is the same as the resulting state after executing the initial function. However, this approach is too naive, because optimization forces the state to be different: since the function code is part of the program's memory, the optimized program necessarily executes from an initial state that is different and terminates in a state that is different. Our correctness statement must therefore abstract away from the details concerning the code stored in the memory.

## 4 A Generic Proof for the Optimizations

In the sequel, we consider two optimization phases from RTL to RTL, the first one is constant propagation (CP) and the second is common subexpression elimination (CSE). Both are done procedure by procedure and follow the same scheme.

The first step is a data-flow analysis of the procedures. The result of this analysis is a map that associates to each program point the information that is “valid” before its execution. By “valid information” we mean that any execution would reach this program point in a state that is compatible with the value predicted by the analysis.

The second step is the transformation of the code. This transformation is done instruction by instruction and only requires the result of the analysis for one instruction to know how to transform it. Naturally, the power of the transformation depends directly on the analysis.

We wrote a generic program and its proof to automatically build the analyses, the transformations, and their proofs.

### 4.1 Analysis Specification

A data-flow analysis is given by a set  $D$  (the domain of the analysis) representing abstractions of states and an analysis function  $A_F$  that takes as input a function declaration and returns a map from program points to  $D$ . These two parameters represent the computation part of the analysis. For CP and CSE the domain is

a mapping from registers to abstract values, e.g. a concrete value or “unknown” for CP, or a symbolic expression for CSE.

An analysis is correct when it satisfies some extra properties. In particular, we need an analysis property  $A_{\mathcal{P}}$  that describes when a concrete state in  $S$  is compatible with an abstract state in  $D$ . The result of the analysis should have the property that every reachable instruction can only be reached in a state that is compatible with the abstract state that is predicted by the analysis.

The first part of the correctness statement expresses that every state should be compatible with the abstract point that is computed for the entry point; this statement is called  $A_{\text{entry}}$ .

The second part of the correctness statement expresses that every computation state should map a state compatible with the abstract state for an instruction to states that are compatible with the abstract states of all the successor; this statement is called  $A_{\text{correct}}$ .

All the components of a valid analysis are gathered in a module type for Coq’s module system.

#### Module Type ANALYSIS.

Parameter  $D$  : Set.

Parameter  $A_F : F \rightarrow \text{Map.T } D$ .

Parameter  $A_{\mathcal{P}} : D \rightarrow S \rightarrow \text{Prop}$ .

Parameter  $A_{\text{entry}} : \forall f S, A_{\mathcal{P}}(A_F(f_{\text{ep}}), S)$ .

Parameter  $A_{\text{correct}} : \forall f c_1 c_2 S_1 S_2,$

$$c_1 \in f_C \Rightarrow (S_1, f_C(c_1)) \mapsto (S_2, c_2) \Rightarrow A_{\mathcal{P}}(A_F(c_1), S_1) \Rightarrow A_{\mathcal{P}}(A_F(c_2), S_2).$$

End.

Correct analyses can be obtained in a variety of ways, but we rely on a generic implementation of Kildall’s algorithm [2]: the standard solver for data-flow equations using a work list. To compute an analysis, Kildall’s algorithm observes each instruction separately. Given a program point  $c$ , it computes the value of the analysis for the instruction  $c$  using a function  $A_i$ . Then it update the abstract state of every successor of  $c$  with an upper bound of itself and the value of  $A_i$  at  $c$ . The function  $A_i$  is the analysis of an instruction, it takes what is known before the instruction as argument and returns what is known after it. The termination of this algorithm is ensured because the code graph for a function is always finite and the domain of the analysis is a well-founded semi-lattice : a set  $D$  with a strict partial order  $>_D$ , a decidable equality, a function that returns an upper bound of every pair of elements, and the extra property that the strict partial order is well-founded.

We don’t give more details about Kildall’s algorithm, for lack of place, but in our formal development, it is defined using Coq’s module system as a functor (a parameterized module) taking as arguments the well-founded semi-lattice, the analysis function, and a few correctness properties. This functor is in turn used to define a generic functor that automatically builds new analyses. This functor takes as argument modules that satisfy the following signature:

Module Type ANALYSIS\_ENTRIES.

Declare Module  $D$  : SEMILATTICE.

Parameter  $A_i : F \rightarrow \mathbf{c} \rightarrow D.T \rightarrow D.T$ .

Parameter  $A_{\mathcal{P}} : D.T \rightarrow \mathbf{S} \rightarrow \text{Prop}$ .

Parameter  $\text{top}_{\mathcal{P}} : \forall S, A_{\mathcal{P}}(D.\text{top}, S)$ .

Parameter  $A_{\text{correct}} : \forall f \mathbf{c}, \mathbf{c} \in f_{\mathbf{C}} \Rightarrow$

$\forall \mathbf{c}' S S', (S, f_{\mathbf{C}}(\mathbf{c})) \mapsto (S', \mathbf{c}') \Rightarrow \forall d, A_{\mathcal{P}}(d, S) \Rightarrow A_{\mathcal{P}}(A_i(d, \mathbf{c}), S')$ .

Parameter  $A_{\geq D} : \forall d_1 d_2, d_2 \geq_D d_1 \Rightarrow \forall S, A_{\mathcal{P}}(d_1, S) \Rightarrow A_{\mathcal{P}}(d_2, S)$ .

End.

Module Make\_Analysis (AE : ANALYSIS\_ENTRIES) <: ANALYSIS.

First, the semi-lattice's top element should be compatible with any state ( $\text{top}_{\mathcal{P}}$ ). At the entry point we make the assumption that nothing is known, hence the initial abstract state of the analysis is top. This property ensures that any initial state will satisfy the analysis property at the entry point. Second, the parameter  $A_{\text{correct}}$  ensures that each time the Kildall algorithm analyzes an instruction the result will be compatible with the states occurring in the concrete execution. Then, the last property  $A_{\geq D}$  expresses that the order relation on abstract states should be compatible with the subset ordering on  $S$ . This ensures that each time Kildall's algorithm computes a new upper bound, the result remains compatible.

## 4.2 Program Transformations

The second part of an optimization is a transformation function that uses the results of the analysis to change the code, instruction per instruction. Here again, there is a generic structure for CP and CSE. First, a transformation is given by two functions  $T_F$  and  $T_P$  to transform function declarations and whole programs, respectively. The correctness of this transformation is expressed by saying that executing a program to completion should return the same result and equivalent states, whether one executes the original program or the optimized one ( $T_{\text{correct}}$ ). Two states are equivalent when all locations in the memory have the same value, except that unoptimized code is replaced with optimized code, using a generic replacement map called  $T_{\text{state}}$ .

Module Type TRANSFER.

Parameter  $T_F : F \rightarrow F$ .

Parameter  $T_P : P \rightarrow P$ .

Parameter  $T_{\text{correct}} : \forall p S v, p \xrightarrow{P} (S, v) \Rightarrow T_P(p) \xrightarrow{P} (T_{\text{state}}(T_F, S), v)$ .

End.

Here again, correct transformations can be obtained in a variety of ways, but we rely on a generic implementation that takes a correct instruction-wise transformation as a parameter.

We actually describe instruction-wise transformations that rely on an analysis  $A$ . The transformation itself is given by a function  $T_i$ , which takes as input an instruction and the abstract state capturing what the compiler knows

about the state before executing this instruction, and returns a new instruction. The correctness of this transformation is then expressed by a few extra conditions. First, the successors of the new instruction should be a subset of the original successors ( $T_{\text{edges}}$ ). Second the execution of the transformed instruction in a state that is compatible with the abstract state obtained from the analysis should lead to the same new state and the new location in the code ( $T_{\text{correct}}$ ). Third, a return instruction should be left unchanged ( $T_{\text{return}}$ ). Fourth, the compatibility of a state with respect to an abstract state should not depend on the actual code functions that is stored in memory. All these aspects of an instruction-wise transformation are gathered in a module type `TRANSFER_ENTRIES`.

Correct instruction-wise transformations never remove any instruction from the code graph. On the other hand, they may remove edges, if the analysis makes it possible to infer that these edges are never traversed during execution. This treatment also means that dead code is optimized at the same time as useful code. The actual removal of dead code is not part of these optimizations, it is just a side effect of the later translation from the LTL language to the Linear language.

Module Type `TRANSFER_ENTRIES`.

Declare Module  $A : \text{ANALYSIS}$ .

Parameter  $T_i : A.D \rightarrow I \rightarrow I$ .

Parameter  $T_{\text{edges}} : \forall d \ i \ s, s \in \text{Succ } T_i(d, i) \Rightarrow s \in \text{Succ } i$ .

Parameter  $T_{\text{correct}} : \forall i \ s \ S \ S' \ d, A_{\mathcal{P}}(d, S) \Rightarrow$

$(S, i) \mapsto (S', s) \Rightarrow (S, T_i(d, i)) \mapsto (S', s)$ .

Parameter  $T_{\text{return}} : \forall d \ r \ \bar{s}, T_i(d, (\text{lreturn } r, \bar{s})) = (\text{lreturn } r, \bar{s})$ .

Parameter  $A_{\text{mem}} : \forall (g : F \rightarrow F) \ d \ S, A_{\mathcal{P}}(d, S) \Rightarrow A_{\mathcal{P}}(d, T_{\text{state}}(g, S))$ .

End.

Transformations are thus obtained through an application of a functor that takes modules of type `TRANSFER_ENTRIES` as input and returns modules of type `TRANSFER`. Some parts of the correctness proofs are done once and for all in this functor, but other parts are specific and done when establishing the facts  $T_{\text{edges}}$ ,  $T_{\text{correct}}$ ,  $T_{\text{return}}$ , and  $A_{\text{mem}}$ , which represent a big part of the proof (in number of lines) but are usually easy.

## 5 Instantiation

In this section we describe how we instantiate our functors to obtain the CP and CSE optimizations. The term *optimization* is a misnomer because there is no guarantee that the result code is the best possible. It is only the best that we can obtain using the information gathered by the analysis.

### 5.1 Constant Propagation

The role of constant propagation is to replace some operations by a more efficient version: for example, by a “load constant” instruction when the result of the

operation is known at compile time, or by an immediate operation when only one argument is known.

To use our predefined functor for analysis and transformation, the most difficult part is to define the semi-lattice corresponding to the domain of the analysis. Here, the goal of the analysis is to collect the known values associated to each register.

The domain of the analysis is the set of maps that associate to each register a numerical value when it is known. The set of maps from  $A$  to  $B$  is a semi-lattice if  $B$  itself is a semi-lattice and if  $A$  is a finite set. Therefore, we need to limit the number of registers. This is done using a dependent type: a register is a pair of a number (its identifier) and a proof that this identifier is less than a parameter `map_limit`.

We define the semi-lattice corresponding to the codomain of the maps as the flat semi-lattice build upon the type of values that we can associate to registers at compile time :

$$v ::= n \mid f \mid (x, n)$$

that is, an integer, a float or the address of a known global variable plus an offset. Such a construction is possible since the equality on the type  $v$  is decidable.

The analysis of an instruction consists in evaluating its result value and updating the register where the result is stored with this value. Naturally, this evaluation is only possible if all the arguments of the instruction have known values. Otherwise, the result register is set to `top`. The correctness criterion for this analysis is that if it claims that a register has a known value at some point, the actual value of the register at this point in every execution is equal to the claimed value.

Then, the transformation of an instruction first tries to evaluate the instruction and replaces the instruction by an instruction that directly stores the result in a register, when this is possible. For example, if  $r_1 = 1$  and  $r_2 = 3$  the instruction `lop Oadd  $r_1$   $r_2$   $r_d$`  will be replaced by `lop (Oconstint 4)  $r_d$` .

If only some of the arguments are known, then the transformation tries to replace the instruction by a more efficient one. For example, if  $r_2 = n$ , then `lop Omul  $r_1$   $r_2$   $r_d$`  will be replaced by `lop (Oconstint 0)  $r_d$`  if  $n = 0$ , by `lop Omove  $r_1$   $r_d$`  if  $n = 1$ , by a left shift over  $r_1$  if  $n$  is a power of two, or by a immediate multiplication `lop (OmulImm  $n$ )  $r_1$   $r_d$`  otherwise.

Memory access is also optimized using a based addressing mode if the address is known at compile time. Finally, conditional branches are replaced by an `Inop` instruction, actually implementing an unconditional branch to the appropriate successor, if the result of the test is statically evaluable.

## 5.2 Common Subexpression Elimination

A program will frequently include multiple computations of equivalent expressions, e.g. array address calculations. The goal of common subexpression elimination is to factor out these redundant computations. An occurrence of an expression  $E$  is called a *common subexpression* if  $E$  was previously calculated and the values associated with the register that appear in  $E$  have not changed

since the previous computation. In this case the compiler can replace the second occurrence of  $E$  by an access to the register containing the result of the first occurrence of  $E$ , thus saving the cost of recomputing  $E$ .

For example the sequence :

$$r_3 = r_1 + r_2; \quad r_4 = r_1; \quad r_5 = r_4 + r_2$$

will be replaced by :

$$r_3 = r_1 + r_2; \quad r_4 = r_1; \quad r_5 = r_3$$

Here the result of the analysis of an instruction is a map that associates to each register a *symbolic expression* corresponding to the operation that is stored in it. More precisely, the values associated to registers are a pair of a symbolic expression (the expression whose result is contained in the register) and a set of registers which contain the same value. The latter set is useful to keep track of multiple registers containing the result of the same computation, as often occurs following `Imove` instructions, and determine equivalence of symbolic expressions up to `Imove` instructions. For instance, in the code fragment above, it is known that the symbolic expressions  $r_1 + r_2$  and  $r_4 + r_2$  are equivalent because  $r_1$  and  $r_4$  contain the same value. Furthermore, we want to keep track that  $r_3$  and  $r_5$  contain the same value, even after a modification of  $r_2$  that will invalidate the expression associated to both registers.

A concrete register set matches the result of the analysis at a given program point if the actual values of the registers satisfy the equalities between registers stated in the abstract values. The mappings of registers to symbolic expressions can be equipped with a well-founded semi-lattice structure, with the least upper bound operation corresponding roughly to the intersection of constraints. This semi-lattice construction is complex and we omit it here due to lack of space.

The analysis of an instruction that computes  $E$  and stores its result in a register  $r_d$  first erases all references to  $r_d$  in the map. If the register  $r_d$  is not used as arguments of the instruction then the analysis tries to find a register containing an expression equivalent to  $E$ . In this case, the analysis updates the value associated to  $r_d$  with the expression and the set of registers that contain the equivalent expression. It also adds the fact that registers in the set are all equal to  $r_d$ . In all other cases, the value associated with  $r_d$  is set to the expression  $E$  and an empty set of equal registers.

The transformation of an instruction  $r_d = E$  is refreshingly simple: it becomes  $r_d = r$  (a `Imove` instruction) if there exists a register  $r$  that contains the value of  $E$ , and is left unchanged otherwise. The `Imove` instructions thus introduced can disappear later during register allocation, if the register allocator manages to assign the same hardware register or stack location to both  $r$  and  $r_d$ .

## 6 Related Work and Conclusions

In parallel with our work, several other teams developed machine-checked proofs of data-flow analyses and transformations. For instance, Cachera et al [1] develop

in Coq a general framework for lattices and data-flow equations over these lattices. Their framework is more general than ours, but they do not consider program transformations. Proofs of Kildall’s work-list algorithm can also be found in machine-checked formalizations of Java byte-code verification such as [3].

Lerner et al [4] propose Rhodium, a domain-specific language to express data-flow analyses and transformations exploiting the results of these analyses. Rhodium generates an implementation of both the analysis and the transformation and proof obligations that, once checked, imply the semantic correctness of the transformation. Rhodium specifications are definitely more concise than instantiations of our generic framework. However, the generated proof obligations do not appear significantly simpler than the proofs we have to provide for our functor applications. Moreover, the Rhodium tools must be trusted to generate correct implementations and sufficient proof obligations, while we obtain these guarantees by doing everything directly in Coq.

In summary, by exploiting the Coq module system, we have been able to factor out a significant part of specifications and correctness proofs for forward data-flow analyses and optimizations that exploit the results of these analyses. While the application of this framework to constant propagation is fairly standard, we believe we are the first to develop a mechanically verified proof for common subexpression elimination using the “Herbrand universe” approach.

While this is not described in this paper, backward data-flow analyses is handled by a simple extension of our framework (basically, just reversing the control flow graph before using the forward analysis framework). We used this to prove correct liveness analysis as used in our register allocation phase.

A limitation of our approach, also found in Rhodium, is that we prove the correctness of transformations using simulation arguments that apply only when every instruction of the source code is mapped to zero, one or several instructions of the transformed code, and the states “before” and “after” the source instruction must match the states “before” and “after” the sequence of transformed instructions. This is not sufficient to prove some optimizations such as code motion, lifting of loop-invariant computations, or instruction scheduling, where computations occur in a different order in the source and transformed code. Because of this limitation, we envision to perform some optimizations such as loop optimizations not on the unstructured RTL intermediate code, but on the structured Cminor source code, whose big-step semantics makes it easier to reorder computations without worrying about intermediate computational states that do not match.

## References

1. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *13th European Symposium on Programming (ESOP’04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 385–400. Springer-Verlag, 2004.
2. G. A. Kildall. A unified approach to global program optimization. In *1st symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.



3. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2002.
4. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *32nd symposium on Principles of Programming Languages*, pages 364–377. ACM Press, 2005.
5. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.
6. G. C. Necula. Proof-carrying code. In *24th symposium Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
7. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.
8. M. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, 1999.

# Formalising Bitonic Sort in Type Theory

Ana Bove and Thierry Coquand

Department of Computer Science and Engineering,  
Chalmers University of Technology,  
412 96 Göteborg, Sweden  
{bove, coquand}@cs.chalmers.se

**Abstract.** We discuss two complete formalisations of bitonic sort in constructive type theory. Bitonic sort is one of the fastest sorting algorithms where the sequence of comparisons is not data-dependent. In addition, it is a general recursive algorithm. In the formalisation we face two main problems: only structural recursion is allowed in type theory, and a formal proof of the correctness of the algorithm needs to consider quite a number of cases. In our first formalisation we define bitonic sort over dependently-typed binary trees with information in the leaves and we make use of the 0-1-principle to prove that the algorithm sorts inputs of arbitrary types. In our second formalisation we use notions from linear orders, lattice theory and monoids. The correctness proof is directly performed for any ordered set and not only for Boolean values.

## 1 Introduction

Bitonic sort [3] is one of the fastest *sorting networks* [3, 13]. A sorting network is a sorting algorithm performing only comparison-and-swap operations on its data. As a consequence, the sequence of comparisons in a sorting network is not data-dependent. This makes sorting networks, and hence bitonic sort, very suitable for implementation in hardware or in parallel processor arrays. The algorithm consists of  $O(m * \log(m)^2)$  comparisons in  $O(\log(m)^2)$  stages and it works on sequences of length  $2^n$  (hence the  $m$  above should be a power of 2).

Bitonic sort is a general recursive algorithm, that is, the recursive calls are performed on arguments that not necessarily are structurally smaller than the input. Although the algorithm is short and computationally simple, it is not intuitive to understand why the algorithm works. Furthermore, formally proving its correctness is not an easy task. The only machine-checked formal proof of bitonic sort we are aware of was performed in PVS by Couturier [7]. In his proof, Couturier needed to consider a maximum of 54 cases. In addition, the type of some of the properties in [7] are rather complex, making the whole formal proof difficult to follow.

In this work, we discuss two implementations of bitonic sort in constructive type theory (see for example [14, 6]), and we describe a formal correctness proof for each of the two implementations, namely, that the result of applying bitonic sort to a sequence of elements of the correct length is a sorted permutation of

the original one. The two formalisations we present here were performed using the proof assistant Agda [1]. In addition, in our first implementation and in its correctness proof (Section 4) we also use Agda’s graphical interface Alfa [2].

When formalising the algorithm and its correctness proof we face two main problems. First, only structural recursion is allowed in type theory, that is, recursive definitions in which each recursive call is performed on arguments structurally smaller than the input. In this way, the termination of a recursive definition can be ensured by its syntax. As a consequence, bitonic sort as commonly expressed cannot be directly translated into type theory. Second, a formal proof of the correctness of the algorithm might need to consider quite a number of different cases (see [7]). The challenge here is to find a suitable way of formalising the notion of *bitonic sequence* such that the properties associated with it can be easily proved and understood, without requiring too many cases.

In our first implementation we define the bitonic sort algorithm over dependently typed binary trees, that is, binary trees indexed by their height, with information in the leaves. In this way, a dependent binary tree of height  $n$  contains exactly  $2^n$  elements. In addition, the algorithm becomes structurally recursive on the height of the tree and it can be straightforwardly defined in the theory.

To prove that the algorithm sorts its input we use the *0-1-principle* [13]. It states that if a sorting algorithm sorts sequences of 0’s and 1’s using only comparison-and-swap operations on its data, it will also sort sequences of arbitrary types. The proof of the sorting property that we present here considers a maximum of 24 cases grouped in six main cases plus 23 cases leading to a contradiction (empty cases). Each case is easy to prove and understand.

In our second implementation we use notions from linear orders, lattice theory and monoids, and we directly proved the correctness theorem for any ordered set. Here, we consider a maximum of three cases plus five empty cases.

The rest of the paper is organised as follows. Section 2 contains a brief description of the Agda notation for those not familiar with this proof assistant. In Section 3 we introduce bitonic sort and we explain how it works. In Section 4 we present our dependently-typed version of the algorithm and we describe its correctness proof. Section 5 uses notions from linear orders, lattice theory and monoids to formalise the algorithm and to prove its correctness. Finally, in Section 6 we discuss some conclusions and related work.

## 2 Brief Description of the Agda Notation

If  $A$  is a type and  $B$  is a family of types over  $A$ , we write  $(x : A) \rightarrow B(x)$  for the type of functions from  $A$  to  $B$ . If  $B$  does not depend on  $A$ , we might simply write  $A \rightarrow B$  for the function type. If  $f$  is a functions from  $A$  to  $B$ , we write  $f :: (x : A) \rightarrow B(x)$ . Function types have abstractions as canonical elements which we write  $\lambda(x : A) \rightarrow e(x)$ , for  $e$  an element of the right type. Alternatively,  $f$  can be defined as  $f (x : A) :: B(x)$ . In this case, the variable  $x$  is known in the body of the function without the need of introducing it with an abstraction in the body of  $f$ .

In what follows, `False` represents the empty set (absurdity), `True` is the set containing only the element `tt`, and `T` and `F` are functions lifting boolean values into sets such that `T false = False`, `T true = True` and `F b = T(not b)` (where `not` is the boolean negation). In addition, `&&` and `||` represent logical conjunction and disjoint on sets, respectively, and `(x)` and `(+)` represent conjunction of sets and disjoint union of the sets, respectively. Canonical elements in the set  $A \times B$  have the form  $\langle a, b \rangle$  for  $a :: A$  and  $b :: B$ .

In Section 5 we make use of Agda’s implicit arguments and signature types. To indicate that  $x$  is an implicit argument in a function type we indistinctly write  $(x :: A) \mapsto B$  or  $f \ (|x :: A) :: B(x)$ . The corresponding notation for abstractions is  $\lambda(x :: A) \mapsto e(x)$ . Signatures define unordered labelled dependent products. If  $S$  is an element in a signature type containing a label  $x$ ,  $S.x$  selects the  $x$  field from  $S$ . The operator  $(.)$  is called projection.

To make the reading of the Agda code that we present here a bit easier, we might not transcribe it with its exact syntax but with a simplified version of it.

### 3 Functional Bitonic Sort

The bitonic sort algorithm that we take as our starting point is the Haskell [12] algorithm presented in Figure 1. Notice that the recursive calls in the function `merge` are performed on non-structurally smaller arguments.

This algorithm works on complete binary trees with information on the leaves and where both left and right subtrees have the same height. Since the `Tree` structure do not guaranty these conditions, the algorithm in Figure 1 is undefined on trees that do not satisfy them. It is possible to construct Haskell trees satisfying the above conditions with the following nested recursive data type: `data Tr a = Lf a | Bin (Tr (a,a))`. However, it is not easy to work with such structure. As we will see on the following two sections, dependent types provides the means to organise the data exactly as we need it for this example.

Before explaining how the algorithm works, we introduce the notion of *bitonic sequence*. Essentially a bitonic sequence is the juxtaposition of two monotonic sequences, one ascending and the other one descending, or it is a sequence such that a cyclic shift of its elements would put them in such a form.

**Definition 1.** *A sequence  $a_1, a_2, \dots, a_m$  is bitonic if there is a  $k$ ,  $1 \leq k \leq m$ , such that  $a_1 \leq a_2 \leq \dots \leq a_k \geq \dots \geq a_m$ , or if there is a cyclic shift of the sequence such that this is true.*

The main property when proving that the algorithm sorts its input is that, given a bitonic sequence of length  $2^n$ , the result of comparing and swapping its two halves gives us two bitonic sequences of length  $2^{n-1}$  such that all the elements in the first sequence are smaller than or equal to each of the elements in the second one.

So, if `bitonicSortT` sorts its input up, then the first call to the function `merge` is made on a bitonic sequence. This is simple because the left subtree is sorted up and the right subtree is sorted down. Now, `merge` calls the function

```

data Tree a = Lf a | Bin (Tree a) (Tree a)

bitonic_sortT :: Tree Int -> Tree Int
bitonic_sortT = bitonicSortT cmpS
  where cmpS x y = if x <= y then (x,y) else (y,x)

bitonicSortT:: (a -> a -> (a,a)) -> Tree a -> Tree a
bitonicSortT cmp (Lf x) = Lf x
bitonicSortT cmp (Bin l r) = merge (Bin (bitonicSortT cmp l)
                                     (reverseT (bitonicSortT cmp r)))

  where reverseT (Lf x) = Lf x
        reverseT (Bin l r) = Bin (reverseT r) (reverseT l)

        merge (Lf x) = Lf x
        merge (Bin l r) = Bin (merge l1) (merge r1)
          where (l1,r1) = min_max_Swap l r

        min_max_Swap (Lf x) (Lf y) = (Lf l,Lf r)
          where (l,r) = cmp x y
        min_max_Swap (Bin l1 r1) (Bin l2 r2) = (Bin a c, Bin b d)
          where (a,b) = min_max_Swap l1 l2
                (c,d) = min_max_Swap r1 r2

```

Fig. 1. Haskell version of the bitonic sort on binary trees

`min_max_Swap` on its two subtrees, which will pairwise compare and swap the elements. If the bitonic sequence had length  $2^n$ , then `min_max_Swap` returns two bitonic sequences of length  $2^{n-1}$  such that all the elements in the first sequence are smaller than or equal to each of the elements in the second one. Next, we call the function `merge` recursively on each of these two bitonic sequences, and we obtain four bitonic sequences of length  $2^{n-2}$  such that all the elements in the first sequence are smaller than or equal to each of the elements in the second sequence, which in turn are smaller than or equal to each of the elements in the third sequence, which in turn are smaller than or equal to each of the elements in the fourth sequence. This process is repeated until we have  $2^n$  bitonic sequences of one element each, where the first element is smaller than or equal to the second one, which in turn is smaller than or equal to the third one, and so on.

## 4 Dependently-Typed Bitonic Sort

This section describes a formalisation of bitonic sort using dependent types. A more detailed presentation of such formalisation can be found in [5].

Let us assume we have a set  $A$  and an inequality relation on  $A$  ( $(\leq) :: A \rightarrow A \rightarrow \text{Bool}$ ). Both  $A$  and  $(\leq)$  will act as global parameters in Agda. We define the type-theoretic datatype of binary trees indexed by its height and two functions constructing elements in this type.

```
DBT (n :: Nat) :: Set = case n of (zero) -> A
                                (succ n') -> DBT n' x DBT n'
```

```
DLf (a :: A) :: DBT zero = a
```

```
DBin (n :: Nat)(l, r :: DBT n) :: DBT (succ n) = <l,r>
```

Elements of this datatype are complete binary trees where both subtrees are of the same height; thus a tree of height  $n$  contains exactly  $2^n$  elements.

Using this datatype we can straightforwardly translate the Haskell version of the bitonic sort algorithm from Figure 1 into type theory. We present the dependently typed bitonic sort in Figure 2. Observe that all the functions in Figure 2 are structurally recursive on the height of the input tree. For the sake of simplicity, in what follows, we might omit the height of the tree in calls to any of these functions.

```
reverse (n :: Nat) (t :: DBT n) :: DBT n
= case n of (zero) -> t
            (succ n') -> case t of <l,r> -> DBin n' (reverse n' r)
                          (reverse n' l)

min_max_Swap (cmp::A -> A -> AxA)(n::Nat)(l,r::DBT n) :: DBT n x DBT n
= case n of (zero) -> cmp l r
            (succ n') -> case l of <l1,r1> ->
                          case r of <l2,r2> ->
                                let <a,b> = min_max_Swap cmp n' l1 l2
                                    <c,d> = min_max_Swap cmp n' r1 r2
                                in <DBin n' a c, DBin n' b d>

merge (cmp::A -> A -> AxA) (n::Nat) (t::DBT n) :: DBT n
= case n of (zero) -> t
            (succ n') -> case t of <l,r> ->
                          let <a,b> = min_max_Swap cmp n' l r
                          in DBin n' (merge cmp n' a) (merge cmp n' b)

bitonicSort (cmp::A -> A -> AxA) (n::Nat) (t::DBT n) :: DBT n
= case n of (zero) -> t
            (succ n') -> case t of <l,r> ->
                          merge cmp (succ n') (DBin n' (bitonicSort cmp n' l)
                          (reverse n' (bitonicSort cmp n' r)))

cmpS (a, b :: A) :: A x A = if (a <= b) then <a,b> else <b,a>

bitonic_sort (n :: Nat) (t :: DBT n) :: DBT n = bitonicSort cmpS n t
```

Fig. 2. Dependently-typed Bitonic sort

## 4.1 The Permutation Property

Proving that the resulting sequence is a permutation of the original one is rather easy. In our proof, we convert trees into lists (defined as expected in type theory) and we prove the permutation property on lists rather than on trees. For our purposes, a permutation on lists is any equivalence relation on lists of the same length (although this is not a formal part of the definition, it could be easily derived from it) that is both commutative and a congruence with respect to concatenation.

## 4.2 The Sorting Property

We start by defining when a tree is sorted. Given the element  $a :: A$  and the trees  $t1$  and  $t2$ , we define the relations  $t1 /<= a$  and  $t1 /<=\ t2$  by recursion on  $t1$  and  $t2$ , respectively, such that  $t1 /<= a$  is satisfied if all the elements in  $t1$  are smaller than or equal to  $a$ , and  $t1 /<=\ t2$  is satisfied if all the elements in  $t1$  are smaller than or equal to each of the elements in  $t2$ . Finally we define:

```
Sorted (n :: Nat) (t :: DBT n) :: Set
= case n of (zero) -> True
            (succ n') -> case t of <l,r> ->
                Sorted n' l && Sorted n' r && l /<=\ r
```

Proving that the resulting sequence is sorted is not trivial. To start with, we need to formalise the notion of bitonic sequence in such a way that it allows proving the necessary properties in a nice way. To this end, we fix the set  $A$  of elements in the tree to the set `Bool` and we make use of the 0-1 principle to generalise our result. In what follows we identify 0 with `false` and 1 with `true`. The 0-1 principle states that if a sorting algorithm sorts sequences of 0's and 1's performing only comparison-and-swap operations on its data, then it also sorts sequences of arbitrary types. We use Reynolds parametricity theorem [15] to prove the 0-1 principle, whose proof follows those in [8] and [10]. The reader is referred to [5] for more details on our proof of this principle.

**Bitonic Sequences and Bitonic Labels.** Since we now consider only boolean sequences, our definition of a bitonic sequence becomes simpler.

**Definition 2.** *A 0-1-sequence  $a_1, \dots, a_m$  is called bitonic, if it contains at most two changes between 0 and 1.*

To determine if the sequence in a binary tree is bitonic we assign *bitonic labels* to the trees. We introduce one label for each of the six possible bitonic sequence and one extra label  $W$  that will be assigned to trees whose sequences are not bitonic.

```
BitLb :: Set = data 0 | I | OI | IO | OIO | IOI | W
```

In addition, we define an equivalence relation (`==`) (`l1, l2 :: BitLb`) `:: Set` on bitonic labels, along with the property `notW (l :: BitLb) :: Set` of not being the label `W`, and a function `bin_label (l1, l2 :: BitLb) :: BitLb` that combines two labels into a new one. The combined label is `W` in many cases, for example `bin_label OI OIO = W`.

Since we have seven labels, many binary functions on labels need to consider up to 49 cases (sometime we do not need to consider all cases, for example, for any `l`, `bin_label W l = W`). All the functions we need on labels are quite trivial.

Below we show how to assign labels to binary trees and we define the property of being a bitonic sequence.

```
label (n :: Nat) (t :: DBT n) :: BitLb
= case n of (zero) -> case t of (true) -> I
                                     (false) -> 0
          (succ n') -> case t of <l,r> ->
                                     bin_label (label n' l) (label n' r)
```

```
Bitonic (n :: Nat) (t :: DBT n) :: Set = notW (label n t)
```

We use the information given by the labels to reason about the results of the operations we perform on a tree. For example, the following lemma gives us information about the result of the `min_max_Swap` operation.

```
label_0_x2min_max_Swap_label_0_x (cmp :: Bool-> Bool-> Bool x Bool)
  ( ... ) (n :: Nat) (l, r :: DBT n) (label l == 0)
  :: (label (fst (min_max_Swap cmp l r)) == 0) &&
     (label (snd (min_max_Swap cmp l r)) == label r)
```

The lemma is proved by induction on the height of the trees. Here, we use the fact that if the label of `l` is `0`, then either `l` is simply `false`, or it is a binary tree whose both subtrees also have label `0`. In the lemma, we need to assume that the operation `cmp` behaves as we want it to with respect to labels. Here we write `( ... )` for such assumptions. These assumptions are used to prove the base cases in the lemma.

Tree labels can also give information about the order of the trees. Below we show a couple of lemmas that can be easily proved by induction on `m`.

```
label_02leq (n, m :: Nat) (t1 :: DBT n) (t2 :: DBT m)
  (label t1 == 0) :: t1 /<=\ t2
```

```
leq_label_OI_0 (n, m :: Nat) (t1 :: DBT n) (t2 :: DBT m)
  (label t1 == OI) (label t2 == 0) (t1 /<=\ t2) :: False
```

We also need lemmas relating the label of the trees to the result of `reverse` as:

```
reverse_label_OI2label_IO (n :: Nat) (t :: DBT n)
  (label (reverse t) == OI) :: label t == IO
```

which can be easily proved by induction on the height of the tree.



Finally, we relate labels to the property of being a sorted tree.

```
sorted2label_0_OI_I (n :: Nat) (t :: DBT n) (Sorted t)
  :: (label t == 0) || (label t == 0I) || (label t == I)

sortedDown2label_0_IO_I (n::Nat) (t::DBT n) (Sorted (reverse t))
  :: (label t == 0) || (label t == IO) || (label t == I)
```

**Bitonic Properties.** We can now prove the two main properties concerning bitonic sequences. The first property is as follows:

```
sorted_sortedDown2bitonic (n :: Nat) (t1, t2 :: DBT n)
  (Sorted t1) (Sorted (reverse t2)) :: Bitonic (DBin t1 t2)
```

This proof is straightforward after considering all possible combinations in the results of `sorted2label_0_OI_I` and `sortedDown2label_0_IO_I`.

Next we state the second property.

```
bitonic2min_max_Swap (cmp :: ...) ( ... ) (l , r :: DBT n)
  (Bitonic (DBin l r))
  :: Bitonic (fst (min_max_Swap cmp l r)) &&
     Bitonic (snd (min_max_Swap cmp l r)) &&
     fst (min_max_Swap cmp l r) /<=\ snd (min_max_Swap cmp l r)
```

The proof is performed by cases both on `label l` and on `label r`. We consider 43 cases, two of them containing three subcases each; hence 47 cases in total. Only 24 cases were valid ones in the sense that no contradiction could be derived from the hypotheses and the labels of the trees. An example of an invalid case is when we have `label l == 0`, `label r == IOI` and `Bitonic (DBin l r)`. The 24 valid cases can be divided into six groups: either the left or right tree has label `0` or label `I`, or the trees have labels `0I` and `IO`, or `IO` and `0I`. Each of these cases are proved by applying previous lemmas.

**Sorted Properties.** Before proving that our algorithm sorts sequences of booleans, we prove some auxiliary lemmas.

```
leq2min_max_Swap_leqL (cmp :: ...) ( ... ) (n,m :: Nat)
  (t1,t2 :: DBT n) (t :: DBT m) (t1 /<=\ t) (t2 /<=\ t)
  ::fst(min_max_Swap t1 t2) /<=\ t && snd(min_max_Swap t1 t2) /<=\ t

leq2merge_leqL (cmp :: ...) ( ... ) (n,m :: Nat) (t1 :: DBT n)
  (t2 :: DBT m) (t1 /<=\ t2) :: merge t1 /<=\ t2
```

We also prove symmetric lemmas `leq2min_max_Swap_leqR` and `leq2merge_leqR`, where the operations `min_max_Swap` and `merge` appear to the right of the symbol `/<=\`. All these lemmas are proved by induction on the height of the trees.

We can now prove that the result of merging a bitonic tree is sorted.

```
mergeSorted (cmp :: ...) (...) (n :: Nat) (t :: DBT n) (Bitonic t)
  :: Sorted (merge t)
```

The interesting case is when  $t$  has the form  $\langle l, r \rangle$ . Let  $\langle a, b \rangle$  be the result of `min_max_Swap cmp l r`. The result of `merge t` is `DBin (merge a) (merge b)`.

Using `bitonic2min_max_Swap` we know `Bitonic a`, `Bitonic b` and  $a \leq b$ . By the inductive hypotheses, we have `Sorted (merge a)` and `Sorted (merge b)`.

Using the lemmas `leq2merge_leqL` and `leq2merge_leqR`, and the fact that  $a \leq b$ , we get `merge a ≤ merge b`. This concludes the proof.  $\square$

We now prove that our bitonic sort returns a sorted tree.

```
bitonicSortSorted (cmp :: ...) ( ... ) (n :: Nat) (t :: DBT n)
  :: Sorted (bitonicSort t)
```

Again, the interesting case is when  $t$  has the form  $\langle l, r \rangle$ . By the inductive hypotheses we know `Sorted (bitonicSort l)` and `Sorted (bitonicSort r)`. Hence, `reverse (bitonicSort r)` is sorted down.

Using the property `sorted_sortedDown2bitonic`, we obtain that `Bitonic (DBin (bitonicSort l) (reverse (bitonicSort r)))`.

The premises of `mergeSorted` are now satisfied. Hence we can conclude that `Sorted (merge (DBin (bitonicSort l) (reverse (bitonicSort r))))`.  $\square$

It only remains to prove that the specific function `cmpS` satisfies all the properties (six) that we have assumed for the argument function `cmp` (in the lemmas above we just referred to them as `( ... )`). They are all trivial when the elements we consider are of type `Bool`.

We can now establish that our bitonic algorithm sorts sequences of booleans by applying the lemma `bitonicSortSorted` to our specific operation `cmpS` and to the proofs that `cmpS` behaves as needed.

## 5 Bitonic Sort Using Lattice Theory

### 5.1 Motivations

In this section we use notions from lattice theory, linear orders and monoids for the formalisation of bitonic sort and its correctness proof. We first give some heuristic motivations for this approach.

The 0-1 principle is reminiscent of Birkhoff representation theorem [4] that states that any distributive lattice is a sublattice of a power of the lattice  $\{0, 1\}$ . Another way to formulate this is to say that an identity between lattice expressions hold in all lattices if and only if it holds in the lattice  $\{0, 1\}$ . The 0-1 principle can be reformulated as the fact that a sequence of elements in a linear order  $D$  is bitonic if and only if its image by any representation map  $D \rightarrow \{0, 1\}$  is bitonic. Now, to say that a sequence of elements  $x_i, i < n$  in  $\{0, 1\}$  is bitonic can be formulated as the fact that whenever  $i < j < k < l < n$ , we cannot have neither  $x_i = x_k = 1$  and  $x_j = x_l = 0$  nor  $x_j = x_l = 1$  and  $x_k = x_i = 0$ ,

that is the two sequences  $\dots 0 \dots 1 \dots 0 \dots 1 \dots$  and  $\dots 1 \dots 0 \dots 1 \dots 0 \dots$  are not allowed. We can express purely lattice theoretically this in the following way

$$x_i \wedge x_k \leq x_j \vee x_l \qquad x_j \wedge x_l \leq x_k \vee x_i \qquad (*)$$

and we can now take this as a characterisation of bitonic sequences in general: a sequence of elements  $x_i$ ,  $i < n$  in a linear order  $D$  is bitonic if and only if whenever  $i < j < k < l < n$  the relations  $(*)$  hold. Here, we have used the notation  $x \wedge y$  (respectively  $x \vee y$ ) to denote the minimum (respectively maximum) of  $x$  and  $y$ . Notice that the above definition makes sense in any distributive lattice  $D$ . (The generalisation of sorting to elements in a distributive lattice is also considered in exercises in [13].) In this way, we find a direct definition of being a bitonic sequence which does not refer to the consideration of cyclic shift of a sequence like in Definition 1. Notice that this new definition of bitonic is invariant in a cyclic permutation of  $i, j, k, l$ .

We explain now how to represent mathematically the notion of permutation of a sequence. We want to formalise the idea that a sequence  $y_1, \dots, y_n$  is obtained from a sequence  $x_1, \dots, x_n$  only by doing comparison-and-swap operations. It is actually clearer, to consider the more general case of distributive lattices. A comparison-and-swap operation consists in replacing elements  $x_i, x_j$  with  $i < j$  by the elements  $x_i \wedge x_j, x_i \vee x_j$ . We formalise this using ideas from universal algebra. We represent that  $y_1, \dots, y_n$  is obtained from a sequence  $x_1, \dots, x_n$  only by doing comparison-and-swap operation by the equality  $\Sigma\mu(x_i) = \Sigma\mu(y_i)$  for all maps  $\mu : D \rightarrow M$  in a commutative monoid satisfying

$$\mu(x \wedge y) + \mu(x \vee y) = \mu(x) + \mu(y) \qquad (**)$$

Such maps, called valuation maps, are important in the theory of distributive lattices and in measure theory [11, 16]. In the case where  $D$  is a linear order, it can be shown that to have  $\Sigma\mu(x_i) = \Sigma\mu(y_j)$  for all such maps is equivalent to the fact that  $y_1, \dots, y_n$  is a permutation of  $x_1, \dots, x_n$ . Usually, for instance in the reference [16],  $M$  is fixed and taken to be the free commutative monoid generated by the elements  $\mu(x)$ ,  $x \in D$  and the relation  $(**)$ . However instead of working with this fixed monoid, it is equivalent and more convenient to work with *all* commutative monoids and maps  $\mu$  satisfying  $(**)$ . This turns out to be also well-suited for the formalisation in type theory: to express the notion of “arbitrary” commutative monoid and “arbitrary” map satisfying  $(**)$ , we simply introduce a new variable  $M$ , with the axioms that this forms a commutative monoid, and a new variable  $\mu$  with the axioms that this satisfies the relation  $(**)$ . Even in the case where  $D$  is a linear order, this appears to be the right mathematical way to express that  $y_1, \dots, y_n$  is a permutation of  $x_1, \dots, x_n$ .

## 5.2 Formalisation in Type Theory

In order to carry out the actual representation of these mathematical definitions in type theory, it is simpler to work with sequences as being functions from a (finite) decidable linear order to an ordered set. (Intuitively, we represent a

sequence as an array of elements.) A *decidable linear order* DLO consists of a set  $I$  and an inequality relation  $(<) :: I \rightarrow I \rightarrow \text{Bool}$  that is irreflexive, transitive and linear. We denote  $I0$  the linear order whose set contains only one element  $\text{tt}$  and such that  $\text{tt} < \text{tt}$  evaluates to **false**. Given two linear orders  $L$  and  $R$ , we define the linear order  $L + R$  as the linear order whose set is the disjoint sum of the sets in  $L$  and  $R$ , and such that any element in the set of  $L$  is smaller than any element in the set of  $R$ . We can now define a function  $\text{IN}$  from the Natural numbers into decidable linear order in such a way that  $\text{IN } n$  contains  $2^n$  elements.

```
IN (n :: Nat) :: DLO = case n of (zero)-> I0
                                (succ n')-> IN n' + IN n'
```

A *lattice* consists of a set  $D$ , an inequality relation  $(<=) :: D \rightarrow D \rightarrow \text{Set}$  that is reflexive and transitive, and a minimum  $(/\wedge) :: D \rightarrow D \rightarrow D$  and a maximum  $(/\vee) :: D \rightarrow D \rightarrow D$  operations with the expected properties. The lattice is *distributive* if  $(/\wedge)$  and  $(/\vee)$  satisfies the distributive laws.

A *monoid* consists of a set  $M$ , an equivalence relation  $(==) :: M \rightarrow M \rightarrow \text{Set}$ , and an associative and congruent operation  $(+) :: M \rightarrow M \rightarrow M$ . The monoid is *commutative* if  $(+)$  is commutative.

Linear orders, lattices and monoids are defined as signature types in Agda. Hence, selecting their components is performed with the projection operator  $(.)$ .

We define sequences as functions from linear orders to distributive lattices

```
Sequence (DI::DLO) (f::DI.I -> D) :: Set
= (i,j::DI.I) -> i == j -> f i == f j
```

where the equality relations over linear order and over lattices are defined as expected.

The predicates that state if a sequence is increasing  $\text{Incr}$  or decreasing  $\text{Decr}$ , and the relation  $(<=<=)$  stating that all the elements in the first sequence are smaller than or equal to any element in the second sequences are defined as expected. For example,  $(<=<=)$  is defined as:

```
(<=<=) (|DI::DLO) (|DJ::DLO) (|f::DI.I -> D) (|g::DJ.I -> D)
      (seqf::Sequence DI f) (seqg::Sequence DJ g) :: Set
= (i::DI.I)-> (j::DJ.I)-> (f i <= g j)
```

Bitonic sequences are defined as follows:

```
Bitonic (|DI::DLO) (|f::DI.I -> D) (seqf::Sequence DI f) :: Set
= (i,j,k,l::DI.I) -> (ls_ij::T (i < j)) ->
  (ls_jk::T (j < k)) -> (ls_kl::T (k < l)) ->
  (f i /\ f k <= f j \/ f l && f j /\ f l <= f i \/ f k)
```

If  $\text{seqf}$  is a sequence over the linear order  $\text{IN } (\text{succ } n)$ , then selecting the left and the right sequences of the tree domain produce sequences as a result. These operation are called  $\text{leftSeq}$  and  $\text{rightSeq}$  respectively. In addition,

```

mergeF (n::Nat) (f::(IN n).I -> D) :: (IN n).I -> D
  = case n of (zero)-> f
              (succ n')->
                let inlf = leftF n' f; inrf = rightF n' f
                in  conF (IN n') (IN n') (mergeF n' (minF (IN n') inlf inrf))
                  (mergeF n' (maxF (IN n') inlf inrf))

mergeSeq (n::Nat) (|f::(IN n).I -> D) (seqf::Sequence (IN n) f)
  :: Sequence (IN n) (mergeF n f)
  = case n of (zero)-> seqf
              (succ n')->
                let leftS = leftSeq n' seqf; rightS = rightSeq n' seqf
                in mergeSeq n' (leftS /\ rightS) *
                  mergeSeq n' (leftS \+/ rightS)

bitonicSort (n::Nat) (f::(IN n).I -> D) :: (IN n).I -> D
  = case n of (zero)-> f
              (succ n')->
                let inlf = leftF n' f; inrf = rightF n' f
                in  mergeF (succ n')
                  (conF (IN n') (IN n') (bitonicSort n' inlf)
                    (revF n' (bitonicSort n' inrf)))

bitonicSortSeq (n::Nat) (|f::(IN n).I -> D) (seqf::Sequence (IN n) f)
  :: Sequence (IN n) (bitonicSort n f)
  = case n of (zero)-> seqf
              (succ n')->
                let leftS = leftSeq n' seqf; rightS = rightSeq n' seqf
                in mergeSeq (succ n') (bitonicSortSeq n' leftS *
                  revSeq n' (bitonicSortSeq n' rightS))

```

**Fig. 3.** Bitonic sort using linear orders and lattices

`revSeq seqf` produces a sequence in the reverse order. The underneath functions in the definition of the sequences are called `leftF`, `rightF` and `revF` respectively.

If `seqf` and `seqg` are sequences over the same linear order domain `DI` with functions `f` and `g`, respectively, then, `seqf /\ seqg` and `seqf \+/ seqg` produce sequences such that, for all `i` in `DI.I`, we have that `f i /\ g i` and `f i \\/ g i`, respectively. The underneath functions are called `minF` and `maxF` respectively.

If `DI` and `DJ` are linear orders, and if `sf` is a sequence over `DI` and `sg` is a sequence over `DJ` then, `sf * sg` is a sequence over the linear order `DI + DJ` with `conF` as underneath function.

Let `DL` be a distributive lattice with set `D`. Figure 3 presents the formalisation of bitonic sort using the notions we describe above.

### 5.3 The Permutation Property

Let  $L$  be a lattice with set  $D$  and  $\mathbf{CM}$  be a commutative monoid with set  $M$ . Let  $\mu :: D \rightarrow M$  be a function such that for all  $a, b :: D$  then  $\mu a + \mu b == \mu (a \wedge b) + \mu (a \vee b)$  is satisfied. We then define

```
Sigma (n::Nat) (f::(IN n).I -> D) :: M
= case n of
  (zero)-> mu (f tt)
  (succ n')-> Sigma n' (leftF n' f) + Sigma n' (rightF n' f)
```

If  $f, g :: (IN n).I \rightarrow D$ , the following properties can be easily proved by induction on  $n$  and transitivity of equality:

```
Sigma n f == Sigma n (revF n f)
```

```
Sigma n f + Sigma n g ==
  Sigma n (minF (IN n) f g) + Sigma n (maxF (IN n) f g)
```

It is also immediate to prove that

```
Sigma (succ n) (conF (IN n) (IN n) f g) == Sigma n f + Sigma n g
```

We can finally prove that

```
mergeSigma (n::Nat) (f::(IN n).I -> D)
  :: Sigma n f == Sigma n (mergeF n f)
```

```
bitonicSortSigma (n::Nat) (f::(IN n).I -> D)
  :: Sigma n f == Sigma n (bitonicSort n f)
```

by induction on  $n$ , transitivity of equality and the properties we mentioned above.

### 5.4 The Sorting Property

Let  $L$  be a lattice with set  $D$ .

Most of the properties needed on sequences in order to prove that the bitonic algorithm sorts its input are very easy to prove by induction, case analysis or almost straightforwardly. A couple of examples of such properties are:

```
incr2decr_rev (n::Nat) (|f::(IN n).I -> D)
  (seqf::Sequence (IN n) f) (up::Incr seqf)
  :: Decr (revSeq n seqf)
```

```
min_seq_LEq_max_seq (|DI::DL0) (|f, |g::DI.I -> D)
  (seqf::Sequence DI f) (seqg::Sequence DI g)
  (bit_fg::Bitonic (seqf * seqg))
  :: seqf /\ seqg <=< seqf \/ seqg
```

The proof that the result of concatenating an increasing sequence with a decreasing sequences is a bitonic sequence requires looking into three cases plus five empty cases (that is, we can derive absurdity from them).

```
incr_decr2bitonic (|DI::DLO) (|DJ::DLO) (|f::DI.I -> D)
  (|g::DJ.I -> D) (seqf::Sequence DI f) (seqg::Sequence DJ g)
  (up::Incr seqf) (dw::Decr seqg) :: Bitonic (seqf * seqg)
```

The proof goes as follows. Given  $i, j, k, l :: (DI + DJ).I$  such that  $ls_{ij} :: T (i < j)$ ,  $ls_{jk} :: T (j < k)$  and  $ls_{kl} :: T (k < l)$  we need to prove  $i \wedge k \leq j \vee l$  and  $j \wedge l \leq i \vee k$ . We have the following three cases:

- $k :: DI.I$  and hence  $i, j :: DI.I$ : Here  $i \wedge k \leq i \leq j \leq j \vee l$  and  $j \wedge l \leq j \leq k \leq i \vee k$
- $k :: DJ.I$  and  $j :: DI.I$ ; hence  $i :: DI.I$  and  $l :: DJ.I$ : Here we have that  $i \wedge k \leq i \leq j \leq j \vee l$  and that  $j \wedge l \leq l \leq k \leq i \vee k$
- $k :: DJ.I$  and  $j :: DJ.I$ ; hence  $l :: DJ.I$ : Here  $i \wedge k \leq k \leq j \vee l$  and  $j \wedge l \leq l \leq k \leq i \vee k$  □

The properties that show that if a sequence  $seqf * seqg$  is bitonic then both the sequences  $seqf \ /+\ seqg$  and  $seqf \ \+ seqg$  are bitonic require some inequality reasoning with easy results from lattice theory. The proofs are not difficult to perform but they are not too nice either due to the fact that Agda has no support for inequality reasoning. The type of the first such property is as follows:

```
bitonic_min_seq (|DI::DLO) (|f, |g::DI.I -> D)
  (seqf::Sequence DI f) (seqg::Sequence DI g)
  (bit_fg::Bitonic (seqf * seqg)) :: Bitonic (seqf \+ seqg)
```

After a few easy inductive proofs concerning the result of the merge operation, we are able to establish that both the result of merge and of the bitonic sort are increasing sequences.

```
mergeIncr (n::Nat) (|f::(IN n).I -> D) (seqf::Sequence (IN n) f)
  (btf::Bitonic seqf) :: Incr (mergeSeq n seqf)
```

```
bitonicSortIncr (n::Nat) (|f::(IN n).I -> D)
  (seqf::Sequence (IN n) f) :: Incr (bitonicSortSeq n seqf)
```

Both proofs are performed by induction on  $n$ .

## 6 Conclusions and Related Work

The major challenge and difficulty in this work was to find a suitable representation of a bitonic sequence that would allow us to prove the needed properties in a nice way and without the need of considering too many cases.

In our first formalisation (see Section 4), we define labels on the boolean binary trees to formalise the notion of bitonic sequences. We believe that this representation gives us a lot of intuition about the properties we will or we will not be able to prove, since the label of a tree gives us enough information about the kind of tree we are working with. A disadvantage of this representation is that, when considering cases on the label of the trees, we must deal with many cases that do not make sense, as it was explained before.

We believe one might be able to overcome this problem by working in a proof assistant such as Epigram [9], which provides a more powerful pattern matching facility than the one implemented in Agda. If this is the case, we could define an inductive predicate over dependent trees which exactly characterises those trees that are bitonic. When doing pattern matching on a proof that a tree is bitonic, we will then only obtain the non-empty cases.

Our second formalisation (see Section 5) used notions from linear orders, lattice theory and monoids. In general, this formalisation was shorter and more elegant than the first one.

Finally, it is interesting to point out that, despite of the different approaches we used in the two formalisations, some of the lemmas we used for proving the sorting property were needed in the two correctness proofs that we presented.

## Related Work

To the best of our knowledge, there are not many formal proofs of bitonic sort.

Couturier [7] performed a formal proof of the sorting property of bitonic sort in PVS [17] that does not use the 0-1 principle. In his work, Couturier formalised the general notion of bitonic sequences with an *array* (represented by a function from Natural numbers to Natural numbers) and three indexes: the indexes for the left-most and right-most elements, and the index for the maximum element. Most of the properties proved in [7] involve multiple indexes and several for-all statements. He also had to deal with many cases in some of his proofs, in one proof he deals with 54 cases. In our opinion, it is rather difficult to closely follow the process in [7] because of the complexity in the type of some of the properties.

The reader is referred to [5] for a more complete description of the literature about bitonic sort.

## Acknowledgements

We would like to thank Björn von Sydow for many useful discussions on bitonic sort and on the formalisation we presented here. We would also like to thank an anonymous referee for his/her valuable comments.

## References

1. Agda homepage. <http://www.cs.chalmers.se/~catarina/agda>
2. Alfa homepage. <http://www.cs.chalmers.se/~hallgren/Alfa/>



3. K. E. Batchner. Sorting networks and their applications. In *Spring Joint Computer Conference, AFIPS Proc.*, volume 32, pages 307–314, 1968.
4. G. Birkhoff. *Lattice theory*. Amer.Math.Soc., Providence, 1967.
5. A. Bove. Formalising bitonic sort using dependent types. Available on the WWW: [www.cs.chalmers.se/~bove/Papers/dt\\_bit\\_sort.ps.gz](http://www.cs.chalmers.se/~bove/Papers/dt_bit_sort.ps.gz), October 2004. Technical Report, Chalmers University of Technology.
6. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
7. R. Couturier. Formal engenieering of the bitonic sort using pvs. In *2nd. Irish Workshop on Formal Method*, Cork, Ireland, 1998.
8. N.A. Day, J. Launchbury, and J. Lewis. Logical abstractions in haskell. In *Proceedings of the 1999 Haskell Workshop*, Technical Report UU-CS-1999-28, October 1999.
9. Epigram homepage. <http://www.dur.ac.uk/CARG/epigram/>
10. Qiao Haiyan. *Testing and Proving in Dependent Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 2003.
11. A. Horn and A. Tarski. Measures in Boolean algebras. *Trans. Amer. Math. Soc.* , (64):467–497, 1948.
12. S. Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, April 2003.
13. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1973.
14. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
15. J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83, Proceedings of the 9th IFIP World Computer Congress*, pages 513–523, Paris, France, September 1983. North-Holland.
16. G. Rota. The valuation ring of a distributive lattice. In *Proceedings of the University of Houston Lattice Theory Conference*, pages 574–628, Houston, Tex., 1973.
17. J. Rushby. The pvs verification system. [www.csl.sri.com/pvs.html](http://www.csl.sri.com/pvs.html), 1998.

# A Semi-reflexive Tactic for (Sub-)Equational Reasoning

Claudio Sacerdoti Coen

Project PCRI, CNRS, École Polytechnique, INRIA, Université Paris-Sud  
Claudio.Sacerdoti@inria.fr

**Abstract.** We propose a simple theory of monotone functions that is the basis for the implementation of a tactic that generalises one step conditional rewriting by “propagating” constraints of the form  $x R y$  where the relation  $R$  can be weaker than an equivalence relation. The constraints can be propagated only in goals whose conclusion is a syntactic composition of  $n$ -ary functions that are monotone in each argument. The tactic has been implemented in the Coq system as a semi-reflexive tactic, which represents a novelty and an improvement over an earlier similar development for NuPRL.

A few interesting applications of the tactic are: reasoning in type theory about equivalence classes (by performing rewriting in well-defined goals); reasoning about reductions and properties preserved by reductions; reasoning about partial functions over equivalence classes (by performing rewriting in PERs); propagating inequalities by replacing a term with a smaller (greater) one in a given monotone context.

## 1 Introduction

Equalities are pervasive in mathematics. They are also very easy to reason about since they enjoy the *substitution property*: for each  $P$ , whenever  $x = y$  and  $P$  holds for  $x$ ,  $P$  also holds for  $y$ . In type theory, the substitution property is a trivial consequence of the definition of Leibniz equality.

Sometimes in mathematics it is necessary to consider equivalence relations that are not equalities. The usual mathematical practice is to immediately quotient the carrier w.r.t. the equivalence relation in order to use equality between classes and obtain substitutivity. Extensions of intensional type theories to accommodate quotient types have been proposed in the literature [7], but they are not used in practice. Sometimes the problem can be avoided by defining a normalisation function that picks a canonical representative for each class, replacing the equivalence relation with Leibniz equality over the representatives [3]. More often, however, the user avoids building the quotient sets and explicitly works with the equivalence relation. This approach is called the *setoid approach* [1].

The more serious drawback of working with setoids and equivalence relations is that the substitutivity property does not always hold for each context  $P$ . When it holds, the context is usually called *compatible* or *equality preserving* and the equivalence relation is said to be a *congruence* w.r.t. the context. When

the relation is generalized to be an order, the contexts are usually called *order preserving* or *monotone* and no standard terminology exists when no assumption is made on the relation. Thus I prefer to call *substitutive* the context the substitutivity property holds for independently of the properties of the relations considered. Basin, in a related work [2], calls them *functional*.

In the setoid approach, to substitute  $y$  for  $x$  in  $(P x)$  the user needs to prove that  $P$  is substitutive. In practice, much of this task can be easily automated [9]: since the composition of substitutive functions is substitutive, it is possible to build a tactic that automatically proves that a context is substitutive whenever the context is syntactically a composition of functions whose substitutive property has been previously shown by the user.

In this paper we provide a generalisation of the setoid approach by considering relations that are weaker than equivalence relations, and we characterise corresponding classes of substitutive contexts. As a result we develop a tactic that from the hypotheses  $x R y$  and  $P x$  deduces  $P y$  whenever the context  $P$ , seen as a syntactic composition of functions, can be automatically proved to be substitutive.

Our idea is not original: the theory we develop is a reformulation of a restriction of Window Inferencing [6] and an equivalent tactic has been available for years in the NuPRL [2, 8] and HOL systems. However our implementation is the first one to be semi-reflexive and our formulation of the theory paves the way to reflexive implementations. Moreover we take non reflexive relations seriously.

In Sect. 2 we present a few scenarios that motivate the study of the substitutivity property for relations weaker than equivalence relations. Sect. 3 details the relation with previous work, in particular with Windows Inferencing. In Sect. 4 we present a small theory of monotone functions, called *morphisms*. Sect. 5 describes the semi-reflexive tactic that is based on that theory. The tactic has been already integrated in the code base of the proof assistant Coq, and it will be distributed as soon as the next major version of the system is released. To use it now, a user needs to switch to the development version of Coq, available only via anonymous CVS. A sample application of the tactic is given in Sect. 6. We draw a few conclusions in Sect. 7.

## 2 Motivations

We present now a few scenarios for equational and sub-equational reasoning.

### 2.1 Equivalence Relations

The classical scenario is the one of equivalence relations. Given the goal

$$\frac{H : x \equiv y}{(P (f x))}$$

the user wants to replace  $x$  with  $y$  in the conclusion, obtaining

$$\frac{H : x \equiv y}{(P (f y))}$$

The substitution can be performed iff  $(P (f \square))$  is a substitutive context. To prove that  $(P (f \square))$  is substitutive it is sufficient (but not necessary!) to prove that there exists an equivalence relation  $\equiv'$  defined on the codomain of  $f$  such that: (1)  $f$  is substitutive w.r.t.  $\equiv$  and  $\equiv'$  (i.e. if  $\forall x, x' x \equiv x' \Rightarrow (f x) \equiv' (f x')$ ) and (2)  $P$  is substitutive w.r.t.  $\equiv'$  and  $\iff$  (i.e. if  $\forall x, x' x \equiv' x' \Rightarrow (P x) \iff (P x')$ ). The properties (1) and (2) are usually proved by the user once and for all when  $f$  and  $P$  are defined.

## 2.2 Dropping Symmetry: Preorders

The second scenario considers relations that are transitive and reflexive, but not symmetric. For instance, multi-steps reduction relations belong to this scenario. As an example, the user could be developing a theory of the  $\lambda$ -calculus, and she could face the goal

$$\frac{H : ((\lambda x.x) M) \triangleright_{\beta} M}{(Q (f M))}$$

where  $Q \circ f$  is a property preserved by reduction. Thus substituting  $((\lambda x.x) M)$  in place of  $M$  in the goal is a sound operation that yields the new goal

$$\frac{H : ((\lambda x.x) M) \triangleright_{\beta} M}{(Q (f ((\lambda x.x) M)))}$$

At first glance, the example given looks similar to the one in the previous scenario. However, the fact that the reduction relations are not symmetric introduce subtle consequences for substitutive contexts: a context  $C$  is still substitutive w.r.t. two reduction relations  $\triangleright$  and  $\triangleright'$  iff  $\forall x, x' x \triangleright x' \Rightarrow (C x) \triangleright' (C x')$ . This means that now  $C$  is substitutive iff it is a monotone *increasing* function. Now let  $\Rightarrow$  be the reflexive, transitive and asymmetric relation considered on the set of all propositions. In order to substitute  $x$  with  $y$  in  $(C x)$  it is necessary to prove that  $(C y) \Rightarrow (C x)$  and,  $C$  being a substitutive context by hypothesis, this can be done iff  $y \triangleright x$ . The conclusion is that only the right hand side of the relation can be replaced by the left hand side in the goal! In order to be able to substitute the right hand side for the left hand side the context  $C$  must be a monotone *decreasing* function.

From the previous observation we deduce that in the case of asymmetric relation the definition of substitutive context (an increasing vs a decreasing monotone function) depends on the direction chosen for the rewriting of the hypothesis.

As in the previous scenario, monotonicity is preserved by function composition. Moreover, it is easy to deduce whether the compound function is increasing or decreasing from the same knowledge on the arguments of the composition. Thus a tactic can automatically decide if a context is substitutive provided that the functions that syntactically compose the context have been already shown to be monotone.

A different example that belongs to the same scenario is the one of “rewriting” of large inequalities. For instance, given the goal

$$\frac{H : \epsilon \geq 0}{-2 * 0 \geq -2 * \epsilon}$$

the user can use the tactic to propagate the hypothesis  $H$  by replacing  $\epsilon$  with 0 in the goal, obtaining:

$$\frac{H : \epsilon \geq 0}{-2 * 0 \geq -2 * 0}$$

From the usual arithmetic properties an automatic tactic should be able to deduce that the context  $-2 * 0 \geq -2 * \square$  is substitutive (i.e. it is a monotone decreasing predicate w.r.t. the order  $\Rightarrow$ ).

### 2.3 Dropping Reflexivity: PERs

The third scenario that we consider is about irreflexive relations. For instance, irreflexive, symmetric and transitive relations (also called Partial Equivalence Relations or simply PERs) can be used in type theory to reason about functions over equivalence classes over a subset of a given set  $X$ : let  $(X, \equiv)$  be a set together with a PER over it. We say that an element  $x \in X$  is *proper* if  $x \equiv x$ . The subset of the proper elements of  $X$  forms a setoid. To represent a setoid-compatible unary function over the elements of the setoid we can use a function  $F$  whose domain is  $X$  such that  $F$  respects the relation  $\equiv$  *on the proper elements only*. In other words,  $F$  is compatible iff  $\forall x, x' \ x \equiv x \wedge x \equiv x' \Rightarrow (F\ x) = (F\ x')$ . All the theorems that deal with  $F$  will have the hypothesis that all the elements of  $X$  are proper.

Alternative solutions to the PER approach (also called Partial Setoid Approach) is to first construct the subset of  $X$  as a sigma-type and then define an equivalence relation over the sigma-type elements. This alternative approach is strictly more expressive [1], but in practice it can be more cumbersome.

With respect to the previous scenarios the definition of substitutive context must not be changed. Moreover, the composition of substitutive functions (if the relation is symmetric) and the composition of strictly monotone functions (if the relation is asymmetric) preserves the property. However, the property is no longer granted for 0-ary functions (the constants that occur in the goal). An example should easily explain the remark. Consider the context  $(f\ x\ \square)$ . We know by hypothesis that  $f$  is substitutive in each argument, i.e.  $\forall x, x', y, y' \ x \equiv x' \wedge y \equiv y' \Rightarrow (f\ x\ y) \equiv (f\ x'\ y')$  and we want to prove that the whole context is substitutive. The proof is trivial provided that we can show that  $x$  is a proper element, i.e.  $x \equiv x$ . The latter proof cannot be fully automated. Thus we will leave it to the user.

## 3 Related Work

(Sub-)equational reasoning is closely related to Window Inferencing (WI) [6], a style of reasoning proposed by Robinson and Staples [10]. Like natural deduction and sequent calculus, WI is a way of forming and validating proofs in a given logic. A proof is built by manipulating a stack of windows. Each window is a

triple  $\Gamma, R, E$  where  $E$  is a formula in a context  $\Gamma$  and  $R$  is a binary relation over the type of  $E$ . The goal of the user is to transform the window  $\Gamma, R, E$  in a new window  $\Gamma, R, E'$  such that  $\Gamma \Vdash ERE'$ . The only actions the user can take are: 1) closing the window (equivalent to say that  $E' = E$ , requires  $ERE$ ); 2) opening a new window  $\Gamma', R', E'$  where  $E'$  is a subterm of  $E$  such that for all  $E''$ , if  $E'R'E''$  under the hypotheses  $\Gamma'$  then  $ERE\{E''/E'\}$  under the hypotheses  $\Gamma$ ; 3) transforming the window to  $\Gamma, R, E'$  in an atomic step (requires  $ERE'$ ). In the initial proposal for WI all relations  $R$  in the windows needed to be equivalence relations; however it was soon observed that symmetry is not required.

WI is surely an interesting style for proof development, especially for equational proofs, and it can be applied very naturally to some special domains such as program refinement. Moreover effective ad-hoc user interfaces have been developed for WI tools [4]. However, WI is a departure from more traditional proof styles such as natural deduction and sequent calculus and it is surely interesting to accomodate the basic ideas of WI in a more traditional setting.

In particular, we can consider a restriction of WI, equivalent to the original formulation, obtained by restricting the available actions to the sequence: 1) recursively open new windows as needed; 2) apply exactly one transformation; 3) recursively close all the opened windows. The sequence can be applied as many times as needed. Notice that this equivalent formulation does not require the relations to be transitive (since only one transformation can be applied before closing the windows): only the topmost relation must be transitive if the user wants to iterate the sequence more than once.

This alternative formulation can be better understood as a natural language deduction proof by *generalized rewriting*, where a term is substituted for an equivalent (but not necessarily congruent) one. The terminology “generalized rewriting” was introduced by Basin in [2]; in the rest of the paper we will adopt *equational reasoning*. To an application of the sequence corresponds the following proof by equational reasoning: the initial goal is  $\Gamma \vdash ER?$  where  $?$  is a metavariable to be instantiated during the proof search and  $E = C_1[\dots[C_n[E_{n+1}]]\dots]$  where  $E_i = C_i[\dots[C_n[E_{n+1}]]\dots]$  is the expression of the  $i$ -th opened window; in a context  $\Gamma_{n+1}$  it is possible to prove the lemma  $E_{n+1}R'E'_{n+1}$ ; the proof proceeds by *rewriting* the lemma, replacing  $E_{n+1}$  with  $E'_{n+1}$  obtaining the new goal  $\Gamma \vdash E'R?$  where  $E' = C_1[\dots[C_n[E'_{n+1}]]\dots]$  that is proved by reflexivity of  $R$  instantiating  $?$  with  $E'$ . To complete the proof a new goal  $\Gamma_{i+1} \Rightarrow E_{i+1}R_iE_{i+1}\{E_{n+1}/E'_{n+1}\}$ ;  $\Gamma_n \vdash E_iR_iE_i\{E_{n+1}/E'_{n+1}\}$  is opened for  $i = 1, \dots, n$  and it is solved (hopefully automatically) using the side condition for opening a new window in WI.

We can now see that a usual proof by rewriting is a special case of WI where all the relations are equivalence relations that are congruences w.r.t. the contexts  $C_i[\_]$ . In particular all the new goals generated by the rewriting step can be closed by using the fact that the appropriate relation is a congruence. Notice that, in this case,  $\Gamma_i = \Gamma$  for each window.

Finally we define a proof by *sub-equational reasoning* as the generalization of a proof by equational reasoning where the relations are not required to be

equivalence relations. Thus a proof by sub-equational reasoning is a special case of WI where each window opening does not change the context  $\Gamma$ , i.e. where  $\Gamma_i = \Gamma$  for each window.

**Observation 1.** *To summarise the requirements over the relations in WI (the restricted formulation), every relation but that of the initial window does not need to be either symmetric or transitive. Reflexivity is also not required, but it is often necessary to prove the side conditions for opening a window. Thus the implementations of WI usually require the relations to be reflexive (or even pre-orders) [11].*

**Observation 2.** *(Sub-)equational reasoning is strictly less expressive than WI because of the restriction on the contexts. For instance, when proving  $\Gamma \vdash (A \wedge B \rightarrow C) \iff ?$  in classical propositional calculus using WI it is possible to open a new window at  $A$  adding to  $\Gamma$  the two new hypotheses  $\neg C$  and  $B$  and then exploiting them to prove  $A \iff E$  for concluding  $\Gamma \vdash (A \wedge B \rightarrow C) \iff (E \wedge B \rightarrow C)$ . To prove the same result using (sub-)equational reasoning only the user needs first of all to prove the lemma  $A \iff E$  in the context  $\Gamma$ , without the two additional hypotheses  $\neg C$  and  $B$ .*

Although (sub-)equational reasoning is less expressive than WI, it is however a very useful generalization of rewriting and it has been exploited for years in the NuPRL system [2, 8] under the name of generalized rewriting. In this paper we present a tactic for (sub-)equational reasoning that is functionally equivalent to the tactic proposed in NuPRL. However, it differs from the NuPRL tactic in a few important aspects:

1) The NuPRL tactic is completely implemented at the meta-level. The meta level is responsible for finding a proof that the rewriting context is substitutive and generating the corresponding proof term. The latter is huge, its size being at least quadratic in the size of the initial expression. The well known problem, already existent for proofs based on simple rewriting, is that each rewriting step requires an explicit term that is the predicate obtained by abstracting the goal over the term to replace, the term to be replaced and the term to be substituted. Our implementation solves the problem by adopting a semi-reflexive approach. The meta level is still responsible for finding a proof that the rewrite context is substitutive, but instead of generating a proof term that details every step of the proof it generates a proof term that applies a correctness theorem to a trace of the proof found. The trace is roughly linear in the size of the initial expression, being an annotation of it (see Sect. 5 and 6). The correctness theorem says that from each well typed trace there exists a proof of the statement.

The semi-reflexive approach does not hinder flexibility in proof search: since the proof search is done at the meta level it is easy to implement backtracking and heuristics, and it is easy to invoke already existent tactics for proving side conditions. A completely reflexive approach (i.e. implementing proof search in the logic) would be much more difficult to maintain and less computationally efficient. Moreover, the proof terms generated by (semi-)reflexive tactics are not only more compact, but they also have additional advantages for proof rendering purposes.

2) The tactic for NuPRL described by Basin [2] does not search the whole space for possible proofs of substitutivity, using an heuristic to avoid backtracking but possibly failing to find a proof. The motivation given for the heuristic is the complexity of the complete method, that is exponential in the depth of the rewritten subterms. However, since in practice the goal is small and not very deep, the exponential complexity is not really a problem and we decided to implement the complete method.

A stronger motivation for a simpler implementation is surely that the proof search is quite a complex operation that can be extremely difficult to debug (since a bug results in a complex not well typed proof term). This motivation becomes much less critical in our approach, since the proof term generated is just a trace of the proof — an annotation of the term to be rewritten — that can be inspected much more easily.

3) Finally in our theory we introduce the notion of *variance* for the arguments of a substitutive function. This notion is missing in the WI approach since it can be simulated by adding to the library of the system the inverse relation  $R^{-1}$  for each relation  $R$ . However, we argue that our approach is somehow more natural since it fits better with the notion of monotone functions we are going to develop and for sure it avoids augmenting the library with redundant relations that should later on be considered equivalent during unification, proof search, etc. Just to make a simple example, our approach avoids 1) having to introduce reverse implication  $A \Leftarrow B$  in the system; 2) proposing to the user confusing goals that mix the two implications, e.g.  $\neg A \Leftarrow \neg B \Rightarrow B \Leftarrow A$ ; 3) introducing in the system automatic conversions to normalise the goals by expanding every definition of  $\Leftarrow$  in a goal during unification and proof search.

## 4 Morphisms and Signatures

The scenarios provided in Sect. 2 should have convinced the reader that replacing one side of a relation with the other in a goal is a natural operation even when the relation is not an equivalence. To perform the replacement the system has to show that the replacement context is substitutive. This can easily be automated. However, we have seen that according to the properties of the relation the steps required to prove substitutivity slightly change. Thus the implementation of the tactic can become quite tricky in practice. In this section we will present a small theory of monotone functions that will be the basis for a semi-reflexive tactic implementation described in Sect. 5.

### 4.1 Basic Definitions

We characterise  $n$ -ary functions that are simultaneously monotone in each argument not only by the types of their arguments and by their output type, but also by the relations associated to each type and by flags that state whether a function is increasing or decreasing in each argument:



**Definition 1 (Signature).** A signature is an expression

$$R_1 \Rightarrow^{d_1} \dots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

where  $(T_1, R_1), \dots, (T_n, R_n), (T, R)$  are pairs whose first component is a type and whose second component is a binary relation<sup>1</sup> over it and  $d_1, \dots, d_n \in \{\triangleright, \triangleleft\}$ .

Notice that each relation  $R$  in a signature uniquely identifies its carrier  $T$ .

**Definition 2.** Let  $R$  be a binary relation.  $R \triangleright \stackrel{\text{def}}{=} R$  and  $R \triangleleft \stackrel{\text{def}}{=} R^{-1}$  (i.e.  $R \triangleleft \stackrel{\text{def}}{=} \lambda xy.yRx$ )

**Definition 3 (Morphism).** A morphism of signature

$$R_1 \Rightarrow^{d_1} \dots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

is a function

$$f : T_1 \rightarrow \dots \rightarrow T_n \rightarrow T$$

that is simultaneously monotone in its arguments:

$$\forall x_1 x'_1, x_1 R_1^{d_1} x'_1 \rightarrow \dots \rightarrow \forall x_n x'_n, x_n R_n^{d_n} x'_n \rightarrow (f x_1 \dots x_n)R(f x'_1 \dots x'_n)$$

To state that a function  $f$  is a morphism of signature  $\Sigma$  we will write  $f : \Sigma$ .

**Definition 4 (Covariance and contravariance).** Let

$$f : R_1 \Rightarrow^{d_1} \dots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

We say that  $f$  is covariant (respectively contravariant) in the  $i$ -th argument if  $d_i$  is equal to  $\triangleright$  (respectively  $\triangleleft$ ).

**Fact 1.** If the  $i$ -th relation  $R_i$  of a signature  $\Sigma$  is symmetric, then every morphism of signature  $\Sigma$  has also signature  $\Sigma'$  where  $\Sigma'$  is obtained from  $\Sigma$  by changing the variance of the  $i$ -th argument.

## 4.2 Morphism Properties

Morphisms can be composed:

**Fact 2 (Morphism composition).** Given

$$f : R_1 \Rightarrow^{d_1} \dots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

and

$$f_i : R_1^i \Rightarrow^{d_1^i} \dots \Rightarrow^{d_{n_i-1}^i} R_{n_i} \Rightarrow^{d_{n_i}^i} R_i \text{ for } i = 1, \dots, n$$

<sup>1</sup> We do not require the relations to be orders, even if we still adopt the corresponding terminology talking of monotone functions.

the compound function

$$f \circ \langle f_1, \dots, f_n \rangle$$

is a morphism of signature

$$R_1^1 \Rightarrow^{d_1^1=d_1} \dots \Rightarrow^{d_{n_1-1}^1=d_1} R_{n_1}^1 \Rightarrow^{d_{n_1}^1=d_1} \\ \dots \\ R_1^n \Rightarrow^{d_1^n=d_n} \dots \Rightarrow^{d_{n_n-1}^n=d_n} R_{n_n}^n \Rightarrow^{d_{n_n}^n=d_n} R$$

where  $d = d'$  is  $\triangleright$  when  $d$  and  $d'$  are the same variance, and  $\triangleleft$  when they are not.

Let  $R$  be a binary relation over  $T$ . The following facts hold.

**Fact 3 (Identity morphism).** *The identity function  $\lambda x : T.x$  is a morphism of signature  $R \Rightarrow^\triangleright R$ .*

**Fact 4 (0-ary morphism).** *The 0-ary constant function  $t$  of type  $T$  is a morphism of signature  $R$  if and only if  $tRt$ .*

**Fact 5.** *If  $R$  is a reflexive relation than every 0-ary function of type  $T$  is a morphism of signature  $R$ .*

Notice that if  $(T, R)$  is a partial setoid, then the 0-ary morphisms  $T$  are the proper elements of the setoid.

**Fact 6 (Contraction (basic case)).** *Given a morphism*

$$f : R \Rightarrow^\triangleright \dots \Rightarrow^\triangleright R \Rightarrow^\triangleleft R \Rightarrow^\triangleleft \dots R \Rightarrow^\triangleleft R'$$

the function  $\lambda xy.(f \ x \ \dots \ x \ y \ \dots \ y)$  is a morphism of signature

$$R \Rightarrow^\triangleright R \Rightarrow^\triangleleft R'$$

The generalisation to an arbitrary interleaving of the covariant and contravariant arguments are left as a trivial exercise to the reader.

In the general case using contraction we can always obtain a new morphism equivalent to a given one such that each relation occurs in the signature at most twice, once in covariant and the other in contravariant position.

As a first step towards a semi-reflexive implementation of our tactic in the Coq system we have formalised in Coq the definition of signature and (an equivalent version of) all the previous facts. To obtain a non-reflexive implementation of the tactic such formalisation is not required.

In the formalisation we have heavily exploited dependent types and internal computation to make the formalisation simpler and more elegant. For example, it is possible to define in Coq two functions `signature_to_type : signature → Type` and `signature_to_compatibility : Π Σ : signature, signature_to_type Σ → Prop` that, given a signature

$$R_1 \Rightarrow^{d_1} \dots \Rightarrow^{d_{n-1}} R_n \Rightarrow^{d_n} R$$

compute respectively the function type

$$f : T_1 \rightarrow \dots \rightarrow T_n \rightarrow T$$

and the proposition

$$\lambda f : T_1 \rightarrow \dots \rightarrow T_n \rightarrow T, \\ \forall x_1 x'_1, x_1 R_1^{d_1} x'_1 \rightarrow \dots \rightarrow \forall x_n x'_n, x_n R_n^{d_n} x'_n \rightarrow (f x_1 \dots x_n) R (f x'_1 \dots x'_n)$$

Another example consists in the function that automatically computes the signature of a morphism composition from the signatures of the arguments and the output type (and relation and variance) of the head function. In this way we reduce the problem of checking whether morphisms have a given signature or whether morphism can be composed to a type-checking problem, that is automatically decided by Coq. Moreover, we obtain proof terms that are relatively small, since we often provide just the signatures in place of long lists of constraints that must be satisfied for instance to compose morphisms.

The current formalisation in Coq is about 664 lines long and required just a few days, despite the fact that Coq automation vis a vis of dependent types (e.g. generation of elimination predicates and unification) is often unsatisfactory.

## 5 The Tactic

Let us focus now on the implementation of the tactic. We define an *applicative context* to be a term generated by the following grammar:

$$t ::= \square \mid x \mid (f t \dots t)$$

An applicative context is a term with a non-linear placeholder  $\square$  that is a syntactic composition of constants applications. The constants  $f$  can be  $n$ -ary or 0-ary. Unbound variables  $x$  can also occur in the applicative context (but they can not be applied to arguments).

We restrict ourselves to those applicative contexts  $C$  such that:

- all the constants have first order non-dependent types
- the constants are fully applied
- the term  $\lambda x. C\{x/\square\}$  is well-typed

Since we are avoiding dependent types we can also deal with function spaces and implications: a formula  $A \rightarrow B$  can be seen as an applicative context ( $\text{Impl } A B$ ) where  $\text{Impl}$  is defined as  $\lambda AB. A \rightarrow B$ .

We suppose that there exists a way for the user to specify for each constant  $f$  a set of signatures  $\Sigma_1, \dots, \Sigma_n$  such that  $f$  is a morphism of signature  $\Sigma_i$  for each  $i$ . Of course the user must provide a proof  $p_i$  of the fact that  $f$  is a signature of type  $\Sigma_i$ . Notice that the set of signatures is never empty, since every constant  $f$  is at least a morphism of signature  $\Sigma_{\mathcal{L}}$  where  $\Sigma_{\mathcal{L}}$  is the signature  $\Rightarrow^{\triangleright} \dots \Rightarrow^{\triangleright} \Rightarrow^{\triangleright} =$  where  $=$  is Leibniz equality. Fact 1 can be used to reduce the number of registered signatures: if a morphism is both covariant and contravariant in an argument only one of the two signatures must be declared.

We also suppose that there exists a way for the user to register for each type a set of binary relations over it.

We can now describe the specification of our tactic:

**Input** *A well-behaved propositional applicative context  $C$ , a relation  $R$  over  $T$  and a rewrite direction  $d \in \{\triangleright, \triangleleft\}$ .*

**Output** *A proof of  $\lambda x : T.C\{x/\square\} : R \Rightarrow^d \text{Impl}$  where  $\text{Impl}$  is propositional implication. The tactic fails if it is not able to produce such a proof.*

Before proceeding in describing the implementation, let us see how the tactic is concretely used. Facing the goal

$$\frac{H : E_1 R E_2}{G}$$

the user can ask to replace either  $E_1$  with  $E_2$  or  $E_2$  with  $E_1$  in  $G$ . In the first case our tactic is invoked with the arguments  $C = G\{\square/E_1\}$ ,  $R$  and  $\triangleleft$ ; in the second case with the arguments  $C = G\{\square/E_2\}$ ,  $R$  and  $\triangleright$ . In both cases the proof built by the tactic can be applied to the hypothesis  $H$  to obtain the new goal  $G\{E_2/E_1\}$  (respectively  $G\{E_1/E_2\}$ ).

We describe now a naive implementation of the tactic based on the facts given in the previous section. We will then sketch the more efficient implementation that is now integrated in the Coq development release.

**Step 1:** the first steps output a set of applicative contexts annotated in each possible way according to the following rules:

- annotate each constant with one of its declared signatures; use fact 1 and the proof of symmetry of the appropriate relation to generate every correct signature w.r.t. variance constraints together with the proof that the constant respects the signature
- annotate each placeholder with the identity signature  $R \Rightarrow^{\triangleright} R$ . This corresponds to interpreting a placeholder as the identity morphism (cfr. fact 3)
- annotate each variable with the appropriate signature for a 0-ary constant (cfr. fact 4); generate one annotation for each relation declared over the type of the variable; prove that the variable is a 0-ary morphism by using fact 5 (if the relation is reflexive) or by asking the user to prove that the variable is a proper element for the relation

An annotated applicative context is a syntactic description of a (possibly wrongly typed) hereditary morphism composition where each placeholder is interpreted as the identity morphism (cfr. fact 3).

**Step 2:** prune out from the set those morphism compositions that are not well-typed according to the constraints given in fact 2 (i.e. the relation associated to each formal argument must match the output relation the actual argument is annotated with).

**Step 3:** prune out the morphisms whose conclusion is not the relation  $\text{Impl}$  (propositional implication).

- Step 4:** compute the signature of the obtained morphisms according to the rules given in fact 2.
- Step 5:** apply contraction to each morphism in the set (fact 6); filter out those morphisms whose signature is not  $R \Rightarrow^d \text{Impl}$
- Step 6:** if the set is empty than fail; otherwise pick one morphism from the set and build a proof of the fact that the morphism has the expected signature by visiting the annotated applicative context and applying the appropriate fact for each node.

The previous implementation is obviously highly inefficient, since it builds a huge search space first and then prunes it. Better implementations can be obtained by constructing and pruning the search space at the same time. This is roughly the strategy we implemented in the Coq system<sup>2</sup>:

```
let rec decorate C R' d' =
  match C with
  [] -> if d=d' && R = R' then ok else error
  |x -> if type x = carrier R' && (reflexive R' || ask-user R' x)
        then ok else error
  | (f C) ->
    let sigset = signatures-of f in
    let sigset' = filter (fun S -> output S = R') sigset in
    exists sigset' (fun S ->
      decorate C (relation (input S))
      (compose-dir R' (variance (input S))))
```

The function `decorate` must be initially called on the context  $C$ , relation `Impl` and direction  $\triangleleft$ . Then it proceeds recursively over the applicative context remembering what is the expected output relation of the morphism  $C$  and whether the morphism  $C$  is supposed to be increasing or decreasing. The function `exists` performs the necessary backtracking over the set of signatures that have the expected output type. When a placeholder is met it is checked whether the expected relation and variance are coherent with the one the user is interested in. The function `ask-user` is supposed to open a new goal to ask the user to prove that its second argument is proper w.r.t. the relation that is the first argument.

We decided to implement the tactic in the Coq system in a semi-reflexive way (i.e. proof search is implemented at the meta-level; a trace of the proof found is reified at the term level and it is used by the type-checker of Coq to reconstruct the whole proof as a typing problem). The proof-search part (roughly the strategy described by the function `decorate`) is implemented in OCaml and it returns a decorated applicative context. Then the decorated applicative context is reified in a Coq term. Finally a Coq function checks that the decorated

<sup>2</sup> To keep the code short only the case of unary morphisms is shown. The  $n$ -ary case is more complicated due to the need of backtracking the computation on all the arguments of the application whenever one branch fail. The actual code must also compute the well-typed annotated applicative context to produce the final proof. It is written in OCaml and it is 229 lines long.

applicative context is well-typed, interprets it as a morphism and produces a proof that the morphism respects its signature. In Sect. 6 we present an example of usage of the tactic and we show the decorated applicative context and the proof term generated for that example.

Thanks to dependent types and internal computation in the Coq logic, checking that the decorated applicative context is well-typed boils down to a checking problem that is decided by the system. The construction of the proof is also done on-the-fly by the Coq system during type-checking. Thus the actual proof term generated by the tactic is just the application of a constant to the term that describes the decorated applicative context, that is linear in the size of the goal. If the tactic was written entirely in OCaml, instead, the generated proof goal would have been much bigger and definitely non linear in the size of the goal.

Implementing the tactic as a fully reflexive one would have meant implementing also the proof-search part in Coq. Since execution of code in Coq is made by an interpreter while OCaml code is compiled, this would have meant a much slower tactic.<sup>3</sup> Moreover, it is simpler to add heuristics and optimisations to the OCaml code without having to prove all of them correct: the Coq part of the tactic is responsible of verifying the correctness of the generated solutions and helps in detecting bugs in the OCaml code. Finally, when a 0-ary morphism over a non reflexive relation is found, our tactic needs to open a new goal. To the best of our knowledge, it is impossible for a fully reflexive implementation to open a new goal, whereas this is easy in the semi-reflexive setting: it is sufficient to reify a trace that contains fresh metavariables.

## 6 An Example

The following example shows a situation where the tactic really shines.

Let  $=^-$  be the smallest reflexive relation over negative integer numbers (i.e.  $x =^- y$  iff  $x = y$  and  $x < 0$ ) and  $=^+$  be the smallest reflexive relation over positive integer numbers. Suppose that the user has already proved and registered the following signatures for less than, negation and multiplication:

signature:	corresponding compatibility statement:
$< : \leq \Rightarrow \triangleleft \leq \Rightarrow \triangleright$	<b>Impl</b> $\forall x, x', y, y' \ x' < x \wedge y < y' \Rightarrow (x < y \Rightarrow x' < y')$
$- : \leq \Rightarrow \triangleleft <$	$\forall x, x' \ x' < x \Rightarrow -x < -x'$
$* : \leq \Rightarrow \triangleleft =^- \Rightarrow \triangleright <$	$\forall x, x', y, y' \ x =^- x' \wedge y' < y \Rightarrow x * y < x * y'$
$* : \leq \Rightarrow \triangleright =^+ \Rightarrow \triangleright <$	$\forall x, x', y, y' \ x =^+ x' \wedge y < y' \Rightarrow x * y < x * y'$

Notice how the compatibility statements are usual preliminary statements found in every arithmetic library about integer numbers, up to the strange notation  $x =^- y$  that must be unfolded to obtain  $x < 0$  and  $x = y$ .

<sup>3</sup> The next version of the Coq system will be equipped with a compiler [5] that should allow to implement complex proof search procedures as reflexive tactics.

Notice also that the transitivity of  $< -$  that is never used by the tactic — can be exploited to prove that  $<$  is a morphism of signature  $< \Rightarrow^{\triangleleft} < \Rightarrow^{\triangleright} \text{Impl}$ . Indeed this is automated in the system for each relation declared to be transitive.

Now consider the goal

$$\frac{n, m, c_1, c_2 : Z \quad H : c_1 < c_2}{c_1 * n * m < -c_1 * n * m}$$

If the user asks to replace  $c_1$  with  $c_2$  by hypothesis  $H$  the tactic is automatically able to generate the new goal

$$\frac{n, m, c_1, c_2 : Z \quad H : c_1 < c_2}{c_2 * n * m < -c_2 * n * m}$$

asking the user to prove either the first two following side conditions (i.e. both  $n$  and  $m$  are negative numbers) or the last two (i.e. both  $n$  and  $m$  are positive numbers):

$$\frac{n, m, c_1, c_2 : Z}{n =^- n} \quad \frac{n, m, c_1, c_2 : Z}{m =^- m} \quad \frac{n, m, c_1, c_2 : Z}{n =^+ n} \quad \frac{n, m, c_1, c_2 : Z}{m =^+ m}$$

Suppose that the user chooses the first possibility (both  $n$  and  $m$  negative numbers). Then the part of the tactic implemented at the meta level — that implements the annotation function `declare` in Sect. 5 — produces the following annotated applicative context  $\mathcal{C}$ :

$$\mathcal{C} \stackrel{def}{=} (\leq (\underline{*} (\underline{*} \square^{<\triangleright} n_{?_1}^{=-\triangleleft})^{<\triangleleft} m_{?_2}^{=-\triangleright})^{<\triangleright} (\underline{*} (\underline{*} (- \square^{<\triangleright})^{<\triangleleft} n_{?_1}^{=-\triangleright})^{<\triangleright} m_{?_2}^{=-\triangleleft})^{<\triangleleft}) \text{Impl}^{\triangleleft}$$

that is just an annotation of  $(< (* (* \square) n) m) (* (* (- \square) n) m)$  (the applicative context in prefix form obtained by abstracting the goal over  $c_1$ ). The semantics of the annotations is described in Table 1. The annotation  $\square^{<\triangleright}$  says that  $\square$  is a monotone increasing context (w.r.t.  $<$ ), according to the user wish to replace the left hand side of the hypothesis  $H$  with its right hand side; thus  $(* \square^{<\triangleright} n^{=-\triangleleft})^{<\triangleleft}$  says that  $(* \square n)$  is a monotone decreasing context (since  $n =^- n$ ); thus  $(* (* \square n)^{<\triangleleft} m^{=-\triangleright})^{<\triangleright}$  says that  $(* (* \square n) m)$  is a monotone increasing context (since  $m =^- m$ ); and so on until we find that the whole applicative context is monotone decreasing (w.r.t. logical implication).

The annotated context is reified as a CIC term and given as an argument to the correctness theorem, obtaining the following proof term  $t$ :

$$t \stackrel{def}{=} (\text{generalised\_rewrite\_ok } Z \ <^{\triangleleft} \ \mathcal{C} \ c_1 \ c_2 \ H)$$

Since the annotated context  $\mathcal{C}$  is linear in the size of the goal, the proof term  $t$  is linear in the size of the goal plus the size of the type of the term to be rewritten

**Table 1.** Annotated applicative contexts

$\underline{f}$	every function $f$ is annotated with (a reference to) the proof of its compatibility statement; i.e. $\underline{f} = (f, p)$ for some previously user-provided proof term $p$
$\square^{R^d}$	each placeholder is annotated with its relation and the rewrite direction chosen by the user
$x_{?_i}^{R^d}$	each constant is annotated with its relation $R^d$ and a proof that $xRx$ ; a metavariable $?_i$ stands for the proof of a new goal that will be proposed to the user
$(\underline{f} C_1^{R_1^{d_1}} \dots C_n^{R_n^{d_n}})^{R^d}$	every application is annotated with its relation $R$ and a rewrite direction and it is well typed iff the relations and rewrite directions of its arguments and of the whole application are coherent with the signature of $f$ (that must be $R_1 \Rightarrow^{d_1=d} \dots R_n \Rightarrow^{d_n=d} R$ )

(i.e.  $Z$ ). The correctness theorem `generalized_rewrite_ok` has type

$$\begin{aligned} \text{III} : \text{Type.II} R : T \rightarrow T \rightarrow \text{Prop.II} d \text{IIC} : (\text{reified\_annotated\_context } T \ R \ d). \\ \text{II} c_1 c_2 : T. \text{II} H : c_1 \ R^d \ c_2. \\ (\text{signature\_to\_compatibility } (R \Rightarrow^d \text{Impl})(\text{deannotate\_context } T \ R \ d \ C)) \end{aligned}$$

where `reified_annotated_context` is the concrete data type for reified annotated contexts of the specified signature (i.e. an inductive type with one constructor for each line of Table 1) and `deannotate_context` returns the (deannotated) Coq term described by a reified annotated context. The kernel of Coq will type-check the reified annotated context and the proof term and it will infer for  $t$  the type

$$\forall c_1, c_2 : Z. c_1 < c_2 \Rightarrow c_2 * n * m < -c_2 * n * m \Rightarrow c_1 * n * m < -c_1 * n * m$$

that asserts that the current goal is substitutive for  $c_1$ . Notice also that since  $t$  is not a closed term (because the two meta-variables  $?_1$  and  $?_2$  occurs in  $C$ ) Coq will accept the proof under the condition that the user will later on instantiate the two metavariables (i.e. she will close the two new sub-goals).

## 7 Conclusions

We described a small theory of monotone functions closed under composition. The theory can be used to study the properties of applicative contexts – a syntactically restricted term with non-linear occurrences of a placeholder. It leads to the implementation of a tactic that can semi-decide whether a given applicative context is a monotone function when interpreted as a function over the argument marked with a placeholder. The tactic exploits predefined proofs of monotonicity for each constant that are user-provided. The tactic can be applied



to the problem of replacing one side of an hypothesis  $E_1 RE_2$  with the other side in the current goal, where  $R$  is a general relation. The stronger the properties of the relations considered (i.e. reflexivity and symmetry), the fewer the user provided proofs required by the tactic.

The tactic has been implemented as a semi-reflexive one in the Coq proof assistant. It can be applied in several useful situations, such as rewriting in total and partial setoid theories; contraction or expansion of terms according to given reduction relation; propagation of large or strict arithmetical inequalities.

The actual status of the tactic is currently satisfactory. It has been used so far in a few test cases about reductions, greatly reducing the size of the proof script. As a tactic that supports the setoid approach both Lionel Mamane and Bas Spitters have used it in yet unpublished works, respectively about the formalization of surreal numbers and for handling implicit and non-computational arguments using monads. It is also being used by Marco Maggesi in the development of a theory of categories based on the partial setoid approach. Whether it can really be useful in the development of arithmetical libraries (using the approach given in Sect. 6) is still to be understood and currently under investigation.

The current implementation as a semi-reflexive tactic is also satisfactory due to the reasonably small size of the generated proof term. However, the tactic can be fully implemented in any other system for Higher Order Logic using the meta-language only. A fully reflexive implementation of the whole tactic seems difficult because of the need of opening new goals when the relations considered are not reflexive.

The only extension of the tactic and of the underlying theory that seems to be worth considering is that of *subrelations*. A relation  $R$  is a subrelation of a relation  $R'$  whenever  $xRy$  implies  $xR'y$ . Subrelations can be used as coercions by the “type-checking” algorithm for signatures. The interesting case is that of a morphism whose signature output is  $R$  found as an actual argument of a function that expects an argument whose output is  $R'$ . Instead of failing, the checker could just verify if  $R$  is registered as a subrelation of  $R'$ , using the witness of the subrelation to build the proof of compatibility for the morphism.

In theory the declaration of subrelations does not add any expressive power to the tactic. In practice it could highly reduce the number of signatures to declare for a given constant. For instance, the “less than” and the equality relations can be both defined as subrelation of “less than or equal”. Subrelations are implemented in the NuPRL system [8, 2].

To conclude, the proposed tactic can be an important building block for more complex tactics. Iterating a one step rewriting tactic and backtracking in case of failure yields a powerful simplification tactic for the Coq system, named *autorewrite*. Since *autorewrite* uses the rewriting tactic as a black box, it was easy to make *autorewrite* use our new tactic for performing rewriting of hypotheses that are not equalities. As an application, *autorewrite* is now able to reduce lambda-terms to their normal forms in a development of the theory of residuals for the lambda-calculus that we have used as a test case for the tactic. Once again, the reduction of the proof script size can be remarkable.

## References

1. Gilles Barthe, Olivier Pons and Venanzio Capretta. “Setoids in type theory”. In *Journal of Functional Programming*, 13(2), 261–293, 2003.
2. David Basin. “Generalized Rewriting in Type Theory”. In *Journal of Information Processing and Cybernetics*, 30(5/6), 249–259, 1994.
3. Pierre Courtieu. “Normalized Types”. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic*, 554–569, 2001.
4. Christoph Lüth and Burkhart Wolff. “TAS — A Generic Window Inference System”. In *Theorem Proving in Higher Order Logics: 13th International Conference*, LNCS 1869, 405–422, 2000.
5. Benjamin Grégoire. *Compilation de termes de preuves: un (nouveau) mariage entre Coq et OCaml*. PhD thesis. Université Paris 7. 2004.
6. Jim Grundy. “Transformational hierarchical reasoning”. In *The Computer Journal*, 39(4):291–302, 1996.
7. Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis. LFCS Edinburgh, 1995.
8. Christoph Kreitz. “The NuPRL 5 Manual”. 135–145.
9. Clément Renard. *Memoire de DEA : Un peu d’extensionnalité en Coq*. INRIA Rocquencourt, 2001.
10. P. J. Robinson and J. Staples. “Formalizing a hierarchical structure of practical mathematical reasoning”. *Journal of Logic and Computation*, 3, 47–61, 1993.
11. Mark Staples. “Window Inference in Isabelle”. In *Proceedings of the Isabelle Users Workshop*, 18–19 September 1995, Cambridge (UK).

# A Uniform and Certified Approach for Two Static Analyses

Solange Coupet-Grimal and William Delobel

Laboratoire d'Informatique Fondamentale de Marseille (UMR 6166),  
CMI-Université de Provence, 39 rue Joliot-Curie, F-13453, Marseille, France

**Abstract.** We give a formal model for a first order functional language to be executed on a stack machine and for a bytecode verifier that performs two kinds of static verifications : a type analysis and a shape analysis, that are part of a system used to ensure resource bounds. Both are instances of a general data flow analyzer due to Kildall. The generic algorithm and both of its instances are certified with the Coq proof assistant.

## 1 Introduction

Over the last decade, research on mobile code has been a hot topic and intensive efforts have been made to reduce the risk of malicious (Java) applets performing a security attack. For this, a crucial functionality of the Java Platform is the bytecode verifier which performs a static type analysis on programs. This kind of analysis ensures integrity properties of the execution environment such as the absence of memory faults. Consequently, there has been considerable interest in specifying formally the Java Virtual Machine and proving the correctness of its bytecode verifier (see for instance [4, 5, 8, 10, 11, 13] ...).

More recently, these investigations have been extended to establishing an additional property that contributes to guarantee the safety of bytecode by ensuring bounds on the computational resources needed by its execution. Within this context, a project has been undertaken [2] which focuses on a rather standard first-order functional programming language with inductive types, pattern matching, and call-by-value, to be executed on a simple stack machine. The language comes with various bytecode static analyses: a standard type analysis, an analysis on the algebraic shape of the values in the stack, an analysis of the size of these values, and an analysis that insures the termination. The last three analyses, and in particular their combination, are instrumental in predicting the space and time required for the execution of a program.

This paper deals with the formal specification of the virtual machine (VM) related to this language and the certification in the Coq proof assistant of an extended bytecode verifier which performs the first two phases of the analysis, that is the type and shape verifications. Our contribution is threefold. First, we present a verifier designed in a uniform way, so as both verification processes become special cases of a general framework for data flow analysis based on

the well known algorithm due to Kildall [9]. Second, we propose a functional specification in Coq of Kildall's algorithm and we prove its correctness. This is related to Klein and Nipkow's work with the system Isabelle/HOL [13, 10]. However our formalization differs from theirs by the heavy use of Coq dependent types and the way of encoding a recursive function which is not structurally recursive. With this approach, properties common to both analyses are established once and for all. Note that this approach also suits size analysis, although this remains work to be done in future. Third, these generic properties are used, for proving not only the correctness of type verification, but also, and this is the novelty, that of algebraic shape verification.

The paper is organized as follows. The first section is a quick description of the problem: we present the language, the bytecode instructions, and both kinds of analyses. Section 3 is dedicated to encoding Kildall's algorithm in Coq and proving its correctness. Section 4 is related to both type and shape analyses which are formally specified and proved correct. In section 5 we make a comparison with related work and give a conclusion. Some proofs are detailed in appendix.

## 2 The Functional Language and the Bytecode Instructions

A formal and rigorous description of the source language, the bytecode instructions, their operational semantics, and of the type and shape analyses can be found in [2]. For lack of space, we have chosen to give here an informal but intuitive presentation that we illustrate by examples to give a quick understanding of the problem. The source language is a first order functional language, with (mutually) inductive types. Functions are defined by a sequence of pattern matching rules of the form  $f(p_1, \dots, p_n) = e$ , where  $e$  is an expression and  $p_1, \dots, p_n$  are

```

type bool = T | F ;;
type nat  = Z | S of nat ;;
type env  = Nil | C of nat * env ;;
type form =
  Var of nat | Not of form | Or of form * form | Ex of nat * form ;;

not (T) = F    or (T,y) = T    eq (Z,Z)      = T          eq (S(x),Z) = F
not (F) = T    or (F,T) = T    eq (S(x),S(y)) = eq (x,y)   eq (Z,S(x)) = F
                or (F,F) = F

member (x,Nil)      = F
member (x,C(y,l)) = or(eq(x,y),member(x,l))

check (Var(x), l)   = member (x,l)          qbf (f) = check(f,Nil)
check (Not(f1), l) = not (check(f1,l))
check (Or(f1,f2), l) = or (check(f1,l),check(f2,l))
check (Ex(x,f1), l) = or (check(f1,l),check(f1, C(x,l)))

```

Fig. 1. A program for evaluating boolean formulae

member:			
0: load(1);		[1 x]	Id
1: branch("Nil",4);		[1 1 x]	Id
2: build("F",0);		[Nil x]	l=Nil
3: return;		[F Nil x]	l=Nil
4: branch("C",13);		[1 1 x]	Id
5: load(0);		[t h C(h,t) x]	l=C(h,t)
6: load(2);		[x t h C(h,t) x]	l=C(h,t)
7: call("eq",2);	[h	x t h C(h,t) x]	l=C(h,t)
8: load(0);		[eq(x,h) t h C(h,t) x]	l=C(h,t)
9: load(3);	[x	eq(x,h) t h C(h,t) x]	l=C(h,t)
10: call("member",2);	[t	x eq(x,h) t h C(h,t) x]	l=C(h,t)
11: call("or",2);	[member(x,t)	eq(x,h) t h C(h,t) x]	l=C(h,t)
12: return;	[or(eq(x,h),member(x,t))	t h C(h,t) x]	l=C(h,t)
13: stop;		[1 1 x]	Id
14: return;	⊥		

Fig. 2. Symbolic execution of function *member*

linear patterns (a variable occurs at most once). As an illustration, Fig.1 displays a program that evaluates boolean expressions. Each function is compiled to bytecode to be executed by a VM. At run time, a frame ( $f$ ,  $pc$ ,  $P$ ) is created at each function call:  $f$  is the function's name,  $pc$  is the program counter that indicates the index of the current instruction (initially 0), and  $P$  is a stack of values that is initialized by the arguments of the function. This frame is pushed on the top of the current configuration, that is the stack constituted by all the frames of the functions currently active. Fig.2 shows a possible bytecode program for function *member* in Fig.1, as well as a symbolic execution of the function on natural number  $x$  and environment  $l$ . Each line shows the value  $pc$  of the program counter, the related instruction, and the expression stack on which the instruction of index  $pc$  is executed (the top of the stack is on the left). Let us describe the operational semantics of the instructions that appear in this piece of bytecode.

- Instruction  $load(j)$  pushes on top of  $P$  the element of index  $j$ , counting from the bottom of the stack.  $pc$  is incremented. Note that the bottom of  $P$  is index 0.
- Instruction  $branch(c, j)$  matches element  $e$  on the top of the stack with constructor  $c$  of arity  $m$ . If the matching succeeds,  $e$  is popped out from the stack, it is deconstructed and its  $m$  arguments are pushed on the stack.  $pc$  is incremented. If the matching fails, the stack is left unchanged and  $pc$  is set to  $j$ .
- Instruction  $build(c, m)$ , where  $c$  is a constructor of arity  $m$ , discards the  $m$  values  $v_1, \dots, v_m$  on the top of  $P$  and pushes  $c(v_1, \dots, v_m)$ . Program counter  $pc$  is incremented.

- Instruction *return* returns to the environment the result on the top of the stack. The current function is deactivated, that is its frame is popped out from the current configuration.
- Instruction *stop* stops the execution.
- Instruction *call*( $g, m$ ) pushes, over the frame of the calling function, a new frame for executing function  $g$  of arity  $m$ . In this new frame, the program counter is set to 0 and the value stack is constituted by  $m$  values, popped out from the value stack of the calling function.

**Shape Analysis as a Symbolic Execution.** Fig.2 is nothing but the result of a symbolic execution of function *member* in the sense that the stack contains algebraic expressions, built with constructors, functions names and variables, instead of values built from constructors only (as in a true execution). In particular, the arguments of the function have been replaced by two variables  $x$  and  $l$ . Within this context of symbolic execution, each instruction *branch*( $c, j$ ) gives rise to a substitution that matches the expression on the top of the stack with a pattern  $c(x_1, \dots, x_m)$ . Here,  $m$  is the arity of constructor  $c$  and  $x_1, \dots, x_m$  are fresh variables. Substitutions in the last column of Fig.2, keep track of these pattern matchings. The current substitution is updated when encountering a *branch* instruction, by composing it with that resulting from the new pattern matching. Symbolic execution furnishes information on the shape of the values in the stack during an actual execution. Although it is out of the scope of this paper, let us mention that this kind of analysis can be used to ensure bounds on the resources required by a program execution. For that, it is assumed that the bytecode comes with polynomial annotations for the constructors and the functions. These polynomials (introduced by Marion [12] under the name of quasi-interpretations) are intended to provide bounds on the size of the values built with the constructors or returned by the functions. A bound on the overall memory used by the program is then deduced from considerations on termination. One can refer to [2] for more details.

**Type Analysis as an Abstract Execution.** The bytecode is also supposed to be accompanied by type annotations, that is functions' and constructors' signatures. Type analysis comes before shape analysis. It consists in an abstract execution of the bytecode, in which the values are replaced by their types. The type verification checks the compatibility between the types in the stack and the current instruction, with respect to the signatures. When executing an instruction *branch*( $c, j$ ) for instance, one checks that the value on the top of the stack is the type of constructor  $c$ . In case of error, an error state  $\top$  is produced. One proves that if the bytecode is correctly typed, there will be no out of bounds access to the value stack, all function calls will apply to well typed arguments, and each new value is built with a constructor and arguments the types of which are consistent.

Both abstract and symbolic executions can be viewed as instances of a generic data flow analyzer (DFA) due to Kildall ([9]) that we present in the next section.

**Notations.** In the sequel, we will use the following notations :

- $e :: l$  is the list with head  $e$  and tail  $l$
- $l@l'$  is the concatenation of lists  $l$  and  $l'$
- $|l|$  is the size of list  $l$
- $[v_0; \dots; v_k]$  is the list of elements  $v_0, \dots, v_k$ . So  $[\ ]$  is the empty list.
- $l[i]$  the  $i^{\text{th}}$  element of the list, elements being indexed from 0
- $e \in l$  means  $e$  is an element of list  $l$

### 3 Kildall's Algorithm

#### 3.1 Parameterizing the Generic Data Flow Framework

Kildall's algorithm traverses the control flow graph of a function. It is a graph whose vertices are the instructions' indices in the bytecode program. There is an edge between  $p$  and  $q$  if instruction of index  $q$  can be executed immediately after that of index  $p$ . This graph has an only source, 0, that corresponds to the entry point of the function. The generic data flow framework is parameterized by:

- the number  $n$  of instructions of the bytecode, that characterizes the set  $\{0, \dots, (n-1)\}$  of the vertices also called *instructions* when it is clear from the context.
- a function *succs*, which associates with each instruction in  $\{0, \dots, (n-1)\}$  the list of instructions that can immediately follow it. It characterizes the set of the edges.

Moreover, with each vertex of the graph is associated a *state*, which generalizes the symbolic stack facing each instruction in Fig.2, or a type stack in case of the type analysis. Thus, we introduce:

- $\sigma$ , the type of the states.  $\sigma$  is equipped with a relation  $>_\sigma$ , a supremum function  $sup_\sigma$ , and a top element  $\top$ . Moreover, we assume that all ascending chains in  $\sigma$  are finite. Intuitively, the states can be seen as constraints on the value stacks handled by the VM during the evaluation of a function. Relation  $>_\sigma$  compares constraint strength: if instruction  $p$  can be executed in state  $s$ , it has to be so for every state  $s'$  such that  $s >_\sigma s'$ .

The length- $n$  lists  $ss$  of elements of  $\sigma$  are called *function states*. In the case of symbolic analysis for example, they correspond to the lists of  $n$  symbolic stacks as those displayed in Fig.2.

The algorithm also relies on a flow function *step* that takes as arguments an instruction  $p$  and a state  $s$ . (*step p s*) is the list of all states (one for each possible successor of instruction  $p$ ) resulting from the execution of  $p$  in state  $s$ . Functions *step* and *succs* are combined to define function *step'* such that, if (*step p s*) =  $[t_1, \dots, t_k]$  and (*succs p s*) =  $[q_1, \dots, q_k]$  then (*step' p s*) =  $[(q_1, t_1), \dots, (q_k, t_k)]$ .

### 3.2 Description of the Algorithm

As already mentioned, the algorithm traverses the function's flow graph. A function state  $ss$  associates state  $ss[p]$  in  $\sigma$  with each vertex  $p$ . In practice, initially all the vertices except 0 will have a special state that represents a constraint always satisfiable. It is encoded as the least element of  $\sigma$  (which is not mentioned in the previous section since it is not used for specifying and proving the algorithm). For vertex 0, the initial constraint depends on the kind of analysis that is performed. When an instruction  $p$  is reached, a call  $(step' p ss[p])$  computes a new state  $t$  for each successor  $q$  of  $p$ . The current state  $ss[q]$  is updated by  $(sup_{\sigma} t ss[q])$ . If  $ss[q]$  has actually been modified,  $q$  is moved to the *working list* of the instructions to be examined again. The process goes on until stability. Therefore, the stability of an instruction  $p$  with respect to a function state  $ss$  is defined as follows:

$$(stable\ ss\ p) := \forall (q, t) \in (step' p\ ss[p]),\ ss[q] \geq_{\sigma} t$$

Kildall's algorithm starts with any function state  $ss$  and the list (called (*worklist*  $ss$ )) of all instructions  $p$  that are not stable for  $ss$ . It calls a main loop, *iterate*, which takes as arguments a function state  $ss$  and a working list  $w$ . Iterate examines each element  $p$  in  $w$  to make it stable. This is achieved through a call to the propagation function *propagate*, which, for all successors  $q$  of  $p$ , updates  $ss[q]$  and adds  $q$  to  $w$  if  $ss[q]$  has been changed.

$$\begin{aligned} (kildall\ ss) &:= (iterate\ (ss,\ (worklist\ ss))) \\ (iterate\ (ss,\ w)) &:= \mathbf{match}\ w\ \mathbf{with} \\ &\quad [] \Rightarrow ss \\ &\quad p :: w' \Rightarrow (iterate\ (propagate\ ss\ w'\ (step' p\ ss[p]))) \\ (propagate\ ss\ w\ l) &:= \mathbf{match}\ l\ \mathbf{with} \\ &\quad [] \Rightarrow (ss,\ w) \\ &\quad (q, t) :: l' \Rightarrow \mathbf{if}\ (sup_{\sigma}\ t\ ss[q]) = ss[q]\ \mathbf{then}\ (propagate\ ss\ w\ l') \\ &\quad \quad \mathbf{else}\ (propagate\ ss[q] \leftarrow (sup_{\sigma}\ t\ ss[q]))\ q :: w\ l' \end{aligned}$$

It can be noticed that the propagation function is defined by recursion on the structure of its third argument. In contrast, function *iterate* is not defined by structural recursion. This requires us to exhibit a well-founded order on pairs  $(ss, w)$ , and a non-trivial building of *iterate* from a termination proof. This is detailed in the second part of section 3.3. Lastly, remark that although quite straightforward, the definition of *propagate* requires the equality to be decidable on  $\sigma$ .

### 3.3 Encoding in Coq Kildall's Algorithm

As far as the algorithm's specification in Coq is concerned, and comparatively to Klein and Nipkow's work with Isabelle [13, 10], two main points must be emphasized: the use of dependent types and the way of encoding function *iterate*. These are the essential differences and we shall briefly discuss the advantages of each approach.



**Specifying with dependent types.** As we have seen, *function states* are lists of states whose length is meant to remain constant and equal to the number  $n$  of instructions in the function's bytecode. This data type is encoded quite naturally in Coq by a dependent type:

```
Inductive sized_list : nat -> Set: sd_nil : (sized_list 0) |
  sd_cons : ( $\forall$  n:nat),  $\alpha$  -> (sized_list n)-> (sized_list (S n)).
```

Notice that these lists are polymorphic: they depend on set  $\alpha$  that is supposed to be declared as a parameter in the current section. Outside the section,  $\alpha$  is discharged and thus must appear explicitly in the type (as in the definition of `propagate` below in this section or in the definition of `bytecode` in section 4.1). This specification avoids the presence of hypotheses of the form  $|ss| = n$  in a great number of lemmas, all throughout the development. Similarly, we define inductively the lexicographic order and the componentwise order on lists with a type that expresses that only same-length lists can be compared:

```
lex_n, <_n : (sized_list n) -> (sized_list n) -> Prop
```

With this approach, we can prove by induction on parameter  $n$  that the lexicographic order is well-founded provided the underlying order on the elements is well-founded. Since for all natural numbers  $n$ ,  $\text{lex}_n$  is weaker than  $<_n$ , we can conclude that  $<_n$  is well-founded too.

Function `succs` computes the list of the successors of an instruction  $p$ . Instruction  $p$  and its successors all must be natural numbers less than  $n$ . Instead of taking the hypothesis:

$$(\forall p : \text{nat}), p < n \rightarrow (\forall q : \text{nat}), q \in (\text{succs } p) \rightarrow q < n \text{ (*)}$$

we define the type `dep_list` of the lists the elements of which satisfy a certain predicate:

```
dep_list : ( $\forall$   $\alpha$ : Set)( $\alpha$  -> Prop) -> Set.
```

So, we obtain the type `d_list` of the lists of natural numbers less than  $n$  as an instance of this data type :

```
d_list : = $\lambda$ n:nat.(dep_list nat  $\lambda$ p:nat.p<n)
```

Consequently, the types of functions `succs`, `step`, and `step'` are the following:

```
succs : ( $\forall$  p : nat), p<n -> (d_list n)
step   : ( $\forall$  p : nat), p<n ->  $\sigma$  -> (list  $\sigma$ )
step'  : ( $\forall$  p : nat), p<n ->  $\sigma$  -> (dep_list nat* $\sigma$   $\lambda$ (q,t):nat* $\sigma$ .q<n)
```

As an example, a version without dependent types leads to establishing

$$\text{propagate}(ss, w, l) = (ss', w') \rightarrow |ss'| = |ss|$$

while this is implicitly stated in the type of function `propagate` in our development.

```

propagate: (sized_list  $\sigma$  n) -> (d_list n)->
           (dep_list nat* $\sigma$   $\lambda(q,t):nat*\sigma.q < n$ ) ->
           (sized_list  $\sigma$  n) * (d_list n)

```

To be fair, this has been achieved through an increased effort on preliminary results, mainly concerning lists (consequent libraries on lists with dependent types have been built). But doing so, we “factorize” some proofs by moving them from specialized parts of the development to generic ones. Thus they become reusable and they are performed once and for all. Moreover, let us point out that not only the statements of the lemmas are simplified but also the proofs and the use of the lemmas, since in their applications, fewer hypotheses must be shown to be satisfied. However, using dependent types raises some difficulties. In such specifications there is a strong interdependence between logical parts, namely proof terms elegantly expressing constraints on data types, and purely computational parts. In practice one has to establish that these logical parts are irrelevant as far as computational aspects are concerned. For example, as the elements of lists of type *dep\_list* are pairs made of an element and a certificate, one must define a projection :

```

dep_list_to_list : ( $\forall \alpha : \text{Set}$ ) ( $\forall P : \alpha \rightarrow \text{Prop}$ )
                  (dep_list  $\alpha$  P) -> (list  $\alpha$ ).

```

and an equivalence on dependent lists by:

```

l  $\equiv$  l' := (dep_list_to_list l) = (dep_list_to_list l')

```

The specification and the verification of Kildall’s algorithm are performed under the following hypotheses:

```

(H1): ( $\forall p : \text{nat}$ ) ( $\forall C : p < n$ ) ( $\forall s : \sigma$ ), |(succs p C)| = |(step p C s)|
(H2): ( $\forall p : \text{nat}$ ) ( $\forall C, C' : p < n$ ), (succs p C)  $\equiv$  (succs p C')
(H3): ( $\forall s : \sigma$ ) ( $\forall p : \text{nat}$ ) ( $\forall C, C' : p < n$ ),
      (step p C s) = (step p C' s)
(H4): ( $\forall p : \text{nat}$ ) ( $\forall C : p < n$ ) ( $\forall s, t : \sigma$ ),
       $s \leq_{\sigma} t \rightarrow$  (step p C s)  $\leq$  (step p C t)

```

In (H4),  $\leq$  stands for the componentwise relation on the standard lists. This hypothesis expresses the monotonicity of function *step*. One could argue that dependent typing leads to extra hypotheses such as (H2). But, with a standard specification style, we would have instead hypothesis (\*) in the previous page. We have compared the verifications of Kildall’s algorithm with both specification styles. Using dependent types decreases of about 25% the number of lemmas. This does not take into account the libraries related to lists.

This case study is an ideal example where dependent types are of interest. Indeed, it is well known that dependent typing may offer difficulties due to the fact that types depending on terms that are provably equal but not convertible are distinct. But here, all the lists we handle have a fixed length *n* which parameterizes all the development. So, we are certain to never do arithmetic on the

length of the lists. Moreover, parameter  $n$  plays an essential role and is referred to continuously in the definitions and the lemma statements. Thus, as we hope to have demonstrated, integrating it into the type of the lists, leads to concise and elegant programming.

**Defining function *iterate* by well-founded induction.** The specification of function *iterate* in Coq is not straightforward since this system only supports total functions defined by structural recursion. This specification must include a proof of termination within its structure. The approach that we take here is similar to Bertot and Balaa's [3]. The term is built by approximations in a style inspired by Tarski's fixpoint theorem. Here are the main steps for constructing the term *iterate*.

1. We define a family of relations  $\prec_n$ , that we prove to be well-founded, on the set of pairs

$$(\mathbf{ss}, \mathbf{w}): (\text{sized\_list } \sigma \ n) * (\text{d\_list } n)$$

$$\text{by: } (\mathbf{ss}', \mathbf{w}') \prec_n (\mathbf{ss}, \mathbf{w}) := (\mathbf{ss}' >_n \mathbf{ss}) \vee (\mathbf{ss} = \mathbf{ss}' \wedge (|\mathbf{w}'| < |\mathbf{w}|))$$

2. We prove that for all  $n$ -length function states  $ss$ , instructions  $p$ , proofs  $C : p < n$ , and instruction lists  $w$ :

$$(\text{propagate } \mathbf{ss} \ \mathbf{w} \ (\text{step}' \ p \ C \ \mathbf{ss}[p])) \prec_n (\mathbf{ss}, \mathbf{p}::\mathbf{w})$$

Thus, the recursive call in the evaluation of (*iterate* ( $ss$ ,  $w$ )) is on an argument strictly less than ( $ss$ ,  $w$ ) with respect to the well-founded relation  $\prec_n$ .

3. Let  $F$  be the functional defined by:

$$(F \ f) = \lambda(\mathbf{ss}, \mathbf{w}) \ \text{if } \mathbf{w} = [] \ \text{then } \mathbf{ss} \ \text{else}$$

$$\text{let } (\mathbf{p}, \mathbf{C})::\mathbf{w}' = \mathbf{w} \ \text{in } (f \ (\text{propagate } \mathbf{ss} \ \mathbf{w}' \ (\text{step}' \ \mathbf{p} \ \mathbf{C} \ \mathbf{ss}[\mathbf{p}])))$$

We prove that for any function *bot* on pairs ( $ss$ ,  $w$ )

$$\forall(\mathbf{ss}, \mathbf{w}) \ \exists v \ (\exists k_0:\text{nat}) \ (\forall k > k_0) \ (F^k \ \text{bot} \ (\mathbf{ss}, \mathbf{w})) = v$$

For a given pair ( $ss$ ,  $w$ ) the proof is performed by induction on the fact that this pair is accessible for relation  $\prec_n$ , which follows from the well-foundedness of the relation.

4. A proof term of such a statement is a pair ( $f$ ,  $h$ ) where:

–  $f$  is a function which associates with each argument ( $ss$ ,  $w$ ) value  $v$

–  $h$  is a proof that  $(\exists k_0:\text{nat}) \ (\forall k > k_0) \ (F^k \ \text{bot} \ (\mathbf{ss}, \mathbf{w})) = v$

By deconstructing such a pair, it is possible to forget the logical comment  $h$  and to get the computational part of the term, that is program  $f$ .

5. This function  $f$  is in fact the function *iterate* that we intend to define. Indeed, we prove that  $f$  satisfies the fixpoint equation  $(F \ f) = f$ .

This is to be compared with the specification in Isabelle. In an early version of their work [13], Klein and Nipkow used an opaque well-founded recursion whereas in a more elaborate version [10] they express the function in terms of the predefined *while*-combinator of type  $(\alpha \Rightarrow \text{bool}) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$  which satisfies the equation

$$\text{while } b \text{ c } s = (\text{if } (b \text{ } s) \text{ then } (\text{while } b \text{ c } (c \text{ } s)) \text{ else } s)$$

This equation is a directly executable functional program. It makes it possible to define functions without proving any well-foundedness. However, to reason on such functions, establishing their termination is mandatory. As a matter of fact, proving that a certain property  $Q$  holds on a returned value ( $\text{while } b \text{ c } s$ ) is achieved through the following *while-rule*:

$$P \text{ } s \wedge (\forall s, P \text{ } s \wedge b \text{ } s \Rightarrow P \text{ } (c \text{ } s)) \wedge (\forall s, P \text{ } s \wedge \neg b \text{ } s \Rightarrow Q \text{ } s) \wedge \text{wf } r \wedge (\forall s, P \text{ } s \wedge b \text{ } s \Rightarrow (c \text{ } s, s) \in r) \Rightarrow Q \text{ } (\text{while } b \text{ c } s)$$

Moreover, this approach only applies to tail recursive functions. Of course, this covers a large class of functions.

### 3.4 Correctness of Kildall's Algorithm

We mentioned that the greatest element  $\top$  of  $\sigma$  stands for the error state. Therefore, all the bytecodes whose analysis generates a function state containing  $\top$  will be rejected. In order to prove that only erroneous programs are rejected, we establish that Kildall's algorithm produces the least stable function state, greater than its argument (for relation  $<_n$ ). This is done by using the monotonicity of function *step*. Now, how can this DFA be used for bytecode verification? And first of all what does it mean that the bytecode is correct with respect to a certain kind of analysis? This is expressed by a parameter  $wi$  to be later instantiated by a compatibility relationship between the instructions and a function state  $ss$ . For instance, in case of type verification, if instruction of index  $p$  is the *return* instruction,  $(wi \text{ } ss \text{ } p \text{ } \_)$  holds if and only if the element on the top of stack  $ss[p]$  is less than or equal to the return type of the function. Assuming the following relationship between predicates  $wi$  and *stable*:

$$(H5): \forall ss, \top \notin ss \rightarrow ((\forall p: \text{nat})(\forall C: p < n), (wi \text{ } ss \text{ } p \text{ } C) \leftrightarrow (\text{stable } ss \text{ } p \text{ } C))$$

we can deduce the following two propositions:

$$\top \notin (\text{Kildall } ss) \rightarrow (\forall p: \text{nat})(\forall C: p < n), (wi (\text{Kildall } ss) \text{ } p \text{ } C) \quad (1)$$

$$((\exists ts \geq_n ss) (\forall p: \text{nat})(\forall C: p < n), (wi \text{ } ts \text{ } p \text{ } C)) \rightarrow \top \notin (\text{Kildall } ss) \quad (2)$$

We do not detail the proofs here. They are similar to that in [13]. The differences are not in the proof schemes themselves, but rather in the specification style.

## 4 Application to Two Static Analyses

Let us now apply this algorithm to perform type and shape analyses on the function bytecodes for the language introduced in section 2. We start by encoding

in Coq the bytecode instructions and the VM. This part of the development is parameterized by the set *name* of the names of types, functions, and constructors. The only axiom set on *name* is the decidability of equality over it.

#### 4.1 The Virtual Machine

**Instructions.** It is assumed that every function in the program passed the following preliminary verifications : for each instruction  $p$ , successors of  $p$  are valid indices in the function's bytecode. That condition falls into two parts :

- last instruction of a bytecode is not one of *load j*, *call g ar*, *build c ar* or *branch c j* (whose successors contain the instruction which immediately follows it in the bytecode)
- jump indices  $j$  in *branch c j* instructions are less than the length of the bytecode

We choose to represent the bytecode programs by using the dependent type of lists of fixed length  $n$ . This allows us to force the second condition directly in the instruction definition, by adding a third argument of type  $j < n$  to instruction *branch*. Therefore, type *instruction* itself depends on  $n$ . It is defined as follows:

```
Inductive instruction (n:nat) : Set : return : instruction n |
  stop : instruction n |
  load : nat -> instruction n |
  call : name -> nat -> instruction n |
  build : name -> nat -> instruction n |
  branch : name -> forall (j:nat), j<n -> instruction n.
```

We can now introduce the following definition:

```
Definition bytecode := (∀ n: nat), (sized_list (instruction n) n).
```

As expected, the successors set of an instruction of index  $p$  is  $[p]$  for a *return* instruction,  $[]$  for a *stop* instruction,  $[p + 1; j]$  for (*branch c j*), and  $[p + 1]$  otherwise. The function that computes successors is encoded so as to produce lists of natural numbers less than  $n$ . It has type:

```
Succs: (∀ n: nat), (bytecode n) -> (∀ p: nat), p<n -> d_list n.
```

We can easily prove that function `succs := (Succs n bc)` fulfills condition (H2) in section 3.3.

**Functions.** Type `function` is that of records of the form  $\tilde{f} = mkfun(f, sig_f, |f|, bc_f)$ , where

- $f$ , of type *name*, is the name of the function.
- $sig_f$ , of type  $name^*(list\ name)$ , is the signature of the function. It is a pair made of the return type, and the list of the arguments' types.
- $|f|$  is an integer that denotes the bytecode's length.
- $bc_f$  is the bytecode of the function, of type (*bytecode*  $|f|$ ).

For clarity, we describe record types in a simplified notation. For instance, if we denote by  $\tilde{f}$  a term of type *function*,  $f$  is an abbreviation for  $\tilde{f}.\text{fun\_name}$ . Similarly,  $\text{sig}_f$  stands for  $\tilde{f}.\text{fun\_sig}$ ,  $|f|$  stands for  $\tilde{f}.\text{fun\_size}$ , and  $bc_f$  stands for  $\tilde{f}.\text{fun\_bytecode}$ .

Another parameter in this part of the development is `functions:(list function)` that represents the list of the functions the program is made of. Elements of this list are assumed to fulfill the following hypotheses :

$$(H6) : (\forall \tilde{f} : \text{function}), \tilde{f} \in \text{functions} \rightarrow |f| > 0$$

$$(H7) : (\forall \tilde{f} : \text{function}), \tilde{f} \in \text{functions} \rightarrow \text{last\_return\_or\_stop } bc_f$$

Here, *last\_return\_or\_stop* is a predicate on lists of instructions that expresses that the last instruction of  $bc_f$  (i.e. element at index  $|f| - 1$ ) is either a *return* or a *stop* instruction.

Function `Get_function: name  $\rightarrow$  (Opt function)` will be used to find in list `functions` the first record with name field  $f$ . *Get\_function*'s return type is optional to handle the case where no function named  $f$  appears in the program.

**Constructors.** The type `constructor` is that of records

$\tilde{c} = \text{mkcons}(c, \text{ret}_c, \text{args}_c)$ , where

- $c$ , of type *name*, is the constructor's name.
- $\text{ret}_c$ , of type *name*, is the name of the type built by  $\tilde{c}$ .
- $\text{args}_c$ , of type *(list name)*, is the list of the types of its arguments.

As for functions, parameter `constructors: (list constructor)` contains all the constructors declared in the program. `Get_constr` plays for *constructors* a role similar to *Get\_function* for *functions*.

**Frames.** Their type `frame` is that of records

$\bar{f} = \text{mkfr}(f, pc_f, \text{stack}_f, \text{args}_f)$ , where

- $f$  is the name of the function being evaluated in  $\bar{f}$ .
- $pc_f$  is the index of the current instruction.
- $\text{stack}_f$ , of type *(list value)*, is the value stack.
- $\text{args}_f$ , of type *(list value)*, is the initial stack, i.e. the arguments on which function  $f$  is evaluated.

Type `value` is that of trees the nodes of which are elements of type *name*. The fourth component in a frame does not appear in the description of the VM as given in section 2. It must be considered as a dummy field without any computational relevance. It will only be used to achieve proofs. Let us point out that with these simplified notations, given a term  $\bar{f}$  of type *frame*,  $f$  is an abbreviation for  $\bar{f}.\text{frm\_name}$ . Similarly  $bc_f$  will stand for  $\text{x}.\text{fun\_bytecode}$  where  $(\text{Get\_function } \bar{f}.\text{frm\_name}) = (\text{Some x})$ . Therefore, invoking  $bc_f$  given a frame  $\bar{f}$  implicitly induces the presence of a function named  $\bar{f}.\text{frm\_name}$  in parameter *functions*.

**Configurations.** Type `configuration` aliases (*list frame*) and it represents the machine states at runtime. The top frame (i.e. the most recent one) is the head of this list. The empty configuration `[]` represents an erroneous configuration. The instructions' semantics (see section 2) is encoded by an inductive predicate `reduction` on the configurations, which is easier to work with than a functional definition. Let us take instruction `call` as illustrative example. Its formal semantics is expressed by the rule:

$$\frac{pc_f < |f| \quad bc_f[pc_f] = (\text{call } g \text{ ar}) \quad g \in \text{functions} \quad stack_f = \text{args}@l \quad |args| = ar}{(f, pc_f, stack_f, args_f) :: M \rightarrow (g, 0, args, args) :: (f, pc_f, l, args_f) :: M} \quad (3)$$

Predicate `reduction` is defined inductively in Coq. Here is the constructor for instruction `call` :

```
red_call : (∀ M : configuration) (∀ f̄ : frame) (∀ x,y : function)
           (∀ g : name) (∀ ar:nat) (∀ args, l : list value),
           Get_function f = Some x ->
           x.fun_bytecode[pc_f] = Some (call x.fun_size g ar) ->
           Get_function g = Some y ->
           split_k_elements ar stack_f = Some (args, l) ->
           (reduction f̄ :: M
            (mkfr(g,0,args,args) :: mkfr(f,pc_f,l,args_f) :: M)).
```

Here, (`split_k_elements k l`) returns an optional pair `Some (lk, lr)`, with  $l = l_k @ l_r$  and  $|l_k| = k$  if  $|l| \geq k$ , `None` otherwise. Condition  $pc_f < |f|$  is implicitly expressed by the fact that the  $pc_f^{th}$  element of the bytecode of  $f$  is of the form (`Some ...`). We can now express that a predicate is *invariant* by reduction :

```
(invariant P) :=
  (∀ M M' : configuration), (P M) → (reduction M M') → (P M')
```

**Well-formed configurations.** Predicate `wellformed_configuration` holds on all configurations  $M$  that satisfy both conditions **wf1** and **wf2**:

**(wf1)** for each frame  $\bar{f}$  in  $M$ , (`Get_function  $\bar{f} \neq \text{None}$` ) and  $pc_f < |f|$   
**(wf2)** for each pair of consecutive frames  $(\bar{f}, \bar{h})$  in  $M$ , frame  $\bar{f}$  has been created by the last evaluated instruction in  $\bar{h}$ , i.e.  $bc_h[pc_h] = \text{Some}(\text{call } |h| f |args_f|)$

This predicate enjoys the following property, proved by case analysis on the reduction rule applied:

**Lemma 1.** *Predicate `wellformed_configuration` is invariant by reduction.*

**Executions** are defined by an inductive predicate

```
execution: name -> (list value) -> (list configuration) -> Prop
```

such that `(execution f args L)` holds if and only if  $L$  is an initial segment of the history of the configurations met when running the program that computes function  $f$  on arguments  $args$ . It is introduced by two constructors:

```

ex1 : (∀f: name)(∀args: (list value)),
      (execution f args [mkfr(f,0,args,args)])
ex2 : (∀f: name)(∀args: (list value)) (∀ L: (list configuration))
      (∀ M, M': configuration), (execution f args (M:: L)) ->
      (reduction M M') ->(execution f args (M':: M:: L))

```

It is shown that all properties  $P$  preserved through reduction are satisfied by all the configurations of an execution, provided  $P$  is true on the initial configuration :

**Lemma 2.**  $(\forall P: \text{configuration} \rightarrow \text{Prop}) (\forall f: \text{name})$   
 $(\forall \text{args}: (\text{list value})),$   
 $(P [\text{mkfr}(f,0,\text{args},\text{args})]) \rightarrow (\text{invariant } P) \rightarrow$   
 $(\forall L: (\text{list configuration})), (\text{execution } f \text{ args } L) \rightarrow$   
 $(\forall M: \text{configuration}), M \in L \rightarrow (P M)$

This statement is proved by induction on term  $(\text{execution } f \text{ args } L)$ .

**Results and errors.** By definition,  $(\text{config\_result } M v)$  is satisfied if and only if

- configuration  $M$  contains a sole frame  $\bar{f}$ ,
- the instruction of index  $pc_f$  in the bytecode of function named  $f$  is a *return* instruction,
- value  $v$  is the top element of value stack  $stack_f$ .

Similarly, an erroneous configuration is either the empty configuration, or a configuration on which no reduction can be performed and such that  $\forall v: \text{value}, \neg(\text{config\_result } M v)$ .

## 4.2 Type Verification

**Type instantiation.** We instantiate Kildall's generic algorithm in order to obtain a type verification algorithm called  $\text{Kildall}_T$ . This is done inside a Coq section parameterized by a function  $\tilde{f}: \text{function}$ . Therefore, in the terms introduced now, parameter  $\tilde{f}$  will be either implicit (inside the Coq section) or explicit (when they are referred to outside the Coq section). As usual,  $bc_f$  denotes the bytecode of  $\tilde{f}$  and  $Rt = (fst \text{ sig}_f)$  denotes its return type. Terms of type  $\sigma_T$  may be either abstract stacks encoded as lists of type names, or a special element  $\top_T$  that stands for an erroneous state, or  $\perp_T$  that reflects the absence of constraints.

**Inductive**  $\sigma_T : \text{Set} := \top_T : \sigma_T \mid \perp_T : \sigma_T \mid$   
 $\text{Types} : (\text{list name}) \rightarrow \sigma_T.$

Relation  $>_{\sigma_T}$  is the flat order:  $\top_T$  is the greatest element,  $\perp_T$  is the least element, and all the other elements are incomparable.  $\text{Kildall}_T$  will be run on the initial function state

$(\text{init}_T \tilde{f}) := (\text{Types } (\text{reverse}(\text{snd } \text{sig}_f)))::[\perp_T; \dots; \perp_T]$

since, when starting the analysis, the only constraint is to call the function with well typed arguments. The flow function *step* is instantiated by a function  $\text{Step}_T$



defined by cases on  $bc[p]$ . For lack of space we only present the case of instruction *return*.

```
(StepT p C s) :=
  match bcf[p] with
  return => match s with
    ⊥T => [⊥T] | ⊤T => [⊤T] |
    (Types l) => match l with
      [ ] => [⊤T] |
      Ret::t => if Ret=Rt then [s] else [⊤T]
```

Predicate  $wi$  is instantiated by a predicate  $Wti$  that specifies whether instruction  $p$  in bytecode  $bc$  is well-typed with respect to function state  $ss$ . As for  $Step_T$ , we only give the case related to the *return* instruction.

```
(Wti ss p C) :=
match ss[p] with
  ⊤T => False | ⊥T => True |
  Types l => match bcf[p] with
    return => match l with
      Ret::t => if Ret = Rt then True else False |
      _ => False
```

It is now mandatory to prove that the hypotheses taken in 3 are fulfilled by these terms. More precisely, we establish that function  $Step_T$  is monotone, that  $(Step_T p C s)$  does not depend on certificate  $C$ , that  $(\sigma_T, >_{\sigma_T})$  has the expected properties, and lastly that  $Wti$  coincides with *stable* on all top-free function states. Though some of these proofs are long, none is difficult. They will not be shown here. We have now at our disposal a certified type verifier, program  $Kildall_T$ , such that:

$$\top_T \notin (Kildall_T ss) \rightarrow (\forall p : \text{nat}) (\forall C : p < n), (Wti (Kildall_T ss) p C) \quad (4)$$

$$((\exists ts \geq_n ss) (\forall p : \text{nat}) (\forall C : p < n), (Wti ts p)) \rightarrow \top_T \notin (Kildall_T ss) \quad (5)$$

We can now establish two kinds of results for programs that have passed successfully the type analysis: a well-typedness property on the executions and a progress property.

**Well-typed frames.** Let us assume that every function in the program has passed the type verification, that is:

$$(H8) \forall \tilde{f} : \text{function}, \tilde{f} \in \text{functions} \rightarrow \top_T \notin (Kildall_T \tilde{f} (\text{init}_T \tilde{f}))$$

Then we introduce the notion of well-typed frames. Let  $\bar{f}$  be a frame and  $\tilde{f}$  the function retrieved in parameter *functions* from the function name appearing in  $\bar{f}$ . Frame  $\bar{f}$  is well-typed if and only if the types of the elements in its

value stack actually are those in the abstract stack of index  $pc_f$  in the function state computed by  $(Kildall_T \tilde{f} (init_T \tilde{f}))$ :

`welltyped_frame`  $\tilde{f} := \text{stack\_typing stack}_f (Kildall_T \tilde{f} (init_T \tilde{f})) [pc_f]$

**Well-typed configurations.** The notion of well-typedness is extended to configurations. A configuration satisfies predicate `welltyped_configuration` if and only if:

**wt1** :  $M$  is a well-formed configuration,

**wt2** : the top frame of  $M$  is well-typed,

**wt3** : for each pair of consecutive frames  $(\bar{f}, \bar{h})$  in  $M$ , frame

$$mkfr(h, pc_h, args_f @ stack_h, args_h)$$

is well-typed. Remark that this frame *is not* in the configuration.

As a matter of fact, no frame in a valid configuration, except the top one, is well-typed. They are in an intermediate state, in which the current instruction is a *function call*, but the arguments of the function have been popped out from the stack. The main results are in the following three lemmas. For lack of space we do not detail the proofs.

**Lemma 3.** *Predicate `welltyped_configuration` is invariant by reduction.*

**Lemma 4.** *All configurations in all executions are well-typed provided the initial function call occurs on well-typed values.*

**Lemma 5. (Progress)**  $(\forall M : \text{configuration}), (\text{welltyped\_configuration } M) \rightarrow (M = [ ] ) \vee (\exists v : \text{value}), (\text{config\_result } M \ v) \vee (\exists M' : \text{configuration}), (\text{reduction } M \ M')$

### 4.3 Shape Verification

**Shape Instantiation.** Since shape verification is done by performing a symbolic execution, it handles algebraic *expressions* built from variables, function names and constructor names. We shall consider distinguished expressions called *patterns*, in which no function symbol occurs. Fresh variables created by an instruction (*branch c \_*) at index  $p$ , with a  $m$ -ary constructor  $c$ , and a stack of height  $h$ , are named  $x_{p,h}, \dots, x_{p,h+m-1}$ . Similarly, initial arguments in the symbolic execution of a  $m$ -ary function are  $x_{0,0}, \dots, x_{0,m-1}$ . Set *name* of constant symbols is extended to a set `Name` that contains both constants symbols and variables.

**Inductive Name:** `Set := x : nat → nat → Name | symbol : name → Name`  
**Type Expression** will be that of trees whose nodes and leaves are marked with elements of *Name*. Elementary substitutions (of type `subst_elem`) are records made of two variable indices and an expression. We will note  $\{x_{i,j} \leftarrow expr\}$  such an elementary substitution. As substitutions are compositions of elementary substitutions, type `Substitution` is that of lists of elementary substitutions. Various functions are defined to handle expressions and substitutions :

- `apply_elem_tree` and `apply` respectively apply an elementary substitution and a substitution to an expression.

- (**fresh p h m**) returns a forest of single-node trees  $x_{p,h}, \dots, x_{p,h+m-1}$ .
- (**init\_vars  $\tilde{f}$** ) is the list of expressions  $[x_{0,m-1}; \dots; x_{0,0}]$  with  $m = |\text{snd sig}_f|$
- **init\_subst:nat->nat->(list value)->(Opt Substitution)**.  
Term (**init\_subst h m args**) is the substitution that matches variables  $x_{0,h}, \dots, x_{0,h+m-1}$  with  $\text{args}$ , that is the list :  

$$[\{x_{0,h+m-1} \leftarrow \text{args}[0]\}; \dots; \{x_{0,h} \leftarrow \text{args}[m-1]\}].$$
It equals *None* if  $|\text{args}| \neq m$ .
- **tree\_is\_pattern: Expression -> Prop** indicates whether its argument is a pattern or not.
- **make\_substitution: (forest Name)->(forest name)->(Opt Substitution)**  
matches a forest of expressions with a forest of values.

As in section 4.2, all terms introduced in the sequel of this section are implicitly parameterized by a given function  $\tilde{f}$ . Kildall’s algorithm is particularized, resulting in algorithm  $\text{Kildall}_S$ . The new state type  $\sigma_S$  is defined by:

**Inductive  $\sigma_S$ :Set:=**  $\top_S : \sigma_S \mid \perp_S : \sigma_S \mid$   
**Shapes: Substitution->(list Expression)-> $\sigma_S$ .**

As for types, relation  $>_{\sigma_S}$  is the flat order over  $\sigma_S$ .  $\text{Kildall}_S$  will be run on initial function state

$$(\text{init}_S \tilde{f}) := (\text{Shapes } [ ] (\text{init\_vars } \tilde{f})) :: [\perp_S; \dots; \perp_S]$$

Let us now describe function  $\text{Step}_S$  when instruction of index  $p$  is a *branch* instruction, which is the most interesting case. The formal semantics of instruction *branch* is defined by two rules:

$$\frac{pc_f < |f| \quad bc_f[pc_f] = (\text{branch } c \_ \_) \quad stack_f = c(a_1, \dots, a_m) :: l}{(f, pc_f, stack_f, args_f) :: M \rightarrow (f, pc_f + 1, [a_m; \dots; a_1]@l, args_f) :: M} \quad (6)$$

$$\frac{pc_f < |f| \quad bc_f[pc_f] = (\text{branch } c \ j \ \_) \quad stack_f = d(\dots) :: l \quad c \neq d}{(f, pc_f, stack_f, args_f) :: M \rightarrow (f, j, stack_f, args_f) :: M} \quad (7)$$

In the definition below, given in a simplified form,  $c$  and  $d$  are constructor names and  $x$  is a variable.

```
(Step_S p C s) = match bc_f[p] with (branch c j _) =>
match s with (Shapes S l) =>
  match l with
  d(e1, ...,em)::l' =>
    if c = d then
      [(Shapes S [em; ...;e1]@l');  $\perp_S$ ]
    else
      [ $\perp_S$ ; s] |
  x::l' =>
```

```

if ‘c is a constructor name’ then
  let (m = arity c) in
    let vars = (fresh p |l| m) in
      let subst = {x ← c(vars)} in
        let l’’ = (map (apply_elem_subst subst) l’) in
          [(Shapes (subst::S) (reverse vars)@l’’ ; s]
        else [⊤S; ⊤S] |
-                                     => [⊤S; ⊤S]

```

Lastly, parameter  $wi$  is instantiated by  $Wshi$ . We describe below the part of its definition related to instruction *branch*. One can observe that the definitions of  $Steps_S$  and  $Wshi$  are much alike.

```

(Wshi ss p _) = match ss[p] with (Shapes S l) =>
match bcf[p] with (branch c j _) =>
  match l with
    d(e1, ...,em)::l’ =>
      if c = d then ss[p+1] = (Shapes S [em;...;e1]@l’)
      else ss[j] = ss[p] |
  x::l’                                     =>
    if ‘c is a constructor name’ then
      let (m = arity c) in
        let vars = (fresh p |l| m) in
          let subst = {x ← c(vars)} in
            let l’’ = (map (apply_elem_subst subst) l’) in
              ss[p+1] = (Shapes subst::S (reverse vars)@l’’)
              ∧ ss[j] = ss[p]
      else False |
-                                     => False

```

From section 3.4, we deduce the following two results:

$$\top_S \notin (\text{Kildall}_S \text{ ss}) \rightarrow (\forall p : \text{nat})(\forall C : p < n), (\text{Wshi} (\text{Kildall}_S \text{ ss}) p C) \quad (8)$$

$$(\exists ts \geq_n \text{ ss}) (\forall p : \text{nat})(\forall C : p < n) (\text{Wshi} ts p C) \rightarrow \top_S \notin (\text{Kildall}_S \text{ ss}) \quad (9)$$

**Well-shaped frames.** Let us assume that every function in the program has passed both the type and shape verifications, that is:

$$(H9): (\forall \tilde{f} : \text{function}), (\tilde{f} \in \text{functions}) \rightarrow \top_S \notin (\text{Kildall}_S \tilde{f} (\text{init}_S \tilde{f}))$$

We introduce the notion of well-shaped frames. Let  $\bar{f}$  be a frame and  $\tilde{f}$  be the function retrieved in parameter *functions* from the function name appearing in  $\bar{f}$ . Frame  $\bar{f}$  is well-shaped if and only if the elements in its value stack actually match the expressions in the symbolic stack of index  $pc_f$  in the function state computed by  $(\text{Kildall}_S \tilde{f} (\text{init}_S \tilde{f}))$ . More precisely, assuming that

$(\text{Kildall}_S \tilde{f} (\text{init}_S \tilde{f}))[\text{pc}_f] = (\text{Shapes } S \ 1)$ , there exists a substitution  $\rho$  such that :

- $\rho$  results from the matching of the actual arguments  $\text{args}_f$  in the function call and  $(\text{map } (\text{apply } S) (\text{init\_vars } \tilde{f}))$ .
- $l$  and  $\text{stack}_f$  are same length.
- For each pattern  $p_j$  of index  $j$  in the symbolic stack  $l$ , if  $v_j$  denotes the value of same index in the value stack  $\text{stack}_f$  then  $v_j = (\text{apply } \rho \ p_j)$ .

**Well-shaped configurations.** Similarly to what has been done for type verification, the notion of well-shapedness is extended to configurations. Configuration  $M$  satisfies predicate `wellshaped_configuration` if and only if:

**wsh1** :  $M$  is a well-typed configuration,

**wsh2** : the top frame of  $M$  is well-shaped,

**wsh3** : for each pair of consecutive frames  $\tilde{f} \ \bar{h}$  in  $M$ , the frame  
 $(h, \text{pc}_h, \text{args}_f @ \text{stack}_h, \text{args}_h)$

is well-shaped.

We establish the following two lemmas that are the analogous of lemmas 3 and 4 in section 4.2.

**Lemma 6.** *Predicate `wellshaped_configuration` is invariant by reduction.*

**Lemma 7.** *All configurations in all executions are well-shaped provided the initial function call occurs on well-typed values.*

Lemma 7 is a straightforward corollary of lemma 6. But proof of lemma 6 is quite long. It is performed by case analysis on the reduction rule that is applied. We detail the case of rule 6 in appendix 5.

Let us mention that, as an instruction *branch* performs a pattern matching on the top of the current stack, the symbolic analysis only makes sense if the top of this stack is a pattern. This condition is not too restrictive and is fulfilled by all the bytecode programs produced by our compiler. In case of untrusted bytecode, it can be easily checked by scanning the function state computed by  $(\text{Kildall}_S \tilde{f} (\text{init}_S \tilde{f}))$ . Consequently, we will assume that:

(H10) *For all function  $\tilde{f}$  in the program that passed the shape analysis, the expression on the top of the symbolic stack on which a branch instruction is performed is a pattern.*

## 5 Conclusion

In other work, such as those cited in the introduction, the java bytecode verifier has been described as an instance of a generic DFA. Here, we really make use of such a generic approach since we apply it to two distinct kinds of static analyses, and we intend to extend it to a third one. All of them are part of the same system designed to ensure bounds of the memory used when executing a program.

Let us point out that Kildall's DFA is much more powerful than needed for our language, since it accommodates any state space which is a well-founded lattice (this is the case of the type lattice for languages with subtyping). In our work (see [2]), we only deal with flat lattices and the functions' flow graphs are supposed to be trees. We intend in the near future to allow bytecode optimizations, code sharing, and to enrich the language with object features. This is why we have treated carefully the generic part of this work. Moreover, such a functional specification of the broadly used Kildall's algorithm is of general interest. In this field, the work closest to ours is that of Klein and Nipkow's [13, 10], and a detailed comparison is given in section 3. Let us also mention the work of Barthe and al. [4] where an "abstract virtual machine" for the JavaCard language is specified in the Coq Proof Assistant. More recently, Cachera, Jensen, Pichardie, and Rusu, have obtained the same kind of certified data flow analyser by extraction from a Coq proof of Tarski's theorem [7]. Let us point out that our analyser can easily be obtained by extraction from our functional description in Coq. Moreover, we also have extracted a certified VM from the *progress* lemma 5, and so we fill all the gaps between the programs that are actually run (DFA and VM) and their specification in Coq. Let us also mention the work of Sălcianu and Arkoudas: they trust the computation of the DFA as a least fixpoint and prove by hand that three conditions are sufficient for an analysis to be correct; then they propose to verify them in Athena [1]. Finally, we refer the reader to a connected work by Bertot, Grégoire, and Leroy, to appear in these proceedings [6] and signaled by the editors. Kildall's algorithm is used to certify in Coq two compiler optimizations. Kildall's algorithm is presented quite concisely, which makes comparisons uneasy.

Our Coq development consists of 14000 lines among which 1900 are related to the generic Kildall's algorithm, 2500 to type analysis, 5500 to shape analysis, and 2600 to dependent lists. This shows in particular that the second verification is much more complex than the first one. Coq files are available at the URL: <http://www.cmi.univ-mrs.fr/~solange/criss.html>.

## References

1. Konstantine Arkoudas Alexander Sălcianu. Machine-checkable correctness proofs for intra-procedural dataflow analyses. In *Compiler Optimization meets Compiler Verification (COCV 2005)*, Edinburgh, Scotland, 2005.
2. Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal-Zilio, and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, pages 265–279, 2004.
3. Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.
4. Gilles Barthe, Guillaume Dufay, Line Jakubiec, and Simão Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 2294 of *Lecture Notes in Computer Science*. Springer, 2002.

5. Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard P. Serpette, and Simão Melo de Sousa. A formal executable semantics of the javacard platform. In *ESOP*, pages 302–319, 2001.
6. Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Posproceedings of Types'04*, 2005.
7. David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *FP5 IST thematic network IST-2001-38957 APPSEM II*, Tallinn, Estonia, 2004.
8. Allen Goldberg. A specification of java loading and bytecode verification. In *ACM Conference on Computer and Communications Security*, pages 49–58, 1998.
9. Gary Arlen Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.
10. Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298:583–626, 2003.
11. Xavier Leroy. Java Bytecode Verification : Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
12. Jean-Yves Marion. *Complexité implicite des calculs de la théorie à la pratique*. PhD thesis, Habilitation à diriger des recherches, Université Nancy 2, December 2000.
13. Tobias Nipkow. Verified bytecode verifiers. In M. Miculan F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *LNCS*, pages 347–363. Springer-Verlag, 2001.

## Appendix : Proof of Lemma 6

Let  $\tilde{f}$  be a function having passed the shape analysis and let us define  $ss = (Kildall_S \tilde{f} (init_S \tilde{f}))$ . From hypothesis (*H9*) (section 4.3), we know that  $ss$  does not contain  $\top_S$ , and from proposition (8) (also in section 4.3) we deduce that

$$\forall p < |f|, (Wshi \ ss \ p) \quad (10)$$

Let  $M$  and  $M'$  be two configurations such that (*reduction M M'*) holds. Assuming that  $M$  is well-shaped, we have to prove that  $M'$  is well-shaped. We proceed by case analysis on the rule applied in the derivation of (*reduction M M'*). We will focus on the sole rule reduction (6). We know that:

$$\begin{aligned} M &= \bar{f} :: M_0, \text{stack}_f = c(a_1, \dots, a_m) :: l \\ \text{and } M' &= (f, pc_f + 1, [a_m; \dots; a_1]@l, args_f) :: M_0. \end{aligned}$$

Moreover,  $bc_f[pc_f] = (\text{branch } c \ \_ \ \_)$ .  $\bar{f}$  being well-shaped,

$$ss[pc_f] = (Shapes \ S \ (e :: L))$$

and there exists a substitution  $\rho$  that matches

$$args_f \text{ and } (\text{map } (\text{apply } S)(\text{init\_vars } \tilde{f}))$$

that is :

$$(\text{map } (\text{apply } \rho) (\text{map } (\text{apply } S) (\text{init\_vars } \tilde{f}))) = \overline{args_f}^1 \quad (11)$$

<sup>1</sup> Here,  $\overline{args_f}$  is the conversion of a list of values into a list of expressions. This is just syntax, since values can be injected in expressions, and not relevant for the proof.

Moreover, we also have :

$$|e :: L| = |\mathit{stack}_f| \quad (12)$$

Lastly, for each pattern  $p_j$  of index  $j$  in the symbolic stack  $e :: L$ , if  $v_j$  denotes the value of same index in the value stack  $\mathit{stack}_f$  then

$$v_j = (\mathit{apply} \rho p_j). \quad (13)$$

In particular, since  $e$  is a pattern from (H10) (see section 4.3), we have  $(\mathit{apply} \rho e) = c(a_1, \dots, a_m)$ , and thus, we deduce that:

$$(e \text{ is a variable}) \vee (e = c(e_1, \dots, e_m) \wedge \forall i \in \{1, \dots, m\} (\mathit{apply} \rho e_i) = a_i) \quad (14)$$

From lemma 3,  $M'$  satisfies **wsh1**. It can be shown without difficulty that **wsh3** is also satisfied by  $M'$ . Let us consider condition **wsh2**. We have thus to prove that the frame  $(f, pc_f + 1, [a_m; \dots; a_1]@l, \mathit{args}_f)$  is well-shaped, that is:

- (a)  $ss[pc_f + 1] = (\mathit{Shapes} S' L')$
- (b)  $|L'| = |[a_m; \dots; a_1]@l|$
- (c) there exists a substitution  $\rho'$  such that
 
$$(\mathit{map} (\mathit{apply} \rho') (\mathit{map} (\mathit{apply} S') (\mathit{init\_vars} \tilde{f}))) = \overline{\mathit{args}_f}.$$
- (d) for each pattern  $p_j$  of index  $j$  in the symbolic stack  $L'$ , if  $v_j$  denotes the value of same index in the value stack  $[a_m; \dots; a_1]@l$ , then  $v_j = (\mathit{apply} \rho' p_j)$

From (10) we know that  $(Wshi \ ss \ pc_f)$ . From (14) and from the definition of  $Wshi$ , we have:

– **either**  $e = c(e_1, \dots, e_m)$ , then  $ss[pc_f + 1] = (\mathit{Shapes} S [e_m; \dots; e_1]@L)$ . In this case, conditions (a) and (b) are immediately satisfied. Conditions (c) and (d) are fulfilled by choosing  $\rho' = \rho$ .

– **or  $e$  is a variable**. Let  $e = x$ . In this case,

- (i)  $ss[pc_f + 1] = (\mathit{Shapes} \ \mathit{subst} :: S \ (\mathit{reverse} \ \mathit{vars})@SL)$  where:
- (ii)  $\mathit{vars} = (\mathit{fresh} \ pc_f \ |e :: L| \ m)$ . This call to function  $\mathit{fresh}$  generates  $m$  fresh variables in the way described in section 4.3. For readability, in this proof we denote them  $x_1, \dots, x_m$ , and we assume that they are actually fresh. The freshness of the generated variables is formally proved in Coq in a non trivial lemma.
- (iii)  $\mathit{subst} = \{x \leftarrow c(x_1, \dots, x_m)\}$
- (iv)  $SL = (\mathit{map} (\mathit{apply\_elem\_subst} \ \mathit{subst}) L)$

Again, conditions (a) and (b) are trivially satisfied.

Let  $\Sigma = [\{x_1 \leftarrow a_1\}; \dots \{x_m \leftarrow a_m\}]$  and  $\rho' = \rho@ \Sigma$ . Establishing condition (c) amounts to prove, from (11), that for all variables  $y$  in  $(\mathit{init\_vars} \tilde{f})$ :

$$(\mathit{apply} \rho@ \Sigma@ (\mathit{subst} :: S) y) = (\mathit{apply} \rho@ S y)$$

Let us pose  $(\mathit{apply} S y) = \mathit{expr}[x, y_1, \dots, y_k]$  where  $\mathit{expr}$  is a context and  $x, y_1, \dots, y_k$  are the variables occurring in the expression. By definition,

$$(\mathit{apply} (\mathit{subst} :: S) y) = \mathit{expr}[c(x_1, \dots, x_m), y_1, \dots, y_k].$$



Since  $x_1, \dots, x_m$  are fresh, we can deduce the following two equalities, that are proved by using several Coq auxiliary lemmas:

$$\begin{aligned}
 (\text{apply } \Sigma @ (\text{subst} :: S) y) &= \text{expr}[c(a_1, \dots, a_m), y_1, \dots, y_k] \\
 (\text{apply } \rho @ \Sigma @ (\text{subst} :: S) y) &= \text{expr}[c(a_1, \dots, a_m), (\text{apply } \rho y_1), \dots, (\text{apply } \rho y_k)] \\
 &= \text{expr}[(\text{apply } \rho x), (\text{apply } \rho y_1), \dots, (\text{apply } \rho y_k)] \\
 &= (\text{apply } \rho \text{expr}[x, y_1, \dots, y_k]) = (\text{apply } \rho @ S y)
 \end{aligned}$$

Let us now consider condition **(d)**. By hypothesis (13), substitution  $\rho$  matches all patterns in  $x :: L$  with the value of same index in  $c(a_1, \dots, a_m) :: l$ . We have to prove the similar property for substitution  $\rho @ \Sigma$  and stacks  $[x_m; \dots; x_1] @ L$  and  $[a_m; \dots; a_1] @ l$ . We can easily conclude from the fact that variables  $x_1, \dots, x_m$  are fresh.

The proof of the whole lemma in Coq takes approximately 1,000 lines. It uses a lemma concerning function *fresh* whose proof takes 2,000 lines.

# Solving Two Problems in General Topology Via Types

Adam Grabowski

Institute of Mathematics, University of Białystok,  
ul. Akademicka 2, 15-267 Białystok, Poland  
adam@math.uwb.edu.pl

**Abstract.** In the paper we show, based on two problems in general topology (Kuratowski closure-complement and Isomichi's classification of condensed subsets), how typed objects can be used instead of untyped text to better represent mathematical content understandable both for human and computer checker. We present mechanism of attributes and clusters reimplemented in Mizar fairly recently to fit authors' expectations. The problem of knowledge reusability which is crucial if we develop a large unified repository of mathematical facts, is also addressed.

## 1 Introduction

In the paper we describe a solution of the famous Kuratowski closure-complement problem, which is more a kind of mathematical puzzle than regular mathematics. Based on the large Mizar Mathematical Library, especially using its well-developed part devoted to topology, we also construct examples in the topology of the real line illustrating this exercise. We were surprised by the influence of our solution of this problem on the improvement of the whole Mizar library.

The problems described in this work are completely formalized by the author in the Mizar language. We show through advantages and disadvantages of its type system how general topology can be developed in a feasible formal way in the repository of mechanically checked mathematical texts.

Mizar is considered to be a hybrid of (at least) three elements: the first part is the language which was developed to help mathematicians writing their papers in a form understandable by machines; the second is software for checking correctness and collecting the results of this work, and last – but definitely not least – is the Mizar Mathematical Library (MML) which is considered the largest repository of the computer-checked mathematical knowledge in the world.

Since its beginnings back in 1989, the development of the MML has been strictly connected and stimulated by the formalization of topology. Now this tight connection is by no means over: the Białystok team is working to reach completion of the Jordan Curve Theorem proof, Mizar formalization of the *Compendium of Continuous Lattices* (CCL, [7]) is still ongoing, and also work on the proof of the Brouwer fixed point theorem for disks on real euclidean planes has been finished recently.

In the early years of MML its development was rather an experiment of how to deal with the system to make it usable not only for computer scientists, but

also for mathematicians to assist them in their ordinary everyday work, now its primary aim is to collect knowledge in a uniform way close to mathematical vernacular, checkable by computers as well as readable for humans. Clearly then, such large repositories can also be a kind of a handbook (or a set of handbooks) for students.

As we have mentioned earlier, the notion of a topological space needed some preliminary development, such as basic properties of sets and subsets, functions and families of subsets of a set. Three months after the start of MML, a Mizar article (only 27th in the collection) defining the notion of a topological space was accepted, namely `PRE_TOPC` [13] dated on April 14, 1989. This indicates that topology is a nice subject for formalization, being also one of the main lectures for students with many good textbooks (but also with many different approaches) to follow. Due to its many independently developed subtopics it can be formalized rapidly by a large group of people.

The paper is organized as follows: in Section 2 we briefly introduce the problem of 14 Kuratowski sets, Section 3 presents the formalization of the elementary topological notions we used. Section 4 presents how the problem was solved, Sect. 5 contains an outline of a formal description of the Isomichi's ideas on condensed subsets, while in the last sections we present a work related to the subject and we draw some concluding remarks.

Because in this paper we focus mainly on the data structure and the outline of our development rather than on technical details of the proofs, we skip correctness proofs anywhere Mizar source is quoted. Unabridged articles can be found at Mizar home page [1] and we will often refer to them by their MML identifiers (file names).

## 2 Fourteen Kuratowski Sets

The primary formulation of this problem was stated by Kuratowski in 1922 [11]. It was popularized by Kelley in [10] in the following form:

*If  $A$  is a subset of a topological space  $X$ , then at most 14 sets can be constructed from  $A$  by complementation and closure. There is a subset of the real numbers (with the usual topology) from which 14 different sets can be so constructed.*

Fig. 1 presents all 14 subsets (arrows denote set-theoretical inclusion).

We can even drop the notion of a topological space to obtain more general form (in some sense, because actually we have to choose a concrete theory to work with) of this theorem. This is quite interesting and strong result, but it has not been formalized in Mizar.

*Let  $A$  be an ordered set and let  $f : A \longrightarrow A$  be an increasing, expanding and idempotent mapping, and let  $g : A \longrightarrow A$  be a decreasing involution. Then, the semigroup generated by  $f$  and  $g$  consists of 14 elements (at most).*



- Jordan Curve Theorem – a version for special polygons (`GOBRD12`),
- Brouwer fix point theorem for real intervals (`TREAL_1`) and for disks on the real euclidean plane (`BROUWER`),
- Stone–Weierstrass theorem (`WEIERSTR`),
- Nagata–Smirnov theorem (`NAGATA_2`).

### 3.1 Structures and Attributes

The backbone structure for the entire topological part of MML is the immediate successor of `1-sorted`, that is `TopStruct` (extended only by the selector `topology` which is a family of subsets of the carrier). Originally it was designed as a type with non-empty carrier, so many of the theorems are still formulated for `non empty TopStruct`. While there are over 100 structure types in the whole MML, here the formal apparatus is very modest and restricted only to this one type.<sup>1</sup> Properties of a topological space are added to it through the attribute mechanism, namely `TopSpace-like`. Since it is very natural (the carrier of the space belongs to the topology, it is closed also for arbitrary unions of elements and binary intersections), we will not cite it here. This adjective is used in the definition of the Mizar type to understand `TopSpace` simply as `TopSpace-like TopStruct`.

### 3.2 Basic Notions

As mentioned earlier, the main Mizar functors used for our purpose were closure (which is not formally introduced in MML by four Kuratowski axioms, although abstract closure operators are also considered in MML, but is of the form below) and complement (defined naturally for a subset  $A$  of a set  $E$  as  $E \setminus A$ ). The closure  $A^-$  is defined as the set of those points  $p$  of topological space  $T$  such that all open subsets of  $T$  to which  $p$  belongs, have non-empty intersection with  $A$ .

```

definition let T be TopStruct, A be Subset of T;
  func Cl A -> Subset of T means    :: PRE_TOPC: def 13
  for p being set st p in the carrier of T holds p in it iff
    for G being Subset of T st G is open holds
      p in G implies A meets G;
end;
```

To be close to natural notation, we will also use a new postfix synonym for `Cl`, namely `-`. We will retain an original notation formalizing Isomichi's observations in Section 5.

The attribute `open` describes elements which belong to the topology of a given space. As a standard, a subset is `closed` if its complementation is `open`.

### 3.3 Some Statistics

About one-fifth of the Mizar library (184 out of 912 articles in MML) deals with topology. Even if calculation is rather rough since there is no classification of

<sup>1</sup> The Mizar type system itself is described in detail in [3].

MML e.g., according to 2000 AMS Subject Classification (and criteria are not exactly clear; as a rule we counted articles which used any notation<sup>2</sup> from [13], [6] or those dealing with formal topology).

There is an advanced searching tool – MML Query – which is useful for authors who are rather well acquainted with Mizar syntax (and behaving better when searching, not when browsing), but due to requests obtained from users to gather articles in a Bourbaki manner we plan to add AMS classification manually in the nearest future. We are considering adding the mandatory field to the bibliography file as AMS 2000 to be filled by the author but we will have to classify 912 articles by hand.

Some measurements of information flow between articles have been done by Kornilowicz. The amount of information that an article  $A$  transfers to an article  $B$  is calculated as the sum of information transferred by all theorems from  $A$  which are referred to in  $B$ , counted according to the Shannon formula. The article  $A$  is a direct ancestor of  $B$  if it transfers the largest quantity of information into  $B$ . Using this criterion, 32 articles are descendants of PRE\_TOPC in a tree of dependence of all MML items.

Most of “topological” articles (over 70) are devoted to the proof of the famous Jordan Curve Theorem (JCT). The first article dedicated to this topic is dated at the end of 1991. Even if in Mizar there is only a version of JCT for special polygons fully proved, the theorem has had a great impact stimulating the development of MML. Statistically, more than five articles devoted to this topic were submitted yearly (three in a year if we count only JORDAN series), basically written by the authors of University of Białystok, Poland and Shinshu University, Nagano, Japan.

One could ask a question, why did it take so long to codify the proof of the relatively simple theorem? The proof considered by the Mizar team was designed by Takeuchi and Nakamura [18] to be understandable for engineers, so it uses methods which are rather not much sophisticated. Furthermore, due to modular (and this modules – Mizar articles – are relatively big) development of MML, authors tend to give thorough (so more time-consuming) description of introduced notions. MML is not designed especially to develop a proof of any particular theorem, but aims at helping mathematicians with their work. The recent projects: C-CoRN [5] library connected with Fundamental Theorem of Algebra, FlySpeck – formalization of a proof of Kepler’s Conjecture as proposed by Hales may differ substantially in dynamics – similarly as his proof of JCT did, comparing with that of Mizar. While the latter project is designed to solve mainly this specific problem, C-CoRN is created to serve mathematicians in the similar (more general) way the MML does.

---

<sup>2</sup> As one of the referees pointed out, looking for used notations to determine which articles are devoted to topology seems risky, because relevant material can be scattered all over the place. It is the case of e.g., arithmetics or Boolean operations on sets, but in our opinion topology behaves much better due to many revisions done.

### 3.4 Topology of the Real Line

To construct a real example we had the possibility of choosing between the two representations of the real line with the natural topology. We decided to use  $\mathbb{R}^1$  which is the result of an application of a functor `TopSpaceMetrc` on the metric space `RealSpace`. This functor defines topology as the family of sets  $V$  such that for every point  $x$  of  $V$  there exists a ball with center in  $x$  and positive radius which is included in  $V$ .

The differences between the available representations of a natural topology on the real line, that is  $\mathbb{R}^1$  and `TOP-REAL 1` are caused by various approaches to the points they represent: the ordinary singleton and the one-elemented finite sequence. This duplication is not a desirable feature of MML, one of these approaches will probably be eliminated in the future releases of the library. The main reason that it is not done yet is that the notion of  $\mathbb{R}^1$  – which is much simpler (no additional objects such as finite sequences are involved in the definition) – cannot be easily generalized into spaces with higher dimensions, contrary to the latter `TOP-REAL 1`. Both notions are also developed in MML comparatively well (at least they share similar number of lemmas).

## 4 The Solution

In this section, we present how the problem of fourteen Kuratowski sets was expressed in the Mizar language. Although at first glance reusability of this fact is rather small, the problem itself is well-known in the literature and deserves to be formalized in MML. Even if many applications in other terms (e.g., group theory) are available, we decided to code it in a clean topological language.

Presumably, it would be better to have the more abstract form of it by first proving the theorem in the group-theoretical setting, and then instantiating that in the topological context. But the main drawback is that although merging mechanisms work in Mizar quite smoothly, there is no intuitive “forgetful” functors allowing the user to use e.g., real numbers apart from the field of reals, but still justifying it in a purely algebraic way.

### 4.1 The Barrier of Fourteen

When we tried to formulate the whole problem in Mizar, at the beginning we met one of the low quantitative restrictions of the Mizar language: enumerated sets can have at most ten elements.<sup>3</sup> This constant can be easily extended, but without changes in the checker itself, eventually we decided to divide Kuratowski sets into two smaller parts with cardinality seven, and thus we introduced the definition of the functor `Kurat14Set` to be a union of these.

---

<sup>3</sup> For some historical limitations the number of arguments in a functor definition cannot be bigger than 10, otherwise the Mizar verifier reports error number 937. Recently we asked Mizar developers to move this limitation upwards, hopefully it will be done in future releases.

When thinking of the partition of the Kuratowski sets, one can imagine a few classifications: one of them is for example to distinguish those where variables occur in positive (i.e., complement operator is applied an even number of times) or negative (odd number of occurrences, respectively) form. This “7+7” model is reasonable when we look at the diagram of Fig. 1. The positive part is the upper one.

The most useful approach however, as we point out later on, was to divide the family of subsets into its closed and open parts, and the rest (the partition “6+6+2”). Thanks to the functorial clusters mechanism we obtain immediately, that the closure of a set is closed, and the complement (‘ in Mizar) of a closed set is open.

```

definition let T be non empty TopSpace, A be Subset of T;
  func Kurat14ClPart A -> Subset-Family of T equals :: KURATO_1:def 3
    { A-, A'-, A'--', A'-, A'--', A'--'-' };
  func Kurat14OpPart A -> Subset-Family of T equals :: KURATO_1:def 4
    { A-' , A'-'- , A'-'-'- , A'-' , A'-'-', A'-'-'-' };
end;

```

```

theorem :: KURATO_1:5
  Kurat14Set A = { A, A' } \ Kurat14ClPart A \ Kurat14OpPart A;

```

This functorial registration assures that the attribute is automatically added to the functor under consideration. Via this stepwise refinement we can gradually narrow the type (give an additional specification).

```

registration let T be non empty TopSpace, A be Subset of T;
  cluster Kurat14Set A -> finite;
end;

```

The registration above is to enable the use of the functor `card` yielding natural numbers. This is needed to allow a comparison with numerals, because essentially the `Card` functor, of which `card` is a synonym, yields ordinal numbers. The following lemma was a rather easily provable property of an enumerated set.

```

theorem :: KURATO_1:7
  card Kurat14Set A <= 14;

```

The generation of all 14 Kuratowski sets can be treated as the result of a certain algorithm, so it would be enough to prove that it terminates.

Our algorithm can be roughly described as follows (in one of the simplest forms): apply the closure operator on a given collection of subsets, enlarge this collection by the results, reduce modulo set of known equations (that is, leave only one copy of the equal terms), then apply the complement operator, enlarge and reduce again; repeat these steps until no new subsets will be generated. Starting with arbitrary subset  $A$ , this algorithm will stop with no more than fourteen distinct sets. The equations which can be treated as a sort of `sos` or `hints` in Otter theorem prover terminology, are as follows:<sup>4</sup>

<sup>4</sup> For the correspondence of these with the second theorem in Section 2 we refer the reader to [17].



$$A^{--} = A^{-} \quad (1)$$

$$A'' = A \quad (2)$$

$$A^{-'-'-' = A^{-'-' \quad (3)$$

A similar problem occurred in the case of the famous elegant Rado proof of the Hall Marriage Theorem [14] when we had to formalize the process of decreasing the cardinality of any element of the family of subsets. Because still there is no feasible method of algorithm coding in Mizar (some are coded in MML, e.g., various sorting algorithms or that of Euclid, but they are stated in an abstract computer setting which in our opinion is too complex), we had to handle it in a different way.

We simply started with the output list of generated subsets, and we showed that both operators do not map outside this list. To prove this, we had to reference only a fact numbered (3) in the above set ( $A^{-'-'-' = A^{-'-'$  in Mizar notation). As it turned out to be convenient, equation (1) should be introduced as a **projectivity** property in Mizar, similarly as it is in case of (2) with a complement operator (**involutiveness**), both force automatization of the reasoning by the unification of underlying terms. To conclude that both operators do not map outside the given set, we formulated and proved the following theorem.

```
theorem :: KURATO_1:6
  for Q being Subset of T st Q in Kurat14Set A holds
    Q' in Kurat14Set A & Q- in Kurat14Set A;

registration let T be non empty TopSpace, A be Subset of T;
  cluster Kurat14Set A -> Cl-closed compl-closed;
end;
```

The above cluster uses new attributes, which can be translated (as in fact they were in the *Journal of Formalized Mathematics* [1] by the machine) as “closed for closure operator” and “closed for complement operator”. The registration has its justification based on KURATO\_1:6, which completes the proof of the first part. As we will see later on, all three implemented types of cluster registrations: existential, assuring that the radix type with the given cluster of adjectives is non-empty, conditional – stating that objects possessing properties expressed in terms of attributes have also other attributes automatically added to their type, and functorial, were extensively used. The latter two make many justifications unnecessary to write by human hand, and allowing the Mizar type analyser for a full work, while the former one allows to use the radix type with underlying adjectives.

Using methods developed and checked in the case of the closure-complement problem, we were able to solve analogously the other problem of Kuratowski, that is closure-interior. The solution was even less problematic because of the smaller number of sets (below the 10-arguments barrier).

### 4.2 The Example of 14-Subset

It is natural to search for an appropriate example in the real line with the natural topology, as Kuratowski proposed, although frankly speaking, the lemmas for the real euclidean plane are developed much better. We decided to use the set as shown below.

$$A = \{1\} \cup ((2; 4) \cap \mathbf{Q}) \cup (4; 5) \cup (5; +\infty),$$

which is translated into Mizar as follows:

```

definition
  func KurExSet -> Subset of R^1 equals :: KURATO_1:def 6
    {1} \\/ RAT (2,4) \\/ ]. 4, 5 .[ \\/ ]. 5,+infty.;
end;
```

To shorten the notation we had to define intervals of rational numbers (only open, as  $\text{RAT}(a,b)$ ). It is worth mentioning here that to keep notations for intervals as compact and close to the mathematical tradition as possible, we introduced  $].a,+infty.[$  as a synonym to the previously defined in MML  $\text{halfline}(a)$ . Both functors have only one argument since obviously  $+infty$  is not an element of the set of reals. For this part of the proof the predicate for an enumerated list of arguments, namely  $\text{are\_mutually\_different}$ , was also essential.

Due to the lack of decimal fractions in MML, the calculus of intervals of numbers was not convenient. We had to state for example a theorem of the density of the irrational numbers, even the formalization of the proof of the irrationality of the base of natural logarithms is relatively new. The fact of the irrationality of the square root of 2, which was advertised well by Wiedijk in his famous comparison of seventeen provers of the world [19], was also useful. Too few examples were previously done by others – but MML focuses on the reusability of introduced theories. The policy, which resulted in a low number of exercises formalized during the CCL project, caused the necessity of the trivial calculus on intervals.

Following [15], to show the mutual difference of subsets for each of the fourteen sets we may create a vector of boolean values 0's and 1's indicating whether fixed

**Table 1.** Incidence of test points in the example set

$$A = \{1\} \cup ((2; 4) \cap \mathbf{Q}) \cup (4; 5) \cup (5; +\infty)$$

$A$	$= [1, 0, 1, 0, 0, 1] :$	41	$A'$	$= [0, 1, 0, 1, 1, 0] :$	22
$A^-$	$= [1, 1, 1, 1, 1, 1] :$	63	$A'^-$	$= [1, 1, 1, 1, 1, 0] :$	62
$A^{-'}$	$= [0, 0, 0, 0, 0, 0] :$	0	$A'^{-'}$	$= [0, 0, 0, 0, 0, 1] :$	1
$A^{-' -}$	$= [1, 1, 0, 0, 0, 0] :$	48	$A'^{-' -}$	$= [0, 0, 0, 1, 1, 1] :$	7
$A^{-' -'}$	$= [0, 0, 1, 1, 1, 1] :$	15	$A'^{-' -'}$	$= [1, 1, 1, 0, 0, 0] :$	56
$A^{-' -' -}$	$= [0, 1, 1, 1, 1, 1] :$	31	$A'^{-' -' -}$	$= [1, 1, 1, 1, 0, 0] :$	60
$A^{-' -' -'}$	$= [1, 0, 0, 0, 0, 0] :$	32	$A'^{-' -' -'}$	$= [0, 0, 0, 0, 1, 1] :$	3

test points lie in the set or not. We can order these binary numbers by means of their decimal representation. As Rusin stated, although  $2^4 > 14$ , the number of four points is insufficient to show that all 14 sets are different. Table 2 presents points incidence where the test points were respectively: 1, 2, 3, 4, 5, 6. In our case all 6 points are needed. We used this intuitive representation only partially because there is no comfortable method for representing it in Mizar and we wanted to use the existing type machinery.

### 4.3 To Type or Not to Type

In order to avoid  $\frac{14 \cdot 13}{2} = 91$  comparisons to check that all 14 sets are different which would be necessary when developing an untyped form, we decided to use adjectives. The following lemma was crucial to reduce the complexity of the second part of the proof.

```
theorem :: KURATO_1:60
  for F, G being
    with_proper_subsets with_non-empty_elements Subset-Family of R^1 st
      F is open & G is closed holds F misses G;
```

In words, arbitrary families of subsets  $F$  and  $G$  of the real line with the natural topology such that all their members are neither empty nor equal to  $\mathbf{R}$  and  $F$  is open and  $G$  is closed (note that being open for a family of subsets means that all its elements are open, similarly for the “closed” adjective) are disjoint.

Recalling that `Kurat14OpPart A` is collectively open (that is, all its elements are open subsets) and `Kurat14ClPart A` is collectively closed, and taking into account that all elements of these families of subsets are both proper and non-empty (all these properties are automatically added via functorial cluster mechanism), we obtain a significantly reduced number of checks (which were done according to the Rusin’s method of test points). Additionally, because we know that `KurExSet` is neither closed, nor open, the final calculation is just summing the cardinalities of disjoint six-elemented sets and a doubleton.

## 5 Isomichi’s Classification of Subsets

The original problem of 14 sets have been raised the wider class of questions, named “problems of Kuratowski type”. Closure and complement can be replaced by other operators, not only topological (as the interior is), but also of more general interest (set-theoretical meet, union, difference etc.).

If we recall that for a topological interior the following equation is true

$$\text{Int } A = A'^{-'}$$

we can reuse Fig. 1 to have the illustration of closure-interior problem, also being investigated by Kuratowski, as the diagram of Fig. 2.

The maximal number of different sets we can obtain by applying the interior and closure operators to any subset  $A$  of an arbitrary topological space is seven.

### 5.1 Supercondensed and Subcondensed Sets

Sambin [16] points out the following method to find counterexamples for the other inclusions than those shown on Fig. 2 (or Fig. 1): one can choose a suitable basic pair, and use the logical expressions for interior and closure to show that they would give some implications which are not valid intuitionistically.

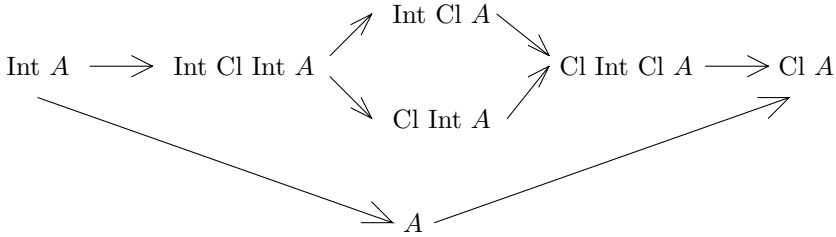


Fig. 2. Seven Kuratowski sets ordered by inclusion

Although Mizar uses classical logic, we can benefit from this observation in a similar way. We could try to describe a topological space in which some of the basic pairs are identified. We can show an example of a subset in the real line with natural topology such as the diagram at Fig. 2 does not reduce to any smaller number of distinct sets. This is the same *KurExSet* as it was considered in Subsection 4.2, and all the proofs were easily reused.

Following Isomichi [8], we call a subset  $A$  of a given topological space *subcondensed* (resp., *supercondensed*) if and only if  $Cl\ Int\ A = Cl\ A$  ( $Int\ Cl\ A = Int\ A$ )

```

definition let T be TopStruct, A be Subset of T;
  attr A is supercondensed means :: ISOMICHI:def 1
    Int Cl A = Int A;
  attr A is subcondensed means :: ISOMICHI:def 2
    Cl Int A = Cl A;
end;
```

In this manner  $A$  is a closed (resp., open) domain if and only if  $A$  is subcondensed (supercondensed) and  $A$  is closed (open). The distinction for open and closed domains is an important topic in the general topology, also considered by Kuratowski. The lattice of all open (closed) domains of a given topological space forms a Boolean algebra, which thanks to attributes is a subalgebra of the lattice of all subsets of a given space.

### 5.2 Three Classes of Subsets

Going one step further, we can unify some subsets with respect to the relations of inclusion between them (not arbitrary ones, because e.g., interior of a set is always contained in its closure). And some of the results given by this method can be also observed in the literature (usually with no reference for the Kuratowski problem, though). The connection with the 14 sets is clear if we notice

that the unification of some of them reduces the diamond formed by  $\text{Int Cl } A$ ,  $\text{Cl Int } A$ ,  $\text{Cl Int Cl } A$ , and  $\text{Int Cl Int } A$  at the diagram of Fig. 2. Neither superior nor subcondensed subsets cannot give 14 distinct subsets – just from the definition.

```

definition let T be TopSpace, A be Subset of T;
  attr A is 1st_class means :: ISOMICHI:46
    Int Cl A c= Cl Int A;
  attr A is 2nd_class means :: ISOMICHI:47
    Cl Int A c< Int Cl A;
  attr A is 3rd_class means :: ISOMICHI:48
    Cl Int A, Int Cl A are_c--incomparable;
end;

```

In the above definitions  $c=$  stands for the set-theoretical inclusion,  $c<$  – for the proper inclusion (arguments have to be distinct).  $\text{KurExSet}$  is neither superior nor subcondensed, and as it can be easily verified, it is of the 2<sup>nd</sup> class. Any subset of a given space is an element of a class of this kind, and this classification is unique (since the classes are disjoint), as it is expressed in the form of a theorem and three conditional registrations of a cluster assuring that any set which is element of the one class, cannot be member of any other. The distinction of  $\text{Cl Int}$  and  $\text{Int Cl}$  for  $\text{KurExSet}$  following just from the definition of 2<sup>nd</sup> class could result in shortening of some proofs in Section 4.

```

theorem :: ISOMICHI:46
  A is 1st_class or A is 2nd_class or A is 3rd_class;

```

Given classification of subsets is also discussion connected with a Sambin's question, whether other inclusions in the diagram of Fig. 2 are true.

```

registration let T be TopSpace;
  cluster supercondensed -> 1st_class Subset of T;
  cluster subcondensed -> 1st_class Subset of T;
end;

```

If the only open subsets are the empty set and the carrier  $X$ , i.e., we have anti-discrete topological space, any of its proper subsets  $A$  has interior empty and closure equal to  $X$ , so we have

$$\emptyset = \text{Cl Int } A \subset \text{Int Cl } A = X.$$

```

registration let T be anti-discrete non empty TopSpace;
  cluster proper non empty -> 2nd_class Subset of T;
end;

```

Via conditional cluster registrations we let the Mizar type analyzer automatically add to the type given after an arrow “->” a collection of adjectives before the arrow.

Both empty set and the whole space are 1<sup>st</sup> class subsets. We can thus conclude that anti-discrete topological spaces with at least two elements contain subsets of the 2<sup>nd</sup> class.

```

definition let T be TopSpace;
  attr T is with_2nd_class_subsets means :: ISOMICHI:def 10
    ex A being Subset of T st A is 2nd_class;
end;

```

Similar adjectives are defined for 1<sup>st</sup> and 3<sup>rd</sup> class subsets, although the earlier can be considered somewhat redundant. Because the empty subset is both super- and subcondensed, subsets of the first class exist in an arbitrary topological space.

```

registration let T be TopSpace;
  cluster 1st_class Subset of T;
end;

```

### 5.3 Existence Problems

Relatively recent reimplementations of attributes done by Byliński canceled an issue of the so-called unclusterable attributes<sup>5</sup>, and now types `Subset of T` and `Subset of the carrier of T` became exactly the same objects. Earlier registration with the latter type was unacceptable. So now the only problem with these existential registrations of clusters is whether we can construct an appropriate example. (Recall that we cite here registrations without proofs.)

Sometimes the objects just cannot exist in general, as it is with 2<sup>nd</sup> class subsets, then a type in a locus should be specified more carefully.

```

registration let T be with_2nd_class_subsets TopSpace;
  cluster 2nd_class Subset of T;
end;

```

Essentially, we could try to enumerate equations identifying certain expressions in Fig. 2 obtaining in this way characterizations of various topological spaces. Even if the example set from Section 4.2 in this paper is not 3<sup>rd</sup> class subset, still the real line is a good testbed for counterexamples; there are even many more complicated examples in topology constructed in the MML, e.g., Sorgenfrey line, Sierpiński space or the Cantor set.

Kuratowski's problem is very influential, type-free in primary formulation, raised many questions strictly connected with typed approach to a topological space. Interesting question which can be formulated is an issue of properties of subsets to have 14 distinct sets.

This classification is also important because observing some general properties (top-down clustering of knowledge), we can store the information better. And due to automatization, we do not need reprove similar theorems again and again, even if copy-and-paste techniques are still very often used by the authors of Mizar articles. MML still lacks some solutions which fully reflect expressive capabilities of the Mizar language – topological metric spaces can now be

---

<sup>5</sup> When defining an attribute, all used variables had to be directly (without additional functors) accessible, otherwise an existential cluster with this attribute could not be written.

considered descendants of metric and topological spaces due to multiple inheritance mechanism, but older articles (and hence all which use them) still use a technique of addition of transforming metric spaces into topological ones.

## 6 Related Work

From a predicative point of view the definition of a topological space in MML is unacceptable (quantification over families of subsets of the carrier of a space), also because of the classical logic standing behind the Mizar system, but its library could benefit from translation of some results in formal topology (constructive proofs can be translated into the Mizar language and positively verified by the Mizar checker). As a rule, MML developers tend rather to eliminate predicates and use adjectives instead. So the formal topology as introduced in `FINTOPO` series has very little in common with the constructive approach which is outlined well by Sambin in [16] where many interconnections with other disciplines are studied. Although Mizar's `TopSpace` is a concrete space, it was also tied with posets when codifying [7].

Many recent grand challenges for formalized mathematics deal with general topology. This choice is reasonable for Mizar as it is based on the set theory and classical logic. But the proofs of the Kepler Conjecture and the Jordan Curve Theorem which were chosen by Hales to codify with `HOL-Light`, as well as the impressive formalization of the Four Color Theorem done in `Coq` by Werner and Gonthier are placed in the topological field.

What should be mentioned here is also a didactical value of this formalized discipline: five exercise sets in topology were prepared for PC Mizar by Czuba and Bajguz in 1989–90, ca. 100 pages each. A similar choice for didactical experiments (but without use of Mizar) was made for instance by Cairns and Gow [4].

As we could conclude from successful experiments with lattice theory (solution of the Robbins problem, equational characterization of lattices in terms of Sheffer stroke, disjunction and negation, including short single axioms), many proofs may be retrieved from other specialized proof-assistants. Thanks to a Lisp script `ott2miz` (created by Josef Urban) which converts Otter proof object into an untyped (in some sense, because used objects are of type `set`) self-containing Mizar source code which can be simplified by Mizar scripts and verified by the Mizar checker, the author's responsibility can be restricted mainly to creating a proper input and appropriate data structure (placed possibly deeply within Mizar type hierarchy).

## 7 Conclusion

The experience with formalization of fourteen sets of Kuratowski and Isomichi's paper shows that the Mizar system, in its current state, not only is a good framework for formalization of mathematical structures and their properties, but also, due to the large repository of mathematical knowledge MML, is capable of being

used for the construction of examples and counterexamples. The general topology delivers many problems which can be challenges for various mathematics assistance systems.

The main achievement of our work is the formalization of the Kuratowski problem, its full solution in terms of clean general topology and the construction of an example of a set generating exactly fourteen Kuratowski sets. We put the stress not on the brute force proving, but we tried to find a possibly elegant proof. As a by-product, we obtained the characterization of condensed sets and improved topological type hierarchy in MML.

Even if the problem itself has been solved completely, during the codification a number of ideas arose. Essentially, at the second stage of solving Kuratowski's problem, when the example set had to be constructed, it soon became apparent that the Library Committee of the Association of Mizar Users should consider a massive revision of MML to merge both metric and topological spaces, because without such a revision one has to recode between different approaches. We have different representations of the real line with the natural topology, and this revision should simplify the formal apparatus. It is a question of policy – on the one hand we give users freedom of choice on their own, but on the other – abundance of notions can be quite misleading, even if the Mizar system offers more querying mechanisms than only textual search as it was before.

Roughly speaking, library revisions are caused by two processes. One of them is continuous improvement of the system (e.g., strengthening the Mizar checker or developing new language features). Also formalization projects, especially broader ones, force library developers to revise other articles in the topic. As an example we can give here introducing new lemmas which allow to prove theorems in a more general setting, with the possibility of its further reusing at the same time. It is often the case that the author does not even know about connections with other parts of mathematics (from algebraist's viewpoint, 14 sets of Kuratowski can be just instantiation of some group theory problem in the topological context).

Recent projects to split MML into concrete (or classical) and abstract parts (i.e., using the notion of a structure) have obviously one significant value: in this way some accidental connections between articles may be removed to make MML more compact. The larger items are, the smaller the mess in the library – and consequently knowledge can then be retrieved more effectively. Topology is now clusterized relatively well; it turns out to be the most promising discipline to be the most thorough developed in the Mizar library.

## Acknowledgements

The author would like to thank anonymous referees, whose valuable suggestions much helped improve the quality of the paper. The work was partially supported by the grants: of the Ministry of Scientific Research and Information Technology of the Republic of Poland 4 T11C 039 24 and of FP6 IST TYPES No. 510996.



## References

1. Journal of Formalized Mathematics, accessible at <http://mizar.uwb.edu.pl/JFM/>, Mizar library available at <ftp://mizar.uwb.edu.pl/pub/mml/>.
2. L.K. Bagińska and A. Grabowski, *On the Kuratowski closure-complement problem*, Formalized Mathematics, 11(3), 2003, pp. 321–327, MML Id in [1]: KURATO\_1.
3. G. Bancerek, *On the structure of Mizar types*, in: H. Geuvers and F. Kamareddine (eds.), Proc. of MLC 2003, ENTCS 85(7), 2003.
4. P. Cairns, J. Gow, and P. Collins, *On dynamically presenting a topology course*, Annals of Mathematics and Artificial Intelligence, 38(1-3), 2003, pp. 91–104.
5. L. Cruz-Filipe, H. Geuvers, and F. Wiedijk, *C-CoRN, the Constructive Coq Repository at Nijmegen*, in: A. Asperti, G. Bancerek, and A. Trybulec (eds.), Proc. of MKM 2004, Lecture Notes in Computer Science 3119, 2004, pp. 88–103.
6. A. Darmochwał, *Families of subsets, subspaces and mappings in topological spaces*, Formalized Mathematics, 1(2), 1990, pp. 257–261, MML Id in [1]: TOPS\_2.
7. G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott, *A Compendium of Continuous Lattices*, Springer-Verlag, 1980.
8. Y. Isomichi, *New concepts in the theory of topological space – supercondensed set, subcondensed set, and condensed set*, Pacific Journal of Mathematics, 38(3), 1971, pp. 657–668.
9. M. Jastrzębska and A. Grabowski, *The properties of supercondensed sets, subcondensed sets, and condensed sets*, to appear in Formalized Mathematics, 2005, MML Id in [1]: ISOMICHI.
10. J.L. Kelley, *General Topology*, van Nostrand, 1955.
11. K. Kuratowski, *Sur l'opération  $\bar{A}$  de l'Analysis Situs*, Fundamenta Mathematicae, 3, 1922, pp. 182–199.
12. K. Kuratowski, *Topology*, PWN – Polish Scientific Publishers, Warszawa 1966.
13. B. Padlewska and A. Darmochwał, *Topological spaces and continuous functions*, Formalized Mathematics, 1(1), 1990, pp. 223–230, MML Id in [1]: PRE\_TOPC.
14. E. Romanowicz and A. Grabowski, *Hall Marriage Theorem*, Formalized Mathematics, 12(3), 2004, pp. 315–320, MML Id in [1]: HALLMAR1.
15. D. Rusin, posting available at <http://www.math.niu.edu/~rusin/known-math/94/kuratowski>.
16. G. Sambin, *Some points in formal topology*, Theoretical Computer Science, 305 (1–3), 2003, pp. 347–408.
17. D. Sherman, *Variations on Kuratowski's 14-set theorem*, available at <http://www.math.uiuc.edu/~dasherma/14-sets.pdf>.
18. Y. Takeuchi and Y. Nakamura, *On the Jordan curve theorem*, Shinshu Univ., 1980.
19. F. Wiedijk, *Seventeen provers of the world*, available at <http://www.cs.ru.nl/~freek/comparison/>.
20. E.W. Weisstein et al., *Kuratowski's closure-complement problem*, from MathWorld – a Wolfram Web Resource. <http://mathworld.wolfram.com/KuratowskisClosure-ComplementProblem.html>.

# A Tool for Automated Theorem Proving in Agda<sup>\*</sup>

Fredrik Lindblad and Marcin Benke

Department of Computing Science,  
Chalmers University of Technology/Göteborg University,  
412 96 Göteborg, Sweden

**Abstract.** We present a tool for automated theorem proving in Agda, an implementation of Martin-Löf's intuitionistic type theory. The tool is intended to facilitate interactive proving by relieving the user from filling in simple but tedious parts of a proof. The proof search is conducted directly in type theory and produces proof terms. Any proof term is verified by the Agda type-checker, which ensures soundness of the tool. Some effort has been spent on trying to produce human readable results, which allows the user to examine the generated proofs. We have tested the tool on examples mainly in the area of (functional) program verification. Most examples we have considered contain induction, and some contain generalisation. The contribution of this work outside the Agda community is to extend the experience of automated proof for intuitionistic type theory.

## 1 Introduction

Automated proving in first-order logic is well explored and developed. Systems based on higher-order logic have in general limited automation. This is in particular true for proof-assistants based on intuitionistic type theory. There is strong motivation for working with these formalisms and the tools based on them have a large user community. As a result, a lot of interactive proving is carried out to construct proofs or parts of proofs which could conceivably be solved automatically.

We have developed a tool for automated proving in the Agda system [3]. It is not a complete proof search algorithm. The aim is to automate the construction of the parts of a proof which are more or less straightforward. Often such parts can be tedious to fill in by hand, however significant time could be saved, allowing the user to spend her effort on the key parts of the proof.

Agda is an implementation of Martin-Löf's intuitionistic type theory [10], which, following the paradigm of propositions-as-types and proof-as-objects, can be used as a logical framework for higher-order constructive logic. Agda is a type-checker and an assistant for constructing proof terms interactively. The assistant is not based on tactics. Instead, the user sees a partially completed proof term during the process. The incomplete parts of the term are represented

---

<sup>\*</sup> Research supported by the project *Cover* of the Swedish Foundation of Strategic Research (SSF).

by place-holders, called *meta variables*. Each step consists of refining one of the remaining meta variables, which instantiates it to a term that may contain new meta variables. Refining one meta variable may also instantiate others, as a result of unification.

The tool does not rely on an external solver. During the proof search, the problem and the partial solution are represented as Agda terms. The tool is integrated with the Agda proof assistant, which runs under *emacs*. When standing on a meta variable, the user can invoke the tool, which either inserts a valid proof term or reports failure of finding a solution. Failure is reported if the search space is exhausted or if a certain amount of steps has been executed without finding a solution. There is also a stand-alone version of the tool, intended for development and debugging, where a proof search can be monitored step by step.

Before inserting a proof term, it is always verified by the Agda type-checker. Therefore, the importance of soundness is limited, and we will not further discuss this issue. Still, we believe that the basic steps of the proof search are consistent with the Agda typing rules and moreover that the algorithm is sound.

Since the area of application is within an interactive proof assistant, producing human readable proof terms is important and has received some attention in our work.

The tool handles hidden arguments, which is a new feature of Agda. It has however so far no support for inductive families [7], except for when they are used for representing equalities.

The fundamental restriction of the tool is that it does not do higher order unification. Instead, a decidable extension of first-order unification is used. In e.g. Dowek's algorithm for term synthesis in pure type systems [5], applying an elimination rule produces a constraint, which is successively solved by higher-order unification. Our tool instead only applies an elimination rule if the unification procedure returns a unifier.

In Agda, termination is not verified by the type-checker. There is a separate termination-check. In our tool, inductive proofs are restricted to structural recursion. Determining termination is in general undecidable and defining advanced criteria is an issue in itself. An alternative approach would be to ignore termination and let an external verification accept or reject a proof term. However, the tool is currently designed to produce only one solution, which makes it inappropriate for that approach. Although the flexibility of induction schemes is limited, the tool can do nested induction on several variables and also nested case split on one variable.

Elimination rules are generally only applied when there is a suitable variable in the context to apply it on. The exception to this is that the tool identifies a couple of cases of generalisation. In some cases this leads to a necessary strengthening of the induction hypothesis. The generalisation mechanism is however restricted to what can be syntactically deduced from the current goal type, like replacing a repeated subexpression by a fresh variable. There is also no synthesis of new hypotheses, i.e. the tool is unable to do lemma speculation.

In addition, there are a number of other restrictions, e.g. that new data types and recursive functions are not synthesised. This means that, when searching for a proof of a proposition which is existentially quantified over *Set* or a function space, a new data type or a new recursive function will not be synthesised.

Section 2 contains a small survey of related work. Section 3 describes the tool and contains a few subsections, which are devoted to special features. In section 4 we present a few examples and discuss the limitations of the tool. Section 5 gives conclusions and ideas for how the tool could be improved in the future.

## 2 Related Work

Although the type inhabitation problem is undecidable for a system like Agda, it is semi-decidable simply by term enumeration (plus decidability of type-checking). A complete proof synthesis algorithm for the pure type systems has been presented by Dowek [5]. Cornes has extended this to a complete algorithm for the Calculus of Inductive Constructions [4]. Although these algorithms are of theoretical interest, complete algorithms so far seem too time-consuming to be of practical use. In her thesis, Cornes elaborates on various enhancements, but, to our knowledge, there is no implementation available.

We now turn to a quick survey of related implementations, beginning with a piece of related research in the context of Agda, and then working our way outwards in wider circles. Smith and Tammet [12] have investigated using a FOL-solver to mechanically find proofs in *Alf* [9], the predecessor of Agda. The goal type together with the typing rules of Alf are encoded in first-order logic and a solver, *Gandalf*, is invoked. If a solution is encountered, a proof term is constructed using the information produced by Gandalf. The authors managed to generate some inductive proofs, but the approach seems rather inefficient.

The proof-assistant *Coq* is based on a language closely related to that of Agda. Coq has many sophisticated tactics, but the *auto* tactic, which is the nearest counterpart of our tool, does not produce any inductive proofs.

Also related to Agda is the *Logical Framework*, implemented in the system *Twelf*. Schürmann and Pfenning [11] have developed a proof search tool and supporting theory for a metalogic in *Twelf*, an implementation of the logical framework LF.

Andrews has successfully explored the area of mechanical proofs in classical type theory [2]. His work has resulted in *TPS*, a fully automatic proof system based on higher-order unification.

*ACL2*, *PVS* and *Isabelle* are other major proof assistants. ACL2 and PVS do have automation for induction, but none of the systems produces proof objects.

## 3 Tool Description

Agda has dependently-typed records and algebraic data types. The algebraic data types can be recursively defined. Function arguments are analyzed by

case-expressions rather than pattern matching. Functions can be recursively defined freely using self-reference. There is no fixpoint construction. Type-checking does not include termination check. Although there is a separate termination checker, the restrictions are not clearly defined in the semantics. Hence, a tool for automated proving must either ignore termination issues or define its own criteria for this. With a proof search algorithm capable of producing multiple solutions, the first approach could be used. For each solution an independent termination check is consulted. If it rejects the proof term, the search is continued. Our tool is however designed to come up with only one solution, so it adheres to the second approach. Proof terms are currently restricted to structural recursion.

Agda is monomorphic but polymorphism is in recent versions simulated by argument hiding. Properties and proof terms below are presented with some type arguments omitted. This is done to improve readability, but the hiding mechanism is not further discussed.

Just like the Agda proof assistant, the tool uses place holders to denote incomplete parts of a proof. Place holders are called meta variables and are denoted by ‘?’. They can be seen as existentially quantified variables.

The most significant characteristics of the tool are the following:

- Unification is first-order and is not an interleaved part of the proof search. The search state does not have constraints. Unification is decided immediately when applying elimination is considered.
- The order in which the meta variables are refined is dictated by depth-first traversal. After refining a meta variable, its subproofs are recursively addressed one by one.
- Meta variables are classified as either *parameter meta variables* or *proof meta variables*. Parameter meta variables are those which appear in the type of some other meta variable, whereas the rest are proof meta variables. The parameters are the term-like meta variables. Only proof meta variables are subject to resolution. Parameter meta variables are supposed to be instantiated when unifying one of the types where they appear. In [5] Dowek pointed out that variables should be instantiated from right to left. The parameter/proof distinction is an alternative to this rule and it postpones the instantiation of term-like variables in a more flexible way. The distinction is also made for local variables, although not that explicitly.
- A meta variable is refined by an elimination only when there is a suitable object in the context. This means that for each meta variable, all possible ways to eliminate any object in scope are considered. It can be any combination of function application, record projection and also case split for algebraic data types. The fact that this applies to disjoint unions makes it necessary to tag solutions with *conditions* for which it is valid. As an example, if we have  $[h : A + B] \vdash ? : A$ , then a solution is  $[h \rightarrow \text{inl } a] \vdash a$ , which means  $a$  is a valid term provided that  $h$  is of the form  $\text{inl } a$ . The idea is that conditional solutions at some point should pair off to form an unconditional proof of the full problem.

The program consists of an algorithm which explores the search space and the implementations of a set of refinement rules. The proof search algorithm is presented in subsection 3.1. The refinement rules define the atomic steps of refining a problem and how to construct the corresponding proof term. The implementations of the refinement rules will be referred to as *tactics*. We will not present a formal description of the rules, since they are closely related to the typing rules of Agda. There are refinement rules for constructing  $\lambda$ -abstractions, record objects and algebraic data type objects. There is a rule for elimination, which combines several eliminations of different kind in the same refinement, as described above. There is also a special elimination rule for equalities, which uses transitivity, symmetry and substitutivity in a controlled manner. Then there is one rule for case analysis and induction and one for generalisation. They are presented by example in subsections 3.2 and 3.4 respectively. Finally, there is a rule for case on expression, which combines case analysis and generalisation. It is presented in section 3.3.

In the tactics which perform elimination and at some other places in the search algorithm, first-order unification is used to compare types in the presence of meta variables. Unification is always performed on normalised terms. The tool uses an extension of normal first-order unification. This enables it to deal with more problems without resorting to higher-order unification. The extension is still decidable. However, while a first-order unification problem has either one or no unifier, a problem of the extended unification can have any number of unifiers. The extension is presented in subsection 3.5. When doing first-order unification in the presence of binders, attention must be paid to scope. We have chosen to solve this by having globally unique identifiers for variables. Whenever a meta variable is about to be instantiated, its context is checked against the free variables of the term. If not all variables are in scope, the unifier is rejected.

When a proof term has been found, the tool does a few things to make it more readable. This includes using short and relevant names for local variables and removing redundant locally defined recursive functions.

### 3.1 Proof Search Algorithm

The search space is explored using iterated deepening. This is necessary since a problem may in general be refined to infinite depth. It is also desirable since less complex solutions are encountered first.

We will describe the proof search algorithm by presenting a pseudo program for the main recursion. The style is a mixture of functional and imperative programming, but we hope that the meaning of the program is clear. It refers to a few subfunctions which are only described informally. To make the presentation cleaner, unification is assumed to produce only one unifier, i.e. it does not incorporate the extension described in subsection 3.5.

First we define a few types which describe the basic objects of the algorithm. The elementary entities are denoted by single letters. The same letters are used both to denote the types and the corresponding objects, hopefully without

confusion. The general form of a problem of finding an inhabitant of type  $T$  in the variable context  $\Gamma$  is the following:

$$\Delta_{par}, \sigma, \Gamma, \rho \vdash T \quad (\text{Prb})$$

The corresponding type is called **Prb**. Here,  $\Delta_{par} = [\Gamma_i \vdash ?_i : T_i]$  is the collection of parameter meta variables. For each meta variable, its context and type is given. The next component,  $\sigma = [?_i := M_i]$ , is the set of current parameter meta variable instantiations, which should be taken into account when reducing terms. After the context of the current problem,  $\Gamma$ , we have  $\rho = [M_i \rightarrow \mathbf{c}_i y_{i,1} \cdots y_{i,n_i}]$ . This is the sequence of conditions which have emerged so far. The conditions for proof variables should be taken into account in order to avoid clashes when eliminating disjoint unions. The conditions for parameter variables should be respected when normalising types.

A solution, **Sol**, to a problem has the following form:

$$\sigma', \rho' \vdash M \quad (\text{Sol})$$

The term  $M$  inhabits the target type provided that the meta variable instantiations and conditions of the problem are extended by  $\sigma'$  and  $\rho'$ .

We will also use the notion of *refinement*, **Ref**, which specifies how a problem can be refined to a new set of problems:

$$\Delta'_{par}, \Delta_{prf}, \sigma', \rho' \vdash M \quad (\text{Ref})$$

Just as for a solution, the proof term,  $M$ , is an inhabitant assuming the extra instantiations and conditions,  $\sigma'$  and  $\rho'$ . In general,  $M$  contains new meta variables. The new meta variables are divided into parameters,  $\Delta'_{par}$ , and proofs,  $\Delta_{prf} = [\Gamma_i, \rho_i \vdash ?_i : T_i]$ , according to the classification described above. For proof meta variables the information supplied is different from that of parameter meta variables. Instead of the full context, only the extra local variables are given. Moreover, not only the context and type are given but also a set of extra conditions which should be enforced in the corresponding branch of the proof. This is needed since parameter variables are treated differently from proof variables. The distinction is the same as for meta variables – parameters are variables which appear in some type. While proof variables are eliminated on demand, as described above, case splits for parameter variables must precede the proof of its branches as a separate refinement.

Finally, we need to talk about collections of problems, **PrbColl**, and collections of solutions, **SolColl**.

$$\begin{array}{l} \Delta_{par}, \sigma, \Gamma, \rho \vdash \Delta_{prf} \quad (\text{PrbColl}) \\ \sigma', \rho' \vdash \sigma^* \quad (\text{SolColl}) \end{array}$$

A problem collection is a set of common instantiations and a set of common conditions followed by a list of proof meta variables. A corresponding solution collection contains the extra sets of common instantiations and conditions as well as a set of instantiations,  $\sigma^*$ , which gives a term for every proof meta variable in the problem collection.

The following functions describe the proof search algorithm:

```

search : Prb → [Sol]
search (Δpar, σ, Γ, ρ ⊢ T) =
  refs := createRefs (Δpar, σ, Γ, ρ ⊢ T)
  sols := []
  for each (Δ'par, Δ'prf, σ'1, ρ'1 ⊢ M) in refs
    prbcoll := ((Δpar ++ Δ'par), (σ ++ σ'1), (ρ ++ ρ'1) ⊢ Δ'prf)
    solcolls := searchColl prbcoll
    for each (σ'2, ρ'2 ⊢ σ*) in solcolls
      case (compose ((σ'1 ++ σ'2), (ρ'1 ++ ρ'2), M, σ*)) of
        none → sols := sols
        some sol → sols := addSol (sol, sols)
      end case
    end for
  end for
return sols

searchColl : PrbColl → [SolColl]
searchColl (Δpar, σ, ρ ⊢ []) = ([], [] ⊢ [])
searchColl (Δpar, σ, ρ ⊢ ((Γi, ρi ⊢ ?i : Ti) : prbs)) =
  prb := (Δpar, σ, (Γ ++ Γi), (ρ ++ ρi) ⊢ Ti)
  sols := search prb
  solcolls := searchColl (Δpar, σ, ρ ⊢ prbs)
  solcolls' := []
  for each (σ'1, ρ'1 ⊢ M) in sols
    for each (σ'2, ρ'2 ⊢ σ*) in solcolls
      case (combine (σ'1, σ'2, ρ'1, ρ'2)) of
        none → solcolls' := solcolls'
        some σ'cρ'c → solcolls' := (σ'c, ρ'c ⊢ ((?i := M) : σ*)) : solcolls'
      end case
    end for
  end for
return solcolls'

```

The types of the auxiliary functions are the following:

```

createRefs : Prb → [Ref]
addSol : Sol, [Sol] → [Sol]
combine : σ, σ, ρ, ρ → σ, ρ option
compose : σ, ρ, M, σprf → Sol option

```

The function `search` first invokes `createRefs`, which generates the list of refinements which are valid according to the set of refinement rules. Then, for each refinement it compiles a problem collection consisting of its proof meta variables. The collection is passed to `searchColl` which returns a list of solution collections.



For each collection, `compose` is called. This function lifts the parameter instantiations and conditions above the scope of the current refinement. For the instantiations, this means removing the entries which bind a meta variable introduced by the refinement. When removing such an instantiation, the meta variable is substituted for its value in  $M$ . For the conditions, it means checking whether any of the conditioned variables was introduced by the refinement. In that case, the solution would not be valid and the function returns nothing. Otherwise, the values in  $\sigma^*$  are substituted into  $M$  and a solution is returned.

If `compose` returns a solution, then a call to `addSol` adds it to the list of solutions. However, if there is already a better solution present, the new one is discarded. Conversely, if the new solution is better than some old one, that one is discarded. A solution  $A$  is said to be better than a solution  $B$  if  $A$  would combine with a solution  $C$  whenever  $B$  combines with  $C$ . In other words,  $A$  is better than  $B$  when its parameter instantiations and conditions are subsets of those of  $B$ .

Moreover, when adding a solution to the list, its conditions are checked against the conditions of the already present solutions. If there is a collection of solutions which can be combined with respect to instantiations and conditions, and which together discharge one condition, the solutions are merged to a single one. The proof term of this new solution is a case-expression where the case branches correspond to the proof terms of the constituting solutions. As an example, assume that there are two solutions,

$$[h \rightarrow \text{inl } a] \vdash M \quad \text{and} \quad [h \rightarrow \text{inr } b] \vdash N.$$

Then they are merged into the single solution

$$\vdash \text{case } h \text{ of } \{ \text{inl } a \rightarrow M; \text{inr } b \rightarrow N \}.$$

The function `searchColl` first calls `search` to generate the solutions for the first problem in the collection. It then does a recursive call to produce the solution collections for the rest of the problems. For each solution and each solution collection it then invokes `combine`. This function takes a pair of parameter instantiations and a pair of conditions. It merges the instantiations and conditions while checking that no inconsistency appears. Checking this involves comparing terms. A term in an instantiation or in a condition may contain meta variables. Thus, unification is performed rather than comparing and new instantiations may need to be added. If a combination is possible, the combined sets of instantiations and conditions are returned and the result is used to construct a collection which includes a solution for the current problem.

### 3.2 Induction

The tactic which does case split on a variable also adds a locally defined function around the case expression. The function can in a later refinement be invoked as an induction hypothesis. Special information is stored to limit the calls to structural recursion.

Any variable which is of algebraic data type and is a parameter, i.e. appears in a type, is a candidate for the tactic.

We will now give an example. The proof steps are presented as refinements. Unlike the definitions in section 3.1, the meta variable is made explicit in problems and refinements, in order to clarify the structure of the proof search. Types are displayed in their normal form, just as they appear at unification. In applications, some type arguments are omitted to increase readability. Also, in function definitions, the type of some arguments is omitted for the same reason. Variable names essentially correspond to what the tool produces.

The problem is commutativity of addition for natural numbers.

$$[a, b : Nat] \vdash ? : a + b == b + a$$

Addition is defined by recursion on the first argument. The proof search calls the case split tactic, which explores analysis on  $a$ . This gives the following refinement:

$$\begin{aligned} & [[a \rightarrow 0] \vdash ?_b : b == b + 0, \quad [a \rightarrow \mathbf{s} \ a'] \vdash ?_s : \mathbf{s} \ (a' + b) == b + \mathbf{s} \ a'] \vdash \\ ? & := \left( \begin{array}{l} \mathbf{let} \ r \ a \ b : (a + b == b + a) = \mathbf{case} \ a \ \mathbf{of} \ \{0 \rightarrow ?_b; \ \mathbf{s} \ a' \rightarrow ?_s\} \\ \mathbf{in} \ r \ a \ b \end{array} \right) \end{aligned}$$

The local function is given as arguments all parameters which appear in the target type and all hypotheses whose types contain the parameters, in this case parameters  $a$  and  $b$ . The two new proof meta variables have a condition corresponding to each branch.

The base case is solved by induction on  $b$  and appealing to reflexivity, *refl*, and substitutivity for equality, *cong*.

$$\mathit{refl} \ (X : Set) : (x : X) \rightarrow (x == x)$$

$$\mathit{cong} \ (X, Y : Set) : (f : X \rightarrow Y) \rightarrow (x_1, x_2 : X) \rightarrow (x_1 == x_2) \rightarrow (f \ x_1 == f \ x_2)$$

The proof for the base case is constructed by the following refinements:

$$\begin{aligned} & [[b \rightarrow 0] \vdash ?_{bb} : 0 == 0, \quad [b \rightarrow \mathbf{s} \ b'] \vdash ?_{bs} : \mathbf{s} \ b' == \mathbf{s} \ (b' + 0)] \vdash \\ ?_b & := \mathbf{case} \ b \ \mathbf{of} \ \{0 \rightarrow ?_{bb}; \ \mathbf{s} \ b' \rightarrow ?_{bs}\} \\ \vdash ?_{bb} & := \mathit{refl} \ 0 \\ [\vdash ?_p : b' == b' + 0] \vdash ?_{bs} & := \mathit{cong} \ (\lambda x \rightarrow \mathbf{s} \ x) \ b' \ (b' + 0) \ ?_p \\ \vdash ?_p & := r \ 0 \ b' \end{aligned}$$

The first refinement is again generated by the case split tactic. The second and third are generated by the equalities tactics and the last by the normal elimination tactic.

In the step case, the induction hypothesis corresponding to structural recursion on  $a$  is used to rewrite the equality by referring to transitivity, *tran*.

$$\mathit{tran} \ (X : Set) : (x, y, z : X) \rightarrow (x == y) \rightarrow (y == z) \rightarrow (x == z)$$

The refinement is:

$$\begin{aligned} & [\vdash ?_q : \mathbf{s} \ (b + a') == b + \mathbf{s} \ a'] \vdash \\ ?_s & := \left( \begin{array}{l} \mathit{tran} \ (\mathbf{s} \ (a' + b)) \ (\mathbf{s} \ (b + a')) \ (b + \mathbf{s} \ a') \\ (\mathit{cong} \ (\lambda x \rightarrow \mathbf{s} \ x) \ (a' + b) \ (b + a') \ (r \ a' \ b)) \\ ?_q \end{array} \right) \end{aligned}$$

The equalities tactic combines the use of transitivity with the use of substitutivity and symmetry.

For  $?_q$ , case analysis on  $b$  will lead to a solution.

$$[[b \rightarrow 0] \vdash ?_{sb} : \mathbf{s} a' == \mathbf{s} a', [b \rightarrow \mathbf{s} b'] \vdash ?_{ss} : \mathbf{s} (\mathbf{s} (b' + a')) == \mathbf{s} (b' + \mathbf{s} a')] \vdash$$

$$?_q := \left( \begin{array}{l} \mathbf{let} \ r' \ b \ a' : (\mathbf{s} (b + a')) == b + \mathbf{s} a' = \\ \qquad \qquad \qquad \mathbf{case} \ b \ \mathbf{of} \ \{0 \rightarrow ?_{sb}; \ \mathbf{s} \ b' \rightarrow ?_{ss}\} \\ \mathbf{in} \ r' \ b \ a' \end{array} \right)$$

The following refinements complete the proof term:

$$\begin{aligned} \vdash ?_{sb} &:= \mathit{refl} (\mathbf{s} a') \\ [\vdash ?_r : \mathbf{s} (b' + a') == b' + \mathbf{s} a'] \vdash \\ ?_{ss} &:= \mathit{cong} (\lambda x \rightarrow \mathbf{s} x) (\mathbf{s} (b' + a')) (b' + (\mathbf{s} a')) ?_r \\ \vdash ?_r &:= r' b' a' \end{aligned}$$

The equalities tactic generates the first two refinements, while the elimination tactic generates the third by using the induction hypothesis  $(\mathbf{s} (b' + a')) == b' + \mathbf{s} a'$ .

### 3.3 Case on Expression

There is also a tactic for case-on-expression. This tactic looks at the subexpressions of the target type and of the variables in the context. Any data type subexpression which is an application with undefined head position is subject to the tactic. All occurrences of the subexpression are replaced by a fresh variable. Then, case analysis on the new variable is added. This is a special and very useful case of generalisation. Although the occurrences of the subexpression are replaced, new instances may appear at a later stage. Therefore, a proof that the new variable equals the subexpression is supplied.

The following example illustrates the use of case-on-expression. It is about lists and the functions *map*, which is defined recursively in the normal way, and *filter*. In order to allow *filter* reducing in two steps it is defined in terms of *if*.

$$\begin{aligned} \mathit{if} \ \mathbf{true} \ x \ y &= x \\ \mathit{if} \ \mathbf{false} \ x \ y &= y \\ \mathit{filter} \ f \ [] &= [] \\ \mathit{filter} \ f \ (x :: xs') &= \mathit{if} (f x) (x :: \mathit{filter} f xs') (\mathit{filter} f xs') \end{aligned}$$

The reason for defining *filter* this way is that Agda, when normalising a term, only unfolds an application when the definition of the function reduces to something which is not a case expression. This, combined with the fact the first-order unification is used, makes it necessary to define *filter* to reduce in two steps. First it reduces to an *if*-statement when the list is known to be of the form  $x :: xs$ . Then it reduces again when the boolean value of  $(f x)$  is known.

The problem is the following:

$$\begin{aligned} [X, Y : \mathit{Set}, f : X \rightarrow Y, p : Y \rightarrow \mathit{Bool}, xs : \mathit{List} X] \vdash \\ ? : \mathit{filter} p (\mathit{map} f xs) == \mathit{map} f (\mathit{filter} (\lambda x \rightarrow p (f x)) xs) \end{aligned}$$

The proof begins with induction on  $xs$ . In the step case, the goal type reduces to:

$$\begin{aligned} & \text{if } (p (f x)) (f x :: \text{filter } p (\text{map } f xs')) (\text{filter } p (\text{map } f xs')) == \\ & \quad \text{map } f (\text{if } (p (f x)) (x :: \text{filter } (\lambda x \rightarrow p (f x)) xs') (\text{filter } (\lambda x \rightarrow p (f x)) xs')) \end{aligned}$$

The tactic identifies  $p (f x)$  for generalisation and case analysis. The occurrences are replaced by the variable  $px$ . We will not give all refinements, but simply present the final generated proof term:

```

let r xs : filter p (map f xs) == map f (filter (λx → p (f x)) xs)
  [] → refl []
  x :: xs' →
    let g px (peq : px == p (f x)) :
      if px (f x :: filter p (map f xs')) (filter p (map f xs')) ==
        map f (if px (x :: filter (λx → p (f x)) xs') (filter (λx → p (f x)) xs'))
      = case px of
        true → cong (λy → f x :: y) (filter p (map f xs'))
          (map f (filter (λx → p (f x)) xs')) (r xs')
        false → r xs'
    in g (p (f x)) (refl (p (f x)))
in r xs

```

The case-on-expression tactic generates a refinement where the local function  $g$  is defined to **case**  $px$  **of**  $\{\mathbf{true} \rightarrow ?_t; \mathbf{false} \rightarrow ?_f\}$ . Each branch is then solved by the equalities tactic.

### 3.4 Generalisation

The generalisation tactic recognises two cases; generalise subexpression and generalise apart. Generalise apart means replacing multiple occurrences of a single variable with two different variables. Generalising subexpression means picking a subexpression and replacing it with a new variable. It is only applied for subexpressions which occur at least twice, as opposed to the more restricted generalisation introduced by the case-on-expression tactic.

Generalise subexpression seems to be the more useful of the two. We have only made use of generalise apart for simple problems like  $2 \cdot (\mathbf{s } n) == \mathbf{s } (\mathbf{s } (2 \cdot n))$ , where multiplication is defined in terms of addition by recursion on the first argument.

We give an example to illustrate the strengthening of the induction hypothesis using generalise subexpression; reversing a list twice.

$$[X : \text{Set}, xs : \text{List } X] \vdash ? : \text{rev } (\text{rev } xs) == xs$$

Reversing a list is defined in terms of concatenating two lists.

$$\begin{aligned} [] ++ ys &= ys \\ (x :: xs) ++ ys &= x :: (xs ++ ys) \\ \text{rev } [] &= [] \\ \text{rev } (x :: xs) &= \text{rev } xs ++ (x :: []) \end{aligned}$$

The proof begins by induction on  $xs$ . In the step case, where  $xs \rightarrow x :: xs'$ , the goal is rewritten using the induction hypothesis and transitivity, yielding the type:

$$\text{rev } (\text{rev } xs' ++ (x :: [])) == x :: \text{rev } (\text{rev } xs')$$

Here,  $(\text{rev } xs')$  is generalised to a new variable,  $xs''$ . The proof then follows by induction on  $xs''$ . The final proof term is:

```

let r xs : (rev (rev xs) == xs) = case xs of
  [] → refl []
  x :: xs' → tran (rev (rev xs' ++ (x :: []))) (x :: rev (rev xs')) (x :: xs')
    (let g xs'' : rev (xs'' ++ (x :: [])) == x :: rev xs'' = case xs'' of
      [] → refl (x :: [])
      x' :: xs₀ → cong (λx → x ++ (x' :: [])) (rev (xs₀ ++ (x :: [])))
        (x :: rev xs₀) (g xs₀)
    in g (rev xs'))
  (cong (λy → x :: y) (rev (rev xs'))) xs' (r xs')
in r xs

```

### 3.5 Extension of First-Order Unification

The tool is based on first-order unification. Restricted to this, when unification is invoked, the tool simply normalises the terms and first-order unification is applied. The strength of this is obviously limited in a higher-order setting. To improve this without making the tool too inefficient, we have added an extension which, in a sense, does first-order unification for function meta variables. Before the first-order mechanism is called, the terms are examined. Any occurrence of a function application where the head is a meta variable is replaced by a fresh meta variable. Then the usual first-order unification is called. If it was successful, all syntactically possible ways to construct a  $\lambda$ -abstraction and arguments are generated. The restrictions are that the resulting application should  $\beta$ -reduce to the correct term and that the arguments should be type correct.

We illustrate this by a simple example. Consider substitutivity for natural numbers:

$$[P : \text{Nat} \rightarrow \text{Set}, x_1, x_2 : \text{Nat}] \vdash ? : x_1 == x_2 \rightarrow P x_1 \rightarrow P x_2$$

The tool will generate a proof which starts with induction on  $x_1$  followed by induction on  $x_2$ :

$$\begin{aligned}
 ? & := \left( \begin{array}{l} \text{let } r (P : \text{Nat} \rightarrow \text{Set}) (x_1, x_2 : \text{Nat}) (p : x_1 == x_2) (q : P x_1) : P x_2 \\ \quad = \text{case } x_1 \text{ of } \{0 \rightarrow ?_b; s x'_1 \rightarrow ?_s\} \\ \text{in } r P x_1 x_2 \end{array} \right) \\
 ?_s & := \text{case } x_2 \text{ of } \{0 \rightarrow ?_{sb}; s x'_2 \rightarrow ?_{ss}\}
 \end{aligned}$$

For  $?_{ss}$  the problem is:

$$[\dots, p : (s x'_1 == s x'_2), q : P(s x'_1)], [x_1 \rightarrow s x'_1, x_2 \rightarrow s x'_2] \vdash ?_{ss} : P (s x'_2)$$

Applying  $r$  will be considered by the elimination tactic. The application  $(r \ ?_P \ ?_{x_1} \ ?_{x_2} \ ?_p \ ?_q)$  has the type  $(?_P \ ?_{x_2})$  where  $?_P : (Nat \rightarrow Set)$  and  $?_{x_2} : Nat$ . The unification problem to consider is thus:

$$P \ (\mathbf{s} \ x'_2) \stackrel{u}{=} \ ?_P \ ?_{x_2}$$

The application on the right hand side is replaced by a fresh meta variable,  $?'$ . The standard first-order unification returns the unifier  $[?' := P \ (\mathbf{s} \ x'_2)]$ . Now the extended unification produces a unifier for each possible way to partition the expression  $(P \ (\mathbf{s} \ x'_2))$  onto the function,  $?_P$ , and its argument,  $?_{x_2}$ . These are the possibilities:

$$\begin{aligned} &\{?_P = \lambda x \rightarrow P \ (\mathbf{s} \ x'_2)\} \\ &\{?_P = \lambda x \rightarrow P \ x, \ ?_{x_2} = \mathbf{s} \ x'_2\} \\ &\{?_P = \lambda x \rightarrow P \ (\mathbf{s} \ x), \ ?_{x_2} = x'_2\} \end{aligned}$$

The third unifier leads to a terminating proof term.

## 4 Results

We have used the number of generated refinements as a hardware independent measure for the tool's effort when solving a problem. On a normal desktop computer, the number of generated refinements per second is around 500. The problems presented in sections 3.2, 3.3 and 3.4 take between 50 and 100 refinements to solve.

We will now present some more difficult examples which demonstrate the limits of the tool. A larger collection of problems with proofs generated by the tool can be downloaded from [8].

The first three examples are problems in the context of list sorting. The propositions are about a list being sorted, *Sort*, a list being a permutation of another list, *Perm*, and an element being a member of a list, *Member*. The functions are defined as follows:

$$\begin{aligned} \textit{Sorted} \ [] &= \top \\ \textit{Sorted} \ (x :: []) &= \top \\ \textit{Sorted} \ (x :: y :: xs) &= x \leq y \wedge \textit{Sorted} \ (y :: xs) \\ \textit{Perm} \ xs \ ys &= \forall x. \textit{count} \ x \ xs == \textit{count} \ x \ ys \\ \textit{Member} \ y \ [] &= \perp \\ \textit{Member} \ y \ (x :: xs) &= x == y \vee \textit{Member} \ y \ xs \\ \textit{count} \ y \ [] &= 0 \\ \textit{count} \ y \ (x :: xs) &= \textit{if} \ (eq \ x \ y) \ (\mathbf{s} \ (\textit{count} \ y \ xs)) \ (\textit{count} \ y \ xs) \end{aligned}$$

List concatenation and the filter function will also appear. They are defined in the normal way. The relations ' $\leq$ ' and ' $>$ ' will be used both to denote the boolean functions and the correspond propositions.

The first proposition claims commutativity of list concatenation with respect to permutation:

$$\textit{Perm} \ (xs ++ ys) \ (ys ++ xs)$$

The tool can be set to either look for lemmas among all the global definitions or to just use the local variables. The former mode is often too inefficient. When not including the globals, the user can give a list of lemmas as hints. The problem above is solved by giving  $(a, b : \mathbf{Nat}) \rightarrow a + \mathbf{s} b == \mathbf{s} (a + b)$  as a hint and it takes 4653 refinements. The proof consists of induction on  $xs$  and  $ys$ , a few case-on-expressions and some equality manipulation.

The tool has no rewriting system for equalities. The equalities tactic modifies equalities in a quite undirected way. Due to this, a lot of effort is often spent on finding relatively small proofs for problems involving equalities.

Another property of the equality reasoning is that simple lemmas are often needed, like the one for the problem above. Although the tool can easily solve such lemmas separately, it cannot always invent them as part of another proof. This is because transitivity is only applied for an equality when there is already a known fact which can be used to rewrite either the left or the right hand side. The tool never invents an intermediate value.

The next two examples are lemmas for a correctness proof for a quicksort algorithm.

$$\begin{aligned} & Perm\ xs\ (filter\ (x\ >) xs ++ filter\ (x\ \leq)\ xs) \\ & Sorted\ xs \rightarrow Sorted\ ys \rightarrow ((x : X) \rightarrow Member\ x\ xs \rightarrow |x \leq a|) \\ & \rightarrow ((x : X) \rightarrow Member\ x\ ys \rightarrow |a \leq x|) \rightarrow Sorted\ (xs ++ (a :: ys)) \end{aligned}$$

The first of these is solved in 1173 refinements with two hints, namely the same as in the previous examples as well as the proposition  $count\ x\ (xs ++ ys) == count\ x\ xs + count\ x\ ys$ . The second example is solved in 359 refinements with no hints. The proof includes double case analysis on  $xs$ .

Next example is the problem to show associativity for addition of integers defined in the following way:

$$Int = \mathbf{data}\ Zer\ |\ Pos\ (n : \mathbf{Nat})\ |\ Neg\ (n : \mathbf{Nat})$$

The proposition is

$$(p + q) + r == p + (q + r).$$

The proof takes 11285 refinements and no hint is needed.

Finally, consider the problem

$$(n : \mathbf{Nat}) \rightarrow \exists\ Nat\ (\lambda(m : \mathbf{Nat}) \rightarrow (n == 2 \cdot m) \vee (n == \mathbf{s} (2 \cdot m))).$$

To solve this the tool needs  $2 \cdot \mathbf{s} n == \mathbf{s} (\mathbf{s} (2 \cdot n))$  as a hint. The problem is a very simple example of a program carrying proof, namely division by two.

The tool has a few settings, one of which should be mentioned. There is a value defining the maximum number of nested case-splits applied to a variable. In our examples, this is set to two, which is enough for all problems we have tested.

## 5 Conclusion and Future Work

In our opinion, the efficiency of the tool is good enough for it to be useful in a proof assistant. It can definitely save the user time. If one would consider using the tool for larger problems and allowing it more time, we think that it would perform poorly. More advanced techniques to reduce the search space would be needed. Also, since the tool is written in Haskell, it would probably suffer from the automatic memory management.

One should also ask whether the tool is versatile enough to be of practical use. We think this is the case. The tool has some fundamental restrictions, such as first-order unification. But, apart from this, our goal has been to construct a general proof search algorithm.

Instead of writing a tool which performs a proof search directly, another approach is to translate the problem and solve it with a separate first-order solver. This is currently investigated by e.g. Abel, Coquand and Norell with promising results [1]. In this approach, the problems are restricted to first-order logic and no proof term is recovered. On the other hand, it allows making use of the many highly optimized existing first-order solvers. We believe that this could be combined with our work to create a more powerful tool. Different parts of a proof could be addressed by the two components, e.g. induction by the internal proof-search and equality reasoning by the external prover. Equality reasoning has turned out to be a major bottleneck for our tool.

One feature of the tool which may not have been such a good idea is the on-demand elimination of data type objects. This adds the necessity of annotating solutions with conditions and all administration that it brings along. Another thing is the parameter/proof classification which seems a bit to rigid.

One way to continue the work could be to restart with a term synthesis algorithm based on higher-order unification, such as the one presented by Dowek [6]. This would then be enriched by first-order unification, which would serve as a short-cut in the proof search. We believe that also in a system for higher-order logic, most subterms can be resolved by first-order unification, and that it would be beneficial to have a proof search that is biased in that direction.

Another interesting issue is to deal with a dense presence false subproblems. A false subproblem may occur already when applying ordinary *modus ponens*, but if we would add abilities for lemma speculation and stronger generalisation, most subproblems would be false. Maybe one could then improve efficiency by trying to prove the negation of a possibly false subproblem in parallel. If a proof of the negation is found, the corresponding branch can be pruned.

## References

1. Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. Submitted, 2005.
2. Peter B. Andrews. Classical type theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 15, pages 965–1007. Elsevier Science, 2001.



3. Catharina Coquand. The AGDA Proof System Homepage. <http://www.cs.chalmers.se/catarina/agda/>, 1998.
4. Christina Cornes. *Conception d'un langage de haut niveau de représentation de preuves: Récurrence par filtrage de motifs, Unification en présence de types inductifs primitifs, Synthèse de lemmes d'inversion*. PhD thesis, Université Paris 7, 1997.
5. Gilles Dowek. A complete proof synthesis method for the cube of type systems. *J. Logic and Computation*, 3(3):287–315, 1993.
6. Gilles Dowek. Higher-order unification and matching. In Alan Robinson and Andrei Voronkov, editors, *Handbook of automated reasoning*, volume 2, chapter 16, pages 1009–1062. Elsevier Science, 2001.
7. Peter Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. In *Logical frameworks*, pages 280–306. Cambridge University Press, New York, NY, USA, 1991.
8. Fredrik Lindblad. Agsy problem examples. [http://www.cs.chalmers.se/~frelindb/Agsy\\_examples.tgz](http://www.cs.chalmers.se/~frelindb/Agsy_examples.tgz), 2004.
9. Lena Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 1994.
10. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
11. Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for lf. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300. Springer-Verlag LNCS, July 1998.
12. Tanel Tammet and Jan Smith. Optimized encodings of fragments of type theory in first-order logic. *Journal of Logic and Computation*, 8, 1998.

# Surreal Numbers in Coq

Lionel Elie Mamane

Institute for Computing and Information Sciences,  
Radboud University Nijmegen  
lionelm@cs.ru.nl

**Abstract.** Surreal Numbers form a totally ordered (commutative) Field, containing copies of the reals and (all) the ordinals. I have encoded most of the Ring structure of surreal numbers in Coq. This encoding relies on Aczel’s encoding of set theory in type theory.

This paper discusses in particular the definitional or proving points where I had to diverge from Conway’s or the most natural way, like separation of simultaneous induction-recursion into two inductions, transforming the definition of the order into a mutually inductive definition of “at most” and “at least” and fitting the rather complicated induction/recursion schemes into the type theory of Coq.

## 1 Introduction

Surreal numbers, presented by Conway in [9], form a class of numbers containing both the real numbers and all ordinals, in a totally ordered (commutative) Field structure. This, in contrast with the usual four-level construction<sup>1</sup> of  $\mathbb{R}$ , happens with only one (inductive) definition and one addition and one multiplication. Surreal numbers can be seen as filling the holes between the ordinals in the same meaning as  $\mathbb{R}$  fills the holes between the natural numbers.

I have formalised the notion, and nearly all of its commutative Ring structure<sup>2</sup>, in Coq. One of the motivations for doing so was to test, through Coq, Type Theory based proof assistants on a set-theoretically defined notion. See section 6.2 for more on that. Unless otherwise specified, “number” refers to “surreal number” throughout the rest of this article.  $On$  is the collection of all ordinals,  $No$  is the collection of all surreal numbers (to be defined) and pCIC is the Predicative Calculus of Inductive Constructions, the type theory that Coq implements.

We will first briefly describe Surreal Numbers, as presented by Conway. We will then present their encoding in Coq, discussing the problematic or otherwise interesting points of this development:

- The natural presentation of Surreal Numbers uses induction-recursion, a feature absent from most current theorem provers.

---

<sup>1</sup>  $\mathbb{N}$  is constructed from Set Theory,  $\mathbb{Z}$  from  $\mathbb{N}$ ,  $\mathbb{Q}$  from  $\mathbb{Z}$  and  $\mathbb{R}$  from  $\mathbb{Q}$  (or  $\mathbb{Q}^+$  from  $\mathbb{N}$ ,  $\mathbb{R}^+$  from  $\mathbb{Q}^+$  and  $\mathbb{R}$  from  $\mathbb{R}^+$ ), giving four formally different additions and multiplications that have to be shown equivalent, etc.

<sup>2</sup> Associativity of multiplication is currently missing.

- The induction and recursion schemes used in some proofs and the definitions of some operations are somewhat delicate to fit into the pCIC. However, they all fall under a generic class, which I have not formalised yet.
- The definitions show a high level of (anti-)symmetry, but the proofs don't satisfyingly take advantage of it. In [9], (anti-)symmetry is exploited by proving only one case and declaring the other cases “similar”.
- Problems introduced by the lack of universe polymorphism, in particular that the encoding can cover only an (arbitrary large) subset of the surreal numbers, not the full proper class. Furthermore, using a product to construct surreal numbers precludes from considering a pair whose first (or second) element is the class of surreal numbers and vice-versa.

## 2 Surreal Numbers

### 2.1 Definition

**Definition 1 (Surreal Numbers,  $\geq$ ).** *A surreal number  $x$  is a pair of arbitrary sets of surreal numbers  $L_x$  (the left of  $x$ ) and  $R_x$  (the right of  $x$ ), such that*

$$\forall x_l \in L_x, \neg \exists x_r \in R_x, x_l \geq x_r$$

where

$$x \geq y \stackrel{\text{def}}{\iff} (\neg \exists x_r \in R_x, y \geq x_r) \wedge (\neg \exists y_l \in L_y, y_l \geq x)$$

Such an  $x$  is denoted  $\{L_x \mid R_x\}$  and reciprocally,  $x$  denotes  $\{L_x \mid R_x\}$ , and the same for  $y$ . A left of  $x$  is an element of the left of  $x$  and denoted  $x_l$ ; the use of this notation will by itself declare  $x_l$  to be an element of  $L_x$ , this will not always be repeated. Similarly, a right of  $x$  (an element of the right) is denoted  $x_r$ . An option of  $x$  is a left or a right of  $x$ .

**Definition 2.** *Equality (denoted by  $=$  in infix position) is defined as the equivalence relation corresponding to the pre-order  $\geq$ , namely  $x = y \stackrel{\text{def}}{\iff} x \geq y \wedge y \geq x$ . The usual notations for the order are respected:  $x \leq y \stackrel{\text{def}}{\iff} y \geq x$ ,  $x > y \stackrel{\text{def}}{\iff} x \geq y \wedge x \neq y$  and  $x < y \stackrel{\text{def}}{\iff} y > x$ .*

*Set-theoretic equality (double inclusion) is renamed identity and denoted by  $\equiv$  in infix position.  $\equiv$  is strictly more discriminating than  $=$ .*

A surreal number  $x$  is to be interpreted as the “simplest” surreal number lying strictly between its left and its right, i.e.  $(\forall x_l, x_l < x) \wedge (\forall x_r, x < x_r)$ :

$$\begin{array}{ccc} L_x & & R_x \\ \hline & x & \hline \end{array}$$

This explains the necessity of the  $\neg x_l \geq x_r$  condition in the definition on an intuitive level: For a number to lie between  $L_x$  and  $R_x$ , there better not be an element of  $L_x$  that is bigger than or equal to an element of  $R_x$ ! If there were, it would contradict the pre-order property of  $\geq$ : one would have  $x_l < x < x_r \leq x_l$ .

A similar intuitive reasoning explains the necessity of the two conditions in the definition of  $\geq$ . If one had an  $x_r$  such that  $y \geq x_r$ , then, because  $x_r > x$ , one would have  $y > x$  and thus certainly not  $x \geq y$ . If our interpretation is to hold, then  $\neg \exists x_r \in R_x, y \geq x_r$  must be a necessary condition for  $x \geq y$  to hold. Similarly,  $y_l \geq x$  would also bring  $y > x$ .

The interpretation given here is similar to Dedekind cuts taken inductively: The Dedekind cut construction of the reals goes from one level (namely  $\mathbb{Q}$ ) to the next ( $\mathbb{R}$ ); surreal numbers are constructed by making cuts again from the results of cuts, thus constructing a tower of levels. These levels are numbered with the ordinals. Formally:

**Definition 3 (birth date).** *The birth date  $\rho(x)$  of a surreal number  $x$  is the smallest ordinal strictly greater than the birth dates of its options, i.e. the supremum of the successors of the birth dates of its options:*

$$\rho(\{L \mid R\}) \stackrel{\text{def}}{=} \sup_{i \in L \cup R} (\rho(i) + 1)$$

This is all best seen through examples:

### 2.2 Well-Known Subcollections of $No$

$On$  (including  $\mathbb{N}$ ) is injected into  $No$  by the  $\varphi$  defined recursively by

$$\varphi(o) \stackrel{\text{def}}{=} \{\{\varphi(n) : n \in On, n < o\} \mid \emptyset\}$$

Let's note that for any ordinal  $o$ ,  $\rho(\varphi(o)) = o$ . Dyadic real numbers<sup>3</sup> are injected into  $No$  by extending  $\varphi$  with

$$\varphi\left(\frac{x+y}{2}\right) \stackrel{\text{def}}{=} \{\{\varphi(x)\} \mid \{\varphi(y)\}\}$$

for  $x$  and  $y$  consecutive<sup>4</sup> dyadic real numbers. The birth dates of the surreal numbers constructed by this extension are natural numbers. To cover non-dyadic real numbers,  $\varphi$  can be extended with

$$\begin{aligned} \varphi(X) &\stackrel{\text{def}}{=} \{\varphi(x) : x \in X\} \text{ for } X \text{ a set of dyadic real numbers} \\ \varphi(x) &\stackrel{\text{def}}{=} \{\varphi(L) \mid \varphi(R)\} \text{ for } x \text{ a non-dyadic real number and} \\ &\quad L, R \text{ sets of dyadic real numbers such that} \\ &\quad (\forall l \in L, l < x) \wedge (\forall r \in R, x < r) \wedge \\ &\quad (\forall \varepsilon \in \mathbb{R}_0^+, \exists (l, r) \in L \times R, x - l < \varepsilon \wedge r - x < \varepsilon) \end{aligned}$$

The non-unique choice of  $L$  and  $R$  above provides us with examples of surreal numbers that are equal, but not identical.  $\varphi$ , adequately restricted, is an ordered

<sup>3</sup> Real numbers that admit a finite development in base 2.

<sup>4</sup> Let  $(n, n', p, p') \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \times \mathbb{N}$ , such that  $p = 0 \vee n$  is odd and  $p' = 0 \vee n'$  is odd.  $x := \frac{n}{2^p}$  and  $y := \frac{n'}{2^{p'}}$  are consecutive if and only if  $x < y$  and for all dyadic real numbers  $z = \frac{n''}{2^{p''}}$  such that  $x < z < y$ , we have  $p'' > p \wedge p'' > p'$ .

field morphism. The birth date of the surreal numbers constructed by this extension is  $\omega$ , but  $\omega$  is *not* a clean delimitation of  $\mathbb{R}$ -in- $No$ :  $\varphi(\omega)$  also has birth date  $\omega$ , as does  $\varepsilon \stackrel{\text{def}}{=} \{\{0\} \mid \{\frac{1}{2^n} : n \in \mathbb{N}\}\}$ .  $\varepsilon$  is smaller than any strictly positive real, yet still strictly positive. Actually, the existence of such surreal numbers was to be expected: in a totally ordered field, if a number is very big (bigger than any strictly positive real), then its inverse is very small (smaller than any strictly positive real). And  $\varepsilon$  is exactly that:  $\varepsilon \cdot \varphi(\omega) = \varphi(1)$  holds.

### 2.3 In Coq

The definition of surreal numbers given above mutually defines a collection inductively and a predicate over this collection recursively. It may not be clear to all readers that this is sound mathematical practice and indeed, Conway did not feel authorised to leave such a construction in his “precise treatment”. While the validity of such constructions has since been studied for type theory ([11]), and implemented in some rare proof assistants (such as Lego), Coq does not allow such constructions and I thus followed the same several-step construction as Conway, namely first define *games* by simply dropping the order-condition from the definition of surreal numbers<sup>5</sup>, then define the order relation, which is then a partial preorder, and from that define surreal numbers as a subcollection of games.

**Definition 4 (Games).** *A game  $g$  is a pair of arbitrary sets of games  $L_g$  and  $R_g$  (the left and right of  $g$ ).*

**Definition 5 ( $\geq$  on games).**

$$x \geq y \stackrel{\text{def}}{\iff} (\neg \exists x_r \in R_x, y \geq x_r) \wedge (\neg \exists y_l \in L_y, y_l \geq x)$$

**Definition 6 (Surreal numbers as games).** *A surreal number  $x$  is a game such that all its options are surreal numbers and  $\forall x_l \in L_x, \neg \exists x_r \in R_x, x_l \geq x_r$ .*

Let’s note that games<sup>6</sup> could as well have been named bi-sets: They form a notion of sets with two (left and right) element-of relations; the “normal” sets can be injected into bi-sets by leaving the right empty. It will thus not be surprising that my encoding of surreal numbers follows Aczel’s encoding of set theory in type theory ([1, 2, 3]), as adapted for Coq in [21],

$$\text{Inductive Ens} : \text{Type}_{i+1} := \text{sup} : \forall A : \text{Type}_i, \forall f : (A \rightarrow \text{Ens}), \text{Ens}$$

and not what is called sets in the Coq standard library. The first argument of the constructor of `Ens`, `sup`, is an index type: when  $a$  ranges over  $A$ ,  $(f\ a)$  ranges over the set represented by  $(\text{sup } A\ f)$ , possibly with repetitions. In other words,  $(\text{sup } A\ f)$  is  $\{f(a) : a \in A\}$ .

<sup>5</sup> We reuse the notations and nomenclature introduced for surreal numbers for games.

<sup>6</sup> This name stems from game-theoretic interpretations of the structure and operations, not presented here.

To form the bisets, I add a second index type and function, for the second element-of relation.

**Inductive Game** : Type<sub>*i*+1</sub> :=  
 Gcons :  $\forall \text{LI, RI} : \text{Type}_i, \forall \text{Lf} : (\text{LI} \rightarrow \text{Game}), \forall \text{Rf} : (\text{RI} \rightarrow \text{Game}), \text{Game}$

The index types of the left and right need to be different to deal with cases where one, but not both, is empty; the left / right is empty if and only if its index type is. Let's note that with this definition, identity is not the Coq (intentional) equality, but a strictly weaker equivalence relation.

A naive definition of the order

**Inductive**  $\leq$  : Game  $\rightarrow$  Game  $\rightarrow$  Prop :=  
 lte\_cons :  $\forall x, y : \text{Game}, (\forall x_l : L_x, \neg y \leq x_l) \rightarrow (\forall y_r : R_y, \neg y_r \leq x) \rightarrow x \leq y$

is not accepted by Coq because of the non-positive (under scope of a negation) occurrences of  $\leq$  in the type of its constructor. This is solved by a method inspired by [15, 16], namely defining  $\leq$  and  $\triangleleft \leftrightarrow \lambda x y. \neg y \leq x$  mutually inductive:

**Inductive**  $\leq$  : Game  $\rightarrow$  Game  $\rightarrow$  Prop :=  
 lte\_cons :  $\forall x, y : \text{Game}, (\forall x_l : L_x, x_l \triangleleft y) \rightarrow (\forall y_r : R_y, x \triangleleft y_r) \rightarrow x \leq y$   
**Inductive**  $\triangleleft$  : Game  $\rightarrow$  Game  $\rightarrow$  Prop :=  
 ngte\_cons<sub>r</sub> :  $\forall x, y : \text{Game}, (\exists x_r : R_x, x_r \leq y) \rightarrow x \triangleleft y$   
 ngte\_cons<sub>l</sub> :  $\forall x, y : \text{Game}, (\exists y_l : L_y, x \leq y_l) \rightarrow x \triangleleft y$

This definition is proven to be (non-constructively) equivalent to the former. The transitivity of  $\leq$  is also proven non-constructively, thereby tainting a large part of the development.

### 3 Operations and Induction

#### 3.1 Addition

**Notation 1.** *When using the  $x_l$  or  $x_r$  notations, the universal quantification or union over  $L_x$  or  $R_x$  will be left implicit and the braces around singletons left out. In a  $\{ \mid \}$  construction, the different components of the left or right will be separated by commas. For example, for  $x, y$  games and  $f$  a function  $\text{Game} \rightarrow \text{Game}$ ,  $\{f\ x_r, y_l \mid f\ x_l\}$  will be a lighter and shorter notation for  $\{\{f\ x_r : x_r \in R_x\} \cup L_y \mid \{f\ x_l : x_l \in L_x\}\}$ .*

Addition is defined recursively as

$$x + y := \{x_l + y, x + y_l \mid x_r + y, x + y_r\}$$

The terms appearing in  $x+y$  can be explained on an intuitive basis: The final goal is to construct an ordered field, which includes compatibility of addition with the order. In particular, for  $x, y, z$  surreal numbers, one wants  $x < y \rightarrow x+z < y+z$ . As  $x_l < x$  and  $x < x_r$ ,  $x_l + y < x + y$  and  $x + y < x_r + y$  are necessary conditions.

It is thus natural to force these to be true by putting these into the left and right, respectively, of  $x + y$ . Similarly for  $x + y_l$  and  $x + y_r$ .

A straightforward encoding of the definition

```

Fixpoint GPlus (x y : Game) : Game :=
  case x of ⟨xLI, xRI, xLf, xRf⟩ ⇒
  case y of ⟨yLI, yRI, yLf, yRf⟩ ⇒
  Gcons (xLI + yLI) (xRI + yRI)
  (λi : (xLI + yLI).case i of
    xli : xLI ⇒ GPlus (xLf xli) y
    yli : yLI ⇒ GPlus x (yLf yli))
  (λi : (xRI + yRI).case i of
    xri : xRI ⇒ GPlus (xRf xri) y
    yri : yRI ⇒ GPlus x (yRf yri))

```

is not well-typed in the pCIC: the recursion scheme used, while well-founded, is not recognised as such (it is not *guarded*; Coq accepts only guarded recursive definitions as a way to enforce termination), essentially because no single argument can justify the well-foundedness by itself: they both occur fully (not as a structurally smaller term) in the recursive calls. This is addressed by decomposing the recursion on  $x$  and the recursion on  $y$  in two different functions,  $+_x$  and  $+_x = \lambda y. x + y$ :

$$x + y := \text{let } +_x z := \{x_l + z, +_x z_l \mid x_r + z, +_x z_r\} \text{ in } (+_x y)$$

which, encoded in the pCIC, gives:

```

Fixpoint GPlus (x y : Game) {struct x} : Game :=
  case x of ⟨xLI, xRI, xLf, xRf⟩ ⇒
  let GPlusAux := (fix GPlusAux (z : Game) : Game
    case z of ⟨zLI, zRI, zLf, zRf⟩ ⇒
    Gcons (xLI + zLI) (xRI + zRI)
    (λi : (xLI + zLI).case i of
      xli : xLI ⇒ GPlus (xLf xli) z
      zli : zLI ⇒ GPlusAux (zLf zli))
    (λi : (xRI + zRI).case i of
      xri : xRI ⇒ GPlus (xRf xri) z
      zri : zRI ⇒ GPlusAux (zRf zri)))
  in (GPlusAux y)

```

$+_x$  is recognised as terminating because its only argument decreases in every recursive call, and  $+$  because its *first* argument always does.

### 3.2 Induction

The same issue arises – as the parallel between induction and recursion suggests – in the proofs of induction principles used to prove properties of the objects, but that’s not all: these induction principles also tend to “shuffle around” variables, like in this example:

$$\begin{aligned} \forall P : \text{Game} \rightarrow \text{Game} \rightarrow \text{Prop}, \\ (\forall x, y : \text{Game}, (\forall y_l, (P \ y_l \ x)) \rightarrow (\forall x_r, (P \ y \ x_r)) \rightarrow (P \ x \ y)) \rightarrow \\ \forall x, y : \text{Game}, (P \ x \ y) \end{aligned}$$

This is not *the* induction principle that Coq gives “for free” when defining the Game type. The parallel recursion scheme, whose well-foundedness proves the above induction scheme, is:

$$\lambda P. \lambda H. \text{ let } (f \ x \ y) := (H \ x \ y \ (\lambda y_l. f \ y_l \ x) \ (\lambda x_r. f \ y \ x_r)) \text{ in } f$$

Again, this scheme is well-founded, but this escapes the pCIC. The pCIC works by comparing the  $n^{\text{th}}$  argument in the recursive calls to the  $n^{\text{th}}$  formal parameter for some  $n$ . The  $n^{\text{th}}$  argument in the recursive calls must then be structurally smaller than the  $n^{\text{th}}$  formal parameter. Here, not only for any choice of  $n$  one of the recursive calls yields a whole argument, but the argument (or structurally smaller term) it yields is the *wrong* one and incomparable to the parameter being considered. E.g. for  $n := 0$ , the formal parameter is  $x$ , but the recursive calls have  $y_l$  and  $y$  at this position. This is solved by unfolding the definition of  $f$  in the recursion scheme until the arguments are in the right order again (which always eventually happens). In the example, this means one more time:

$$\begin{aligned} \lambda P. \lambda H. \text{ let } (f \ x \ y) := H \ x \ y \ (\lambda y_l. H \ y_l \ x \ (\lambda x_{l1}. f \ x_{l1} \ y_l) \ (\lambda y_{l7}. f \ x \ y_{l7})) \\ (\lambda x_r. H \ y \ x_r \ (\lambda x_{r1}. f \ x_{r1} \ y) \ (\lambda y_r. f \ x_r \ y_r)) \\ \text{ in } f \end{aligned}$$

Combined with the previous method, this permits us to prove the induction scheme correct in Coq. This technique works for all induction principles needed, but, for more complex principles, gives rise to rather large proofs<sup>7</sup>, the size being here the amount of text the user has to type. Interestingly enough, we are also essentially writing the lambda-term that is the proof of the induction principle directly, without assistance from the tactics mechanism of Coq. Furthermore, it seems very much that the technique can be automated (and I intend to do so in the future) and would then cover the whole range of permuting inductions based on the structural order (where the  $\lll$  below is “structurally smaller than”):

**Definition 7 (Permuting inductions).** For  $\lll$  a well-founded (strict) order with respect to equality  $\equiv$ , for  $n$  a natural number,  $P$  a predicate with  $n$  parameters, a permuting induction is an induction scheme  $S$  of the following form:

- $S$  is
 
$$(\forall(x_0, \dots, x_{n-1}), H) \rightarrow (\forall(x_0, \dots, x_{n-1}), P(x_0, \dots, x_{n-1}))$$
- $H$  is of the form

$$P(t_{0,0}, \dots, t_{0,n-1}) \rightarrow \dots \rightarrow P(t_{p-1,0}, \dots, t_{p-1,n-1}) \rightarrow P(x_0, \dots, x_{n-1})$$

<sup>7</sup> An example of 100 lines is in the development; it is an induction scheme over three variables, the variables undergoing a circular permutation in the recursive call.



–  $\forall i < p$ ,  $\sigma_i$  is a permutation of  $\{e \in \mathbb{N}, e < n\}$ , and

$$\forall j, t_{i,j} \lll x_{\sigma_i(j)} \vee t_{i,j} \equiv x_{\sigma_i(j)}$$

and

$$\forall i, \exists j, t_{i,j} \lll x_{\sigma_i(j)}$$

I have not formalised the notion of permuting induction yet. Its formalisation may give rise to a direct proof rendering the technique described here (and its automation) unnecessary for this work.

A survey of techniques to handle general recursion in Coq, presented in [6, 5, 7, 4], didn't help. The techniques explained there “reduce” the problem to another one (such as totality of a certain predicate or well-foundedness of certain relation), which are essentially equivalent and equally hard to prove in Coq.

I since tried similar things in some other proof assistants, and none of their released “official” versions seem to be able to handle these kinds of induction and recursion schemes in a totally straightforward way, although extensions that could make the situation better are under study.

### 3.3 Opposition and Multiplication

Negation and subtraction are defined by

$$\begin{aligned} -x &:= \{-x_r \mid -x_l\} \\ x - y &:= x + (-y) \end{aligned}$$

These definitions do not pose any particular problem in the pCIC and can be explained intuitively: If  $x$  sits between  $x_l$  and  $x_r$ , then (in an ordered field, where addition is compatible with the order)  $-x$  is between  $-x_r$  and  $-x_l$ . Note that 0 (that is  $\{ \mid \}$ ) is a fixpoint of negation. The expected properties (such as  $x - x = 0$ ) hold. Multiplication is defined by

$$xy := \{x_l y + x y_l - x_l y_l, x_r y + x y_r - x_r y_r \mid x_l y + x y_r - x_l y_r, x_r y + x y_l - x_r y_l\}$$

This definition can again be understood intuitively: Compatibility of multiplication with the order dictates that  $(x - x_l)(y - y_l) > 0$  hold, because  $x > x_l$  and  $y > y_l$  (remember that addition is compatible with the order). This is equivalent (in an ordered field) to  $xy > x_l y + x y_l - x_l y_l$ , which explains why having  $x_l y + x y_l - x_l y_l$  as a left of  $xy$  is sensible. Similarly for the three other forms of the options of  $xy$ . The encoding in pCIC uses the same technique as for addition.

## 4 Similarity

Due to the high level of (anti)symmetry in the definitions, the proofs typically follow a pattern of decomposing into  $2^n$  similar cases, one case being the other with  $x$  and  $y$  swapped or left swapped with right and  $\leq$  with  $\geq$ , or both. The current development repeats the proofs (as tactic scripts) for all cases, the tactic

script for the subsequent cases being generated by a few textual search/replace operations on the tactic script of the first case.

While this is reasonably efficient, if tedious, for the author writing the proof, it is suboptimal for someone trying to read it: all trace of the link between the cases is gone. This becomes particularly critical if one aims at displaying a formal proof in a way that tries to approximate the style it would have been written (rigorously) by a human, or the style a human would like to read it in: in these styles, repeating the proof for every similar case is a significant flaw; it is expected that one writes something along the lines of “this case is similar to case 1 above, by switching  $x$  and  $y$ ” instead. It also becomes more annoying to the author when he needs to modify an already existing proof, for example to adapt it to a change in the definitions: He has to redo the textual search/replace operations over and over again.

One would thus like to prove *one* case completely and for the other cases give only the transformation that transforms a previously proven case into the one at hand. This corresponds closely to how the author thinks about the proofs and it provides the reader exactly the information he needs to be convinced when he doubts the similarity between the cases.

This could be achieved by the combination of a command to name subproofs, a language to express term transformations and a command to apply these term transformations to a previously named proof and provide the result as proof of the current goal.

Currently, the only way to name a proof in Coq is to make a separate lemma out of it. This has the significant drawback that the definition must thus be made in the section environment, no more. In the cases I encountered the distinction into similar cases was often quite deep into the proof, leading these subproofs to live in a quite big environment (goal containing a lot of hypotheses) and thus the lemma would have a long statement full of premises, that would each have to be instantiated by the *same* thing at every use. It would be much preferable to name a subproof within its environment.

As for the term transformation language, using Ltac’s term transformation language would be a significant first, and possibly sufficient, step. However, one would ideally like to have the possibility to declare transformation rules at a higher level. For example, let’s imagine one has defined, in an environment containing  $x : \text{Game}, y : \text{Game}$  the transformation “ $x \mapsto y, y \mapsto x$ ” (switching  $x$  and  $y$  in a term) under the name `SwitchXY`. If one does induction or destruction on  $x$  and  $y$  (the tactics `induction`, `destruct`, `case`, `elim`, etc), this particular transformation will become meaningless. Assuming the old  $x$  and  $y$  are deconstructed into  $(\text{Gcons } x\text{LI } x\text{RI } x\text{Lf } x\text{Rf})$  and  $(\text{Gcons } y\text{LI } y\text{RI } y\text{Lf } y\text{Rf})$ , respectively, the transformation is naturally replaced by “ $x\text{LI} \mapsto y\text{LI}, x\text{RI} \mapsto y\text{RI}, x\text{Lf} \mapsto y\text{Lf}, x\text{Rf} \mapsto y\text{Rf}$  and vice-versa”. One would like this to happen automatically.

There are other techniques that, eventually augmented by some syntactic sugar, can adequately handle some of the cases this scheme would handle. Their combined power, however, does not cover all these cases:

- Parametric tactic scripts that parametrise everything that is touched by the transformation.

This technique leads, for example when the proof consists in a long series of applications of different lemmas that exist in two flavours (for example one talking about lefts of games and one speaking about rights of games), to unwieldy large parameter lists to the tactic script. The author is still applying the transformation by hand, only to a somewhat smaller text body. And the information of what this transformation *is* is not available to the reader.

- Ad-hoc lemmas. Generalising the first case to a lemma that covers all cases is a fairly natural idea. But it only helps if this generalised lemma can be proven in essentially the same effort as the one case, or at least less effort than doing all cases. In particular, if the only way to prove the lemma is to enter exactly the same case analysis one seeks to escape, no gain is achieved. This is exactly the situation with Conway games when the cases are linked by the left-right antisymmetry.
- Abstraction of common patterns. The most elegant and natural way, mathematically. This is only useful, though, if the work of introducing (for the author) or read and understand (for the reader) the abstraction is less than the work it will save you by using this abstraction later. Specifically, in the case at hand here, it seems to require quite numerous proofs that the definitions indeed present the (anti)symmetries you want to exploit in your proofs, which cancel out the advantage of reuse that one would get.

## 5 Surreal Numbers in the $\text{Type}_i$ Hierarchy

### 5.1 The Hierarchy

The type theory that Coq implements, the Predicative Calculus of Inductive Constructions, has the following sorts<sup>8</sup> structure:

- Prop is the sort of propositions
- Set is the sort of “small” data types. “Small” here means inductive definitions whose constructors do not embed sets or propositions.
- A stratified sort of sorts  $\text{Type}$ : for any natural number  $n$ , a *universe* sort  $\text{Type}_n$ . These are organised into a hierarchy by increasing index.

The following relations hold:  $\text{Prop} : \text{Type}_0$ ,  $\text{Set} : \text{Type}_0$  and  $\text{Type}_i : \text{Type}_{i+1}$ . Moreover,  $T : \text{Type}_i$  implies  $T : \text{Type}_{i+1}$ . In particular,  $\text{Type}_i : \text{Type}_j$  if and only if  $i < j$ . In an inductive definition, if a constructor of the type being defined takes an argument of type  $\text{Type}_i$ , then the inductive type being defined must be of type at least  $\text{Type}_{i+1}$ ; but if a constructor takes an argument of type  $A$  and  $A : \text{Type}_i$ , the inductive type being defined must be of type at least  $\text{Type}_i$ . See [18] for more details.

---

<sup>8</sup> Sorts are the types of types; sorts are sometimes called *universes*; every type has a sort as its type.

Coq, however, never deals with the Type indices explicitly; it deals only with constraints (strict and non-strict inequalities and equalities) between the indices of the sort of various types. These constraints are hidden from the user, generated automatically and checked for consistency automatically.

As the constructors of the Game type do, by necessity, embed sets, it cannot be of sort Set and is pushed in the  $\text{Type}_i$  hierarchy.

## 5.2 Universe Polymorphism

The pCIC, as nearly all the type theories behind active proof assistants, doesn't feature *universe polymorphism* ([12]). This means that every type sits at one precise level of the  $\text{Type}_i$  hierarchy and there is no (implicit or explicit) abstraction over universe levels. I'll explain by example, with this polymorphic "type product" (pairing, tuple of length 2) type, from the Coq standard library:

**Inductive** prodT ( $A, B : \text{Type}_i$ ) :  $\text{Type}_i := \text{pairT} : A \rightarrow B \rightarrow \text{prodT } A B$

No universe polymorphism means that there is a *global scope*  $i$  such that ( $\text{prodT } A B$ ) is of sort  $\text{Type}_i$ . This leads to the problem that the pair  $(\mathbb{N} \times \mathbb{N}, \mathbb{N})$ , naively a perfectly innocuous and reasonable pair to construct, cannot be constructed because it is ill-typed. That pair is encoded by the term  $(\text{pairT } (\text{prodT } \text{nat } \text{nat}) \text{ nat})$ . I'll now show that it is ill-typed. Let  $t$  be  $(\text{prodT } \text{nat } \text{nat})$ .  $t$  is of type  $\text{Type}_i$ , the  $i$  being the same one as the one in the inductive definition above. We are trying to give  $t$  as the first argument to  $\text{pairT}$ . For this to be well-typed, we must have  $t : A$  and  $A : \text{Type}_i$ . We are here instantiating  $A$  with  $\text{Type}_i$  and the condition thus reduces to  $\text{Type}_i : \text{Type}_i$ , which is equivalent to  $i < i$ , which does not hold. In short, pairs of products cannot be constructed, they are not well-typed. The same holds for the type theory of most active type theory based proof assistants.

This problem can be avoided in practical cases by defining as many copies of  $\text{prodT}$  (with different names) as necessary. For example, one would define

**Inductive** prodT ( $A, B : \text{Type}_i$ ) :  $\text{Type}_i := \text{pairT} : A \rightarrow B \rightarrow \text{prodT } A B$   
**Inductive** prodT0 ( $A, B : \text{Type}_j$ ) :  $\text{Type}_j := \text{pairT0} : A \rightarrow B \rightarrow \text{prodT0 } A B$

so that one can form the pair  $(\text{pairT0 } (\text{prodT } \text{nat } \text{nat}) \text{ nat})$ , whose well-typedness only requires  $i < j$ .

For a type theory with a hierarchy of sorts, such as the pCIC, having universe polymorphism would mean that the  $\text{prodT}$  definition would – conceptually – be like

**Inductive** prodT $_i$  ( $A, B : \text{Type}_i$ ) :  $\text{Type}_i := \text{pairT}_i : A \rightarrow B \rightarrow \text{prodT}_i A B$

that is a *family of definitions*, one for each level in the hierarchy. In this approach,  $(\mathbb{N} \times \mathbb{N}, \mathbb{N})$  would be  $(\text{pairT}_1 (\text{prodT}_0 \text{ nat } \text{nat}) \text{ nat})$ .

In my surreal numbers development, I encounter exactly this problem, in a somewhat more involved manner:

**Inductive Game** :  $\text{Type}_g :=$   
 $\text{Gcons} : \forall L, R : \text{Type}_i, \forall L_f : (L \rightarrow \text{Game}), \forall R_f : (R \rightarrow \text{Game}), \text{Game}$   
**Inductive prodT** ( $A, B : \text{Type}_j$ ) :  $\text{Type}_p := \text{pairT} : A \rightarrow B \rightarrow \text{prodT } A B$

Due to the typing rules, we have  $g > i$  and  $p \geq j$ . But in the definition of multiplication, I use a `prodT` of indexes of games (thus  $j \geq i$ ) as an index for a new game (thus  $i \geq p$ ). To this point, we thus have  $g > i = p = j$ .

The use of `prodT` (or something isomorphic to it) for the definition of multiplication is hardly avoidable: Let  $x = \text{Gcons } x\text{LI } x\text{RI } x\text{Lf } x\text{Rf}$  and  $y = \text{Gcons } y\text{LI } y\text{RI } y\text{Lf } y\text{Rf}$ , two games. The definition section 3.3 tells us the left of  $xy$  is  $\{x_1y + xy - x_1y_i \mid x_1 \in L_x, y_i \in L_y\} \cup \{x_r y + xy_r - x_r y_r \mid x_r \in L_x, y_r \in R_y\}$ . It is made out of two components, it is thus natural to index it by a sum (disjoint union) of two types `AI` and `BI`, each indexing one component. If `AI` indexes the left component (the one to the left of the  $\cup$  sign), what should `AI` look like? The need is for a type `AI` such that:

- From each `i:AI`, one can extract an `xli:xLI` and a `yli:yLI`.
- When `i` ranges over `AI`, the extraction above ranges over all combinations of `xli:xLI` and `yli:yLI`.

That’s the description of the product of `xLI` and `yLI`. Similarly, the natural choice for `BI` is the product of `xLI` and `yRI`.

Now, let’s suppose one wants to consider the structure of `Games` equipped with the pre-order relation  $\leq$ . This structure is the pair  $(\text{Game}, \leq)$ , i.e.  $(\text{pairT } \text{Game } \leq)$ . The well-typedness of this pair requires  $j \geq g$ , contradicting the  $g > j$  above and making the whole ill-typed. Had I not defined multiplication, that pair would have been well-typed!

In summary, if a product (of anything) is ever used to construct values of type `A`, then `A` cannot be itself in a pair, and vice-versa.

The “define new copies of `prodT` with different names” technique worked well enough for my needs. However, the lack of universe polymorphism has the potential to grow to a significant problem for more complex developments. For example, if one defines a list type and an extensive library of operations on lists and lemmas on properties of these operations. Suddenly, because one wants to use lists on higher-level types, one has to redefine a second copy of the notion of lists ... and all its operations and properties. And maybe a third copy, and a fourth, ... These can be significant pieces of code, and the number of times they have to be repeated is potentially unbounded. However, it remains to be seen whether this number will grow substantially in practice. It is also possible that this could be tackled through functors, rather than universe polymorphism.

Note that the specific typing rules used here are those of the `pCIC`, but other type theories tend to have the same kind of stratification and level ordering limitations (to define something in universe of level  $n$ , one can use only things of lower level).

### 5.3 The Game Type and the Collection of All Games

Does the `Game` type cover *all* Conway Games / surreal numbers? This essentially boils down to: Can one for any set  $X$  provide as first and second argument of

the constructor of Game a type  $T$  “bigger” than  $X$ , meaning such that there is a surjection from  $T$  to  $X$ . In classical terms, this means that the Game type can embed sets of any cardinality.

Indeed, in  $\text{Gcons } L_I R_I L_f R_f$ , one can replace the left (or right) index type by any other type  $X$ , as long as there is a surjection  $f$  from  $X$  to  $L_I$ :  $\text{Gcons } X R_I (L_f \circ f) R_f$  encodes the same game (in terms of identity).

[21] says that the whole theory of ZFC with  $n$  inaccessible cardinals can be embedded in  $\text{Type}_{n+2}$  (plus some axioms). Thus the pCIC, as a whole, can encode a countable infinity of inaccessible cardinals. The Game type, however, will be able to use only a finite amount of them and it thus does not cover all surreal numbers that the pCIC can encode.

If the pCIC had universe polymorphism, it would give us a family of types (one per Type level) which, taken together, would cover all games.

## 6 Conclusion

### 6.1 Future Outlooks

**Division and Field Structure.** Inversion (and thus indirectly division) is defined in [9] as follows: Let  $x := \{\{0\} \cup L_x \mid R_x\}$  be a positive surreal number such that  $\forall x_l \in L_x, x_l > 0$ . Any positive surreal number is either of this form, or there is  $x'$  of this form such that  $x' = x$ . Let's note that then  $\forall x_r \in R_x, x_r > 0$  is automatic. The inverse of  $x$ , which we call  $y$ , is:

$$\left\{ 0, \frac{1 + (x_r - x)y_l}{x_r}, \frac{1 + (x_l - x)y_r}{x_l} \mid \frac{1 + (x_l - x)y_l}{x_l}, \frac{1 + (x_r - x)y_r}{x_r} \right\}$$

where  $x_l$  is an element of  $L_x$ , and occurrences of  $y_r$  or  $y_l$  are to be understood as: the first left of  $y$  is 0. Every left (respectively right) of  $y$  itself generates new lefts and rights of  $y$ , which in turn generate new lefts and rights.

Implementing this on top of the encoding of Surreal Numbers presented here is delicate, because index types (the first two arguments of  $\text{Gcons}$ ) that would have the right “structure” to make the index functions (last two arguments of  $\text{Gcons}$ ) natural are complex, but it can be done.

**Sign Sequences.** Surreal numbers can be defined as sign sequences (sequences of ordinal length of elements of  $\{+, -\}$ ), rather than with “bisets”, like [9] does. Sign sequences are introduced in chapter 3, on page 30 of [9]. At first glance, this looks strikingly similar to de Bruijn’s way of constructing  $\mathbb{R}$  without the rationals ([10]), formalised in AUTOMATH by J.T. Udding ([20]). It would be an interesting comparison to see if this construction, which one would think more “computer-friendly” at first sight (because of the use of sequences instead of sets), would be simpler to handle than the biset construction.

I have chosen to keep to the biset construction for two reasons:

1. I wanted to “stress-test” Coq precisely on a notion that is opposite to the foundations of Coq; see section 6.2.

2. This approach would need a development of the theory of ordinals – or at least sequences of arbitrary ordinal length – in Coq, which hasn't been done yet.

## 6.2 Conclusions

One of my reasons for starting this development is that I wanted to do a “torture test” of Coq, particularly in the area of dealing with formalisations that are in a certain sense “opposite” to its viewpoint on mathematics:

- a very set-theoretic definition, while Coq is based on Type Theory, Set Theory and Type Theory being competing foundations for mathematics.
- definitions and proofs made classically, that is without particular attention to constructibility, while the natural logic of Coq is intuitionistic.

I wanted to see how much trouble this would create. I played the role of a “classical” mathematician (like Conway) that just wants to check formally whether his nifty, well-developed and mature ideas are correct, with the help of the computer. I wanted to see how much of my proofs and definitions Coq would force me to adapt to it instead of it adapting to me. How much did I have to learn about Type Theory to get the job done?

The answer is, not immensely so, but still significantly. The lack of induction-recursion is not very damaging to passing the test, but only because Conway didn't insist on making use of induction-recursion in the first place. An inductive-recursive presentation *is* the most natural one. The forced separation of  $\leq$  into a mutual induction between  $\leq$  and  $\triangleright$  is more problematic: to a mathematician not intimate with type theory and computer mathematics, this requirement looks totally artificial. And the problem is not only about coming up with this new definition in the first place: in proofs of properties of  $\leq$  (at least a “starting base”, up to equivalence with the Conway definition), one keeps getting  $\triangleright$  and having to prove a similar property on it.

The way to define the Game type looks natural once one knows of Aczel's work, but this, again, requires intimate knowledge of type theory. Trying to get inspiration from the standard library is counter-productive: the notion of “set” in the standard library is exactly the approach of sets in type theory that doesn't work for this use (it is sets as subsets of a preexisting universe, represented by their “element of” predicate).

But it still feels relevant to spend the effort of a full formalisation: I *did* find out that the proof of transitivity of  $\leq$  in [9] is, at best, presented misleadingly simply, abusing the word “similarly”: various random active research mathematicians could *not* easily fill in the missing details (correctly answer the question “what is the induction scheme used here, and spell out the induction hypothesis”).

On the good side of things, the version of Coq following version 8.0 does bring a very positive change: it deals with defined equalities (weaker than Coq's native equality) much better, thanks to the work of Sacerdoti-Coen, presented in [17]. Switching to (a pre-release development snapshot of) that version significantly reduced the size of my proofs.

Note however that the Game type cannot cover all surreal numbers.

**Acknowledgements.** I'd like to thank the referees and editors for their constructive comments.

## References

1. Peter Aczel. The type theoretic interpretation of constructive set theory. In Macintyre, Pacholsky, and Paris, editors, *Logic Colloquium 77*. Springer, 1977.
2. Peter Aczel. The type theoretic interpretation of constructive set theory: Choice principles. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*. North-Holland, 1982.
3. Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definitions. In *Proceedings of Methodology and Philosophy of Sciences*, 1985.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
5. Ana Bove. *General Recursion in Type Theory*. Thesis for the degree of doctor of philosophy, Department of Computing Science - Chalmers University of Technology and Göteborg University, SE-412 96 Göteborg, Sweden, November 2002.
6. Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14<sup>th</sup> International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.
7. Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. To be published in *Mathematical Structures in Computer Science*, 2005.
8. Venanzio Capretta. Recursive families of inductive types. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2000.
9. John H. Conway. *On Numbers and Games*. A K Peters, Ltd, second edition, 2001. First Edition: 1976.
10. N.G. de Bruijn. Introducing the reals as a totally ordered additive group without using the rationals. Memorandum 1975-13, Eindhoven University of Technology, PO Box 513, Eindhoven, The Netherlands, November 1975.
11. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.
12. Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.
13. Donald E. Knuth. *Surreal numbers: how two ex-students turned on to pure mathematics and found total happiness*. Addison-Wesley, 1974.
14. Jacob Lurie. The effective contents of surreal algebra. *Journal of Symbolic Logic*, 63, June 1998.
15. Frank Rosemeier. A constructive approach to conway's theory of games and numbers. In *Seminarberichte aus dem Fachbereich Mathematik der FernUniversität Hagen*, volume 70, pages 1–18, 2001.
16. Frank Rosemeier. On conway-numbers and generalized real numbers. In Ulrich Berger, Horst Oswald, and Peter Schuster, editors, *Reuniting the Antipodes*. Kluwer Academic Publishers, 2001.



17. Claudio Sacerdoti-Coen. A coq tactic for one step conditional rewriting. Small TYPES workshop - Types for Mathematics / Libraries of Formal Mathematics, November 2004. Slides at <http://www.cs.ru.nl/fnds/typesworkshop/#sacerdoti>.
18. The Coq Development Team - LogiCal Project. *The Coq Proof Assistant - Reference Manual*. INRIA, Domaine de Voluceau, Rocquencourt - B.P. 105, F-78153 Le Chesnay Cedex - France, 2005.
19. Claus Tøndering. Surreal numbers - an introduction. HTTP, December 2001. Version 1.2.
20. J.T. Udding. A theory of real numbers and its presentation in Automath. Master's thesis, Eindhoven University of Technology, P.O. Box 13, NL-5600MB Eindhoven, The Netherlands, February 1980. under supervision of dr. N.G. de Bruijn.
21. Benjamin Werner. Sets in types, types in sets. In T. Ito and M. Abadi, editors, *TACS'97*, LNCS, page 1281. Springer-Verlag, 1997. (corresponding Coq code is in user contribution "zermelo-fraenkel").

# A Few Constructions on Constructors

Conor McBride<sup>1</sup>, Healdene Goguen<sup>2</sup>, and James McKinna<sup>3</sup>

<sup>1</sup> School of Computer Science and Information Technology, University of Nottingham

<sup>2</sup> AT&T Labs, Florham Park, New Jersey

<sup>3</sup> School of Computer Science, University of St Andrews

**Abstract.** We present four constructions for standard equipment which can be generated for every inductive datatype: case analysis, structural recursion, no confusion, acyclicity. Our constructions follow a two-level approach—they require less work than the standard techniques which inspired them [11, 8]. Moreover, given a suitably heterogeneous notion of equality, they extend without difficulty to inductive families of datatypes. These constructions are vital components of the translation from dependently typed programs in pattern matching style [7] to the equivalent programs expressed in terms of induction principles [21] and as such play a crucial behind-the-scenes rôle in Epigram [25].

## 1 Introduction

In this paper, we show how to equip inductive families of datatypes [10] with useful standard constructions. When you declare an inductive datatype, you expect the following to be available ‘for free’:

- the ability to write terminating computations by case analysis and structural recursion;
- proofs that its constructors are injective and disjoint; and
- proofs that cyclic equations are refutable.

We show these expectations to be well-founded by exhibiting constructions which fulfil them. These constructions may readily be mechanised, so that we may rely on these intuitive properties of datatypes implicitly as we go about our work.

In prior publications, we have indeed relied implicitly on these properties. Both the ‘elimination with a motive’ tactic [22] and the Epigram programming language [25] make heavy use of this equipment, but the constructions themselves have only appeared in [21]. We hope that this paper will serve as a more accessible technical reference.

Our approach to all of these constructions is more directly computational than others in the literature. In effect, our presentation of the structural recursion operator reworks Giménez’s *inductive* justification of Coq’s `fix` primitive, but for us the notion of ‘guarded by constructors’ is expressed by a *recursive* computation. Meanwhile, the ‘constructors injective and disjoint’ properties are captured by a single ‘no confusion’ theorem which computes its conclusion from

its input. This sidesteps the problems of extending the ‘equality respects predecessors’ method of proving injectivity (automated for simple types by Cornes and Terrasse [8], and still standard [4]) to inductive families. It is, besides, rather neater. The ‘refutation of cyclic equations’ is again presented as a single theorem for each datatype family. To our knowledge, this construction is entirely original.

## 2 Inductive Datatypes

Before we begin, let us be clear about the data with which we deal and also introduce our notational conventions. We mean the strictly positive inductive families of datatypes from Luo’s UTT [18]. Modulo technical details, these correspond to the inductive families found in the Alf system [10], the Coq system [26] and Epigram [25]. The story we tell here is by no means specific to any one presentation. Here we follow Epigram’s presentational style, with declarations of datatype families and their constructors, and the type signatures of function definitions, given by ‘inference rules’ resembling those of natural deduction. As well as avoiding a cumbersome and scarcely legible linearised notation for dependent types, we exploit the ability to systematically omit declarations which may be inferred during typechecking.

*The Tree example.* We require that a datatype be presented by a formation rule and constructors—Tree provides the paradigm:

$$\text{data } \overline{\text{Tree} : \star} \quad \text{where } \overline{\text{leaf} : \text{Tree}} \quad \overline{l, r : \text{Tree} \quad \text{node } l r : \text{Tree}}$$

and equipped with an induction principle like this (noting here, for example, that we need no explicit declaration of the arguments  $l, r : \text{Tree}$  in the hypothesis typing  $\text{node}'$ )

$$\frac{t : \text{Tree} \quad P : \text{Tree} \rightarrow \star \quad \text{leaf}' : P \text{ leaf} \quad \overline{l' : P l \quad r' : P r \quad \text{node}' l r l' r' : P (\text{node } l r)}}{\text{treeInd } t P \text{ leaf}' \text{ node}' : P t}$$

whose computational behaviour is given by reduction schemes like these

$$\begin{aligned} \text{treeInd leaf } P \text{ leaf}' \text{ node}' &\rightsquigarrow \text{leaf}' \\ \text{treeInd (node } l r) P \text{ leaf}' \text{ node}' &\rightsquigarrow \text{node}' l r \\ &\quad (\text{treeInd } l P \text{ leaf}' \text{ node}') \\ &\quad (\text{treeInd } r P \text{ leaf}' \text{ node}') \end{aligned}$$

Another standard practice in Epigram is to make elimination principles (inductive or otherwise) take their arguments in the order they are typically conceived. The user supplies the *target*  $t$  of the elimination; the machine infers the *motive*  $P$  for the elimination from the goal at hand [22]; the user supplies the *methods* by which the motive is pursued in each case.

*The general form.* For each of our constructions, we shall treat `Tree` as a running example, then give the general case. We consider inductive families [10]—mutually defined collections of inductive types, *indexed* by a given telescope,  $\Theta$

$$\underline{\text{data}} \frac{\Theta}{\text{Fam } \Theta : \star} \underline{\text{where}} \cdots \frac{\Delta \quad \cdots \quad \frac{\Psi_i}{v_i \Psi_i : \text{Fam } \vec{r}_i} \quad \cdots}{\text{con } \Delta \ v_1 \ \dots \ v_n : \text{Fam } \vec{s}} \quad \cdots$$

*Telescopes.* We use Greek capitals to denote *telescopes* [9]—sequences of declarations  $x_1 : X_1; \dots \ x_n : X_n$  where later types may depend on earlier variables. We shall freely write iterated binders over telescopes  $\lambda \Theta \Rightarrow T$ , and we shall also use telescopes in argument positions, as in `Fam`  $\Theta$  above, to denote the sequence of variables declared, here `Fam`  $x_1 \ \dots \ x_n$ . For brevity, we shall sometimes also write subscripted sequences  $x_1 \ \dots \ x_n$  in vector notation  $\vec{x}$ .

For any `Fam`  $: \forall \Theta \Rightarrow \star$ , we write  $\langle \text{Fam} \rangle$  to denote the telescope  $\Theta; x : \text{Fam } \Theta$ , thus capturing in a uniform notation the idea of ‘an arbitrary element from an arbitrary family instance’. We may declare  $\vec{y} : \Theta$ , the renamed telescope  $y_1 : X_1; \dots \ y_n : [\vec{y}/\vec{x}]X_n$ . We may assert  $\vec{t} : \Theta$ , meaning the substituted sequence of typings  $t_1 : X_1; \dots \ t_n : [\vec{t}/\vec{x}]X_n$  which together assert that the sequence  $\vec{t}$  is *visible through*<sup>1</sup>  $\Theta$ .

*Constructors.* An inductive family may have several constructors—`con`, above, is typical. It has a telescope of *non-recursive* arguments  $\Delta$  which, for simplicity, we take to come before the *recursive* arguments  $v_i$ . Only as the family for the return types of constructors and their recursive arguments may the symbol `Fam` occur—that is, our families are *strictly positive*. None the less, recursive arguments may be higher-order, parametrised by a telescope  $\Psi_i$ . Note that the indices  $\vec{s}$  of the return type and  $\vec{r}_i$  of each recursive argument are arbitrary sequences visible through  $\Theta$ .

In addition to the above criteria, some restriction must be made on the relative position of `Fam` and its contents in the universe hierarchy of types if paradox is to be avoided—we refer the interested reader to [18, 6] and leave universe levels implicit [14].

For any  $\Delta, \vec{v}, i$  and  $\vec{t} : \Psi_i$ , we say that  $v_i \vec{t}$  is *one step smaller* than `con`  $\Delta \vec{v}$ . The usual notion of being *guarded by constructors* is just the transitive closure of this one step relation.

*Induction and Computation.* The induction principle for a family is as follows:

$$\frac{\vec{x} : \langle \text{Fam} \rangle \quad P : \forall \langle \text{Fam} \rangle \Rightarrow \star \quad \cdots \quad \frac{\Psi_i}{v_i' \Psi_i : P \ \vec{r}_i (v_i \Psi_i)} \quad \cdots}{\text{con}' \ \Delta \ \vec{v} \ \vec{v}' : P \ \vec{s} (\text{con } \Delta \ \vec{v})} \quad \cdots}{\text{famInd } \vec{x} \ P \ \dots \ \text{con}' \ \dots : P \ \vec{x}}$$

It takes a target sequence in  $\langle \text{Fam} \rangle$ , a motive abstracting over  $\langle \text{Fam} \rangle$ , and a method for each constructor. Each method abstracts its constructor’s arguments

<sup>1</sup> Less is visible through a longer telescope, but we see it in more detail.

and the inductive hypotheses associated with recursive arguments. Here are the associated reduction schemes:

$$\begin{array}{c}
\dots \\
\mathbf{famInd} \_ \dots \_ (\mathbf{con} \Delta \vec{v}) P \dots \mathbf{con}' \dots \rightsquigarrow \\
\mathbf{con}' \Delta \vec{v} \quad \dots \quad (\lambda \Psi_i \Rightarrow \mathbf{famInd} \vec{r}_i (v_i \Psi_i) P \dots \mathbf{con}' \dots) \quad \dots \\
\dots
\end{array}$$

The notion of computation captured by these induction principles—higher-order primitive recursion—yields a system which is strongly normalizing for well typed terms [13]. Moreover, they support *large elimination*—the computation of types by recursion, which we use extensively in this paper. Induction principles provide a sound and straightforward basis for programming and reasoning with inductive structures, and whilst not especially attractive, they provide a convenient ‘machine code’ in terms of which higher-level programming and proof notations can be elaborated. There is plenty of scope for exploiting the properties of indices to optimize the execution of induction principles and, by the same token, the representation of inductive data [5].

*Proofs by Elimination.* We present some of our constructions just by writing out the proof terms schematically. For the more complex constructions, this is impractical. Hence we sometimes give the higher-level proof strategy rather than the term to which it gives rise. This is also our preferred way to *implement* the constructions, via the proof tools of the host system. We shall not require any sophisticated machinery for inductive proof [22]—indeed, we are constructing components for that machinery, so we had better rely on something simpler. We require only ‘undergraduate’ elimination, where the motive just  $\lambda$ -abstracts variables already  $\forall$ -abstracted in the goal. Let us put induction to work.

### 3 Case Analysis and Structural Recursion

Coq’s `case` and `fix` primitives conveniently separate induction’s twin aspects of case analysis and recursion, following a suggestion from Thierry Coquand. By way of justification, in [11] Eduardo Giménez shows how to reconstruct an individual `fix`-based recursion in terms of induction by a memoization technique involving an inductively defined ‘course-of-values’ data structure. Here, we show how to build the analogous tools ‘in software’, presenting essentially the same technique in general, and defining course-of-values *computationally*.

A case analysis principle is just an induction principle with its inductive hypotheses chopped off. Correspondingly, its proof is by an induction which makes no use of the inductive hypotheses. For `Tree` we have

$$\frac{t : \mathbf{Tree} \quad P : \mathbf{Tree} \rightarrow \star \quad \mathit{leaf}' : P \ \mathit{leaf} \quad \frac{l, r : \mathbf{Tree}}{\mathit{node}' \ l \ r : P \ (\mathit{node} \ l \ r)}}{\mathbf{treeCase} \ t \ P \ \mathit{leaf}' \ \mathit{node}' : P \ t}$$

$$\mathbf{treeCase} \ t \ P \ \mathit{leaf}' \ \mathit{node}' \Rightarrow \mathbf{treeInd} \ t \ P \ \mathit{leaf}' \ (\lambda l; r; l'; r' \Rightarrow \mathit{node}' \ l \ r)$$

and in general,

$$\frac{\vec{x} : \langle \mathbf{Fam} \rangle \quad P : \forall \langle \mathbf{Fam} \rangle \Rightarrow \star \quad \cdots \quad \frac{\Psi_i}{v_i \Psi_i : \mathbf{Fam} \vec{r}_i} \quad \cdots}{\mathbf{famCase} \vec{x} P \dots \mathit{con}' \dots : P \vec{x}} \quad \cdots$$

$$\mathbf{famCase} \vec{x} P \dots \mathit{con}' \dots \Rightarrow$$

$$\mathbf{famInd} \vec{x} P \dots (\lambda \Delta; \vec{v}; \vec{v}' \Rightarrow \mathit{con}' \Delta \vec{v}) \dots$$

The obvious ‘pattern matching equations’ for **famCase** hold computationally.

$$\cdots$$

$$\mathbf{famCase} \_ \dots \_ (\mathit{con} \Delta \vec{v}) P \dots \mathit{con}' \dots \Rightarrow \mathit{con}' \Delta \vec{v}$$

$$\cdots$$

The recursion principle captures the notion that to compute a  $P t$ , you may assume that you have a  $P$  for every recursive subobject *guarded by constructors* in  $t$ . The computational properties of the system give us a convenient way to capture this notion: inductions proceed when fed with constructors and get stuck otherwise. With the assistance of product and unit types, we may exploit large elimination to define a predicate transformer, capturing the idea that a given predicate  $P$  holds ‘below  $t$ ’. Informally, we write a primitive recursive program in pattern matching style, but the translation to induction is direct:

$$\frac{P : \mathbf{Tree} \rightarrow \star \quad t : \mathbf{Tree}}{P \mathbf{belowTree} t : \star}$$

$$P \mathbf{belowTree} \mathit{leaf} \Rightarrow \mathbf{One}$$

$$P \mathbf{belowTree} (\mathit{node} l r) \Rightarrow (P \mathbf{belowTree} l \wedge P l) \wedge (P \mathbf{belowTree} r \wedge P r)$$

We may now state the recursion principle:

$$\frac{t : \mathbf{Tree} \quad P : \mathbf{Tree} \rightarrow \star \quad \frac{t : \mathbf{Tree} \quad m : P \mathbf{belowTree} t}{p t m : P t}}{\mathbf{treeRec} t P p : P t}$$

When we apply this principle, we do not enforce any particular case analysis strategy. Rather, we install a hypothesis, here the additional argument  $p$ , which will unfold computationally whenever and wherever case analyses may reveal constructors. At any point in an interactive development, this unfolding hypothesis amounts to a menu of templates for legitimate recursive calls. This is how recursion is elaborated in Epigram—examples of its use can be found in [25], including the inevitable Fibonacci function

$$\cdots$$

$$\mathbf{fib} (\mathit{suc} (\mathit{suc} n)) \Rightarrow \mathbf{fib} n + \mathbf{fib} (\mathit{suc} n)$$

Let us now construct **treeRec**:

**treeRec**  $t P p \Rightarrow p t$  (**below**  $t$ ) where

$$\frac{t : \mathbf{Tree} \quad m : P \mathbf{belowTree} t}{\mathbf{step} t m : P \mathbf{belowTree} t \wedge P t}$$

$$\mathbf{step} t m \Rightarrow (m; p t m)$$

$$\frac{t : \mathbf{Tree}}{\mathbf{below} t : P \mathbf{belowTree} t}$$

$$\mathbf{below} \text{ leaf} \Rightarrow ()$$

$$\mathbf{below} (\text{node } l r) \Rightarrow (\mathbf{step} l (\mathbf{below} l); \mathbf{step} r (\mathbf{below} r))$$

In the general case, we construct first **belowFam** by induction, then **famRec**, using the auxiliary functions **step** and **below**:

$$\frac{P : \forall \langle \mathbf{Fam} \rangle \Rightarrow \star \quad \vec{x} : \langle \mathbf{Fam} \rangle}{P \mathbf{belowFam} \vec{x} : \star}$$

$$P \mathbf{belowFam} \vec{x} \Rightarrow$$

$$\mathbf{famInd} \vec{x} (\lambda \langle \mathbf{Fam} \rangle \Rightarrow \star)$$

...

$$(\lambda \Delta; \vec{v}; \vec{B} \Rightarrow \dots \wedge (\forall \Psi_i \Rightarrow (B_i \Psi_i \wedge P \vec{r}_i (v_i \Psi_i))) \wedge \dots)$$

...

$$\frac{\vec{x} : \langle \mathbf{Fam} \rangle \quad P : \forall \langle \mathbf{Fam} \rangle \Rightarrow \star \quad \frac{\vec{y} : \langle \mathbf{Fam} \rangle \quad m : P \mathbf{belowFam} \vec{y}}{p \vec{y} m : P \vec{y}}}{\mathbf{famRec} \vec{x} P p : P \vec{x}}$$

**famRec**  $\vec{x} P p \Rightarrow p \vec{x}$  (**below**  $\vec{x}$ ) where

$$\frac{\vec{y} : \langle \mathbf{Fam} \rangle \quad m : P \mathbf{belowFam} \vec{y}}{\mathbf{step} \vec{y} m : P \mathbf{belowFam} \vec{y} \wedge P \vec{y}}$$

$$\mathbf{step} \vec{y} m \Rightarrow (m; p \vec{y} m)$$

$$\frac{\vec{y} : \langle \mathbf{Fam} \rangle}{\mathbf{below} \vec{y} : P \mathbf{belowFam} \vec{y}}$$

$$\mathbf{below} \vec{y} \Rightarrow$$

$$\mathbf{famInd} \vec{y} (P \mathbf{belowFam})$$

...

$$(\lambda \Delta; \vec{v}; \vec{b} \Rightarrow (\dots; \lambda \Psi_i \Rightarrow (\mathbf{step} \vec{r}_i (v_i \Psi_i) (b_i \Psi_i)); \dots))$$

...

## 4 Heterogeneous Equality

We shall shortly prove that the constructors of inductive families are injective and disjoint, but before we prove it, we must first figure out how to *state* it.

Equality is traditionally defined inductively as follows:<sup>2</sup>

$$\text{data } \frac{x, y : T}{\text{Eq}_T x y : \star} \quad \text{where } \frac{x : T}{\text{reflEq}_T x : \text{Eq}_T x x}$$

As we have often observed, this notion of equality is unsuitable once dependently typed functions and data become the object of study, rather than merely the means to study simply typed phenomena. Even such a triviality as ‘a function takes equal inputs to equal outputs’ ceases to be a proposition (never mind being provable) as soon as the output type depends on the input type. Given a function  $f : \forall x : S \Rightarrow T[x]$ , then we may not equate  $f x : T[x]$  with  $f y : T[y]$ . The first author’s approach to this problem [21, 22] has achieved some currency—liberalize the *formation* rule to admit heterogeneous aspirations to equality whilst retaining a more conservative *elimination* rule delivering equal treatment only for homogeneous equations.

$$\frac{\frac{x : X \quad y : Y}{x = y : \star} \quad \frac{x : X}{\text{refl} : x = x} \quad \frac{x, x' : X \quad q : x = x' \quad \frac{x' : X \quad q : x = x'}{P x' q : \star} \quad p : P x \text{ refl}}{= \mathbf{Elim} \ x \ x' \ q \ P \ p : P \ x' \ q}}{= \mathbf{Elim} \ \_ \ \text{refl} \ \_ \ p \ \sim \ p}$$

Note that  $= \mathbf{Elim}$  is *not* the standard induction principle which we would expect for such a definition, with a motive ranging over all possible equations  $\langle = \rangle$ , nor is either derivable from the other. However, it is plainly the structural elimination principle for the subfamily of homogeneous equations. It is similar in character to the Altenkirch-Streicher ‘K’ axiom [28]

$$\frac{q : \text{Eq}_T x x \quad P : \text{Eq}_T x x \rightarrow \star \quad p : P (\text{reflEq}_T x)}{\mathbf{K} \ T \ x \ q \ P \ p : P \ q}$$

which is plainly the structural elimination rule for reflexive equations. **Eq**-with-**K** is known to be strictly stronger than **Eq**-without-**K** [16]. Heterogeneous equality and **Eq**-with-**K** are known to have the same strength [21]. We shall not repeat the whole construction here, but merely observe that one may express a heterogeneous equation as a homogeneous equation on type-term pairs  $(A; a) : \exists A : \star \Rightarrow A$ . The law  $\text{Eq}_{\exists A : \star \Rightarrow A} (A; a) (A; a') \rightarrow \text{Eq}_A a a'$  is equivalent to **K** [28].

The crucial point is this: we can now formulate *telescopic equations*, and use them to eliminate subfamilies of general **Fam**, as in [22, 25]. If  $\vec{x} : \Delta$  and  $\vec{y} : \Delta$ , then  $\vec{x} = \vec{y}$  is the telescope  $q_1 : x_1 = y_1; \dots; q_n : x_n = y_n$  with  $\text{refl} : \vec{t} = \vec{t}$ .

<sup>2</sup> In some systems, propositions inhabit a separate universe from datatypes: this distinction is unimportant for our purposes.



## 5 No Confusion

The standard techniques for proving that constructors are injective and disjoint go back to Smith’s analysis of the logical strength of universes in type theory [27]—‘large elimination’ being the key technical insight; one ‘projection-based’ approach to their mechanisation may be found in [8]. It is not hard to construct a predicate which is trivial for leaf and absurd for (node  $l r$ )—given a proof that the two are equal, the substitutivity of equality gives you ‘trivial implies absurd’. Meanwhile, left and right projections will respect a proof that node  $l r = \text{node } l' r'$ , yielding  $l = l'$  and  $r = r'$ . Of course, projections are not always definable—here they are *locally* definable because  $l$  and  $r$  are available to use as ‘dummy’ values for the non-node cases. As observed in [4], these techniques work unproblematically for simple types but require more care, not to mention more work, in the dependent case.

Surely one must be at least a little suspicious of an approach which requires us to define predecessor projections for a whole family when we intend only to hit one constructor. Can we not build sharper tools by programming with dependent types? We propose an alternative ‘two-level’ approach: first, compute the relevant proposition relating each pair of values, then show that it holds when the values are equal. The former is readily computed by nested **treeCase**:

$$\frac{t, t' : \text{Tree}}{\mathbf{TreeNoConfusion} \ t \ t' : \star}$$

$$\begin{aligned} \mathbf{TreeNoConfusion} \ \text{leaf} \quad \text{leaf} &\Rightarrow \forall P \Rightarrow P \rightarrow P \\ \mathbf{TreeNoConfusion} \ \text{leaf} \quad (\text{node } l' \ r') &\Rightarrow \forall P \Rightarrow P \\ \mathbf{TreeNoConfusion} \ (\text{node } l \ r) \ \text{leaf} &\Rightarrow \forall P \Rightarrow P \\ \mathbf{TreeNoConfusion} \ (\text{node } l \ r) \ (\text{node } l' \ r') \\ &\Rightarrow \forall P \Rightarrow (l = l' \rightarrow r = r' \rightarrow P) \rightarrow P \end{aligned}$$

In each case, the statement of ‘no confusion’ takes the form of an elimination rule. Given a proof of  $t = t'$  with constructors on both sides, **TreeNoConfusion**  $t \ t'$  computes an appropriate inversion principle, ready to apply to the goal at hand. We give an ‘interactive’ presentation of the proof:

$$\text{?treeNoConfusion} : \forall t, t' : \text{Tree} \Rightarrow t = t' \rightarrow \mathbf{TreeNoConfusion} \ t \ t'$$

The first step is to eliminate the equation—it is homogeneous—substituting  $t$  for  $t'$  and leaving the ‘diagonal’ goal:

$$\text{?treeNoConfDiag} : \forall t : \text{Tree} \Rightarrow \mathbf{TreeNoConfusion} \ t \ t$$

Now apply **treeCase**  $t$ , leaving problems we can readily solve

$$\begin{aligned} \text{?treeNoConfLeaf} &: \mathbf{TreeNoConfusion} \ \text{leaf} \ \text{leaf} \\ &\cong \forall P \Rightarrow P \rightarrow P \\ \text{proof} &\quad \lambda P; p \Rightarrow p \\ \text{?treeNoConfNode} &: \forall l; r \Rightarrow \mathbf{TreeNoConfusion} \ (\text{node } l \ r) \ (\text{node } l \ r) \\ &\cong \forall l; r; P \Rightarrow (l = l \rightarrow r = r \rightarrow P) \rightarrow P \\ \text{proof} &\quad \lambda l; r; P; p \Rightarrow p \vec{\text{refl}} \end{aligned}$$

It is reassuring that we have no need of specific refutations for the impossible  $n^2 - n$  off-diagonal cases, nor need we construct these troublesome predecessors.

The generalization is straightforward: firstly, iterating **famCase** yields the matrix of constructor cases; for equal constructors, assert that equality of the projections can be used to solve any goal; for unlike constructors, assert that any goal holds.

$$\frac{\vec{x}, \vec{y} : \langle \text{Fam} \rangle}{\mathbf{FamNoConfusion} \vec{x} \vec{y} : \star}$$

...

$$\mathbf{FamNoConfusion} (\text{con } \vec{a}) (\text{con } \vec{b}) \Rightarrow \forall P \Rightarrow (\vec{a} = \vec{b} \rightarrow P) \rightarrow P$$

...

$$\mathbf{FamNoConfusion} (\text{chalk } \vec{a}) (\text{cheese } \vec{b}) \Rightarrow \forall P \Rightarrow P$$

...

Now, to prove

$$\begin{aligned} ?\mathbf{famNoConfusion} : \forall \vec{x} : \langle \text{Fam} \rangle ; \vec{y} : \langle \text{Fam} \rangle \Rightarrow \\ \vec{x} = \vec{y} \rightarrow \mathbf{FamNoConfusion} \vec{x} \vec{y} \end{aligned}$$

first eliminate the equations in left-to-right order—at each stage the leftmost equation is certain to be homogeneous, with each successive elimination unifying the types in the next equation. This leaves

$$?\mathbf{famNoConfDiag} : \forall \vec{x} : \langle \text{Fam} \rangle \Rightarrow \mathbf{FamNoConfusion} \vec{x} \vec{x}$$

which reduces by **famCase**  $\vec{x}$  to an easy problem for each case

$$\begin{aligned} \dots \\ ?\mathbf{famNoConfCon} : \forall \vec{a} \Rightarrow \mathbf{FamNoConfusion} (\text{con } \vec{a}) (\text{con } \vec{a}) \\ \cong \forall \vec{a}; P \Rightarrow (\vec{a} = \vec{a} \rightarrow P) \rightarrow P \\ \text{proof} \quad \lambda \vec{a}; P; p \Rightarrow p \vec{\text{refl}} \\ \dots \end{aligned}$$

We remark that the ‘pattern matching rules’ for **famNoConfusion** hold computationally:

$$\begin{aligned} \dots \\ \mathbf{famNoConfusion} \vec{\text{refl}} \text{refl}_{(\text{con } \vec{a})} P p \Rightarrow p \vec{\text{refl}} \\ \dots \end{aligned}$$

## 6 Acyclicity

Inductive data structures admit no cycles—this is intuitively obvious, but quite hard to establish. We must be able to disprove all equations  $x = t$  where  $x$  is *constructor-guarded* in  $t$ . It is deceptively easy for natural numbers. This goal,

$$? : \forall x : \text{Nat} \Rightarrow x \neq \text{suc} (\text{suc} (\text{suc } x))$$

(where  $x \neq y$  abbreviates  $x = y \rightarrow \forall P \Rightarrow P$ ) is susceptible to induction on  $x$ , but only because one `suc` looks just like another. Watch carefully!

$$\begin{aligned} ? : \text{zero} &\neq \text{suc} (\text{suc} (\text{suc zero})) \\ ? : \forall x : \text{Nat} &\Rightarrow x \neq \text{suc} (\text{suc} (\text{suc } x)) \rightarrow \\ &\boxed{\text{suc } x} \neq \text{suc} (\text{suc} (\text{suc} (\boxed{\text{suc } x}))) \end{aligned}$$

The base case follows by ‘constructors disjoint’. The conclusion of the step case reduces by ‘constructors injective’ to the hypothesis, but only because the boxed `sucs` introduced by the induction are indistinguishable from the `sucs` arising from the goal. The proof is basically a ‘minimal counterexample’ argument—given a minimal cyclic term, rotate the cycle to create a smaller cyclic term—but here we get away with just one third of a rotation.

In general, suppose given a cyclic equation  $x = \text{con}_1(\text{con}_2(\dots(\text{con}_n(x))))$ . Then  $x = \text{con}_1(y)$  where  $y = \text{con}_2(\dots(\text{con}_n(x)))$ , and hence by substitution,  $y = \text{con}_2(\dots(\text{con}_n(\text{con}_1(y))))$  is another cyclic equation, governing a term  $y$  below  $x$  in the sub-term ordering. Repeating this trick  $n$  times yields a term  $z$  below  $x$  satisfying the original cyclic equation  $z = \text{con}_1(\text{con}_2(\dots(\text{con}_n(z))))$ . Well-foundedness of the sub-term ordering then yields the required contradiction.

In order to formalise this intuition, let’s prove

$$? : \forall x : \text{Tree} \Rightarrow x \neq \text{node} (\text{node leaf } x) \text{ leaf}$$

Proceeding by **treeRec**, we get

$$\begin{aligned} ? : \forall x : \text{Tree} &\Rightarrow (\lambda y \Rightarrow y \neq \text{node} (\text{node leaf } y) \text{ leaf}) \text{ belowTree } x \rightarrow \\ &x \neq \text{node} (\text{node leaf } x) \text{ leaf} \end{aligned}$$

Now use **treeCase** twice to dig out the cycle. First we get

$$\begin{aligned} ? : \dots &\rightarrow \text{leaf} \neq \text{node} (\text{node leaf leaf}) \text{ leaf} \\ ? : \forall l, r &\Rightarrow \dots \rightarrow \text{node } l \text{ } r \neq \text{node} (\text{node leaf } (\text{node } l \text{ } r)) \text{ leaf} \end{aligned}$$

We suppress the tuples induced by **treeRec** for the moment. The first subgoal is just ‘constructors disjoint’. In the second case, we can follow the ‘cycle-path’, doing **treeCase**  $l$ . Again, we get a trivial off-path leaf case, and an interesting node case:

$$\begin{aligned} ? : \forall ll, lr, r &\Rightarrow \\ &(\lambda y \Rightarrow y \neq \text{node} (\text{node leaf } y) \text{ leaf}) \text{ belowTree } (\text{node } ll \text{ } lr) \text{ } r \rightarrow \\ &\quad \text{node } ll \quad lr \quad r \\ &\neq \text{node} (\text{node leaf } (\text{node } ll \text{ } lr) \text{ } r) \text{ leaf} \end{aligned}$$

Now we have exposed the entire cycle-path, taking us to  $lr$ . The  $\dots$  **belowTree**  $\dots$  computes to a tuple containing a proof of  $lr \neq \text{node} (\text{node leaf } lr) \text{ leaf}$ . Expanding  $\neq$  and applying constructors reveals equational hypotheses  $ll = \text{leaf}$ ,  $r = \text{leaf}$  and  $lr = \text{node} (\text{node } ll \text{ } lr) \text{ } r$ . Substituting the first two in the third allows us to extract a contradiction from the inductive hypothesis.

## 6.1 General Acyclicity

Funny how the  $\dots$  **belowTree**  $\dots$  computes to a tuple containing the contradiction as soon as we expose the cycle-path! Perhaps we can exploit this behaviour more directly. If we had a proof of

$$(x \neq) \text{ belowTree node (node leaf } x) \text{ leaf}$$

we should certainly acquire a handy proof of  $x \neq x$  somewhere in the resulting tuple. For the sake of brevity, let us write

$$\begin{aligned} x \not\prec t & \text{ for } (x \neq) \text{ belowTree } t \\ x \not\preceq t & \text{ for } (x \neq) \text{ belowTree } t \wedge x \neq t \end{aligned}$$

Note that  $x \not\prec \text{node } l r$  is definitionally equal to  $x \not\preceq l \wedge x \not\preceq r$ . We now have the tools we need to state and prove a *general* acyclicity theorem for **Tree**:

$$? : \forall x, t : \text{Tree} \Rightarrow x = t \rightarrow x \not\prec t$$

Eliminating the equation, we get the statement ‘ $x$  is unequal to any of its proper subterms’:

$$? : \forall x : \text{Tree} \Rightarrow x \not\prec x$$

Unfortunately, we cannot build this tuple structure the way our **below** did in our construction of **treeRec**—inequality to  $x$  is not a hereditary property of trees! We must do induction.

$$\begin{aligned} ? : \text{One} \\ ? : \forall s, t \Rightarrow s \not\prec s \rightarrow t \not\prec t \rightarrow (\text{node } s t \not\preceq s \wedge \text{node } s t \not\preceq t) \end{aligned}$$

The base case is trivial. The step case splits in two, either following the  $s$ -path or the  $t$ -path.

$$\begin{aligned} ? : \forall s, t \Rightarrow s \not\prec s \rightarrow \text{node } s t \not\preceq s \\ ? : \forall s, t \Rightarrow t \not\prec t \rightarrow \text{node } s t \not\preceq t \end{aligned}$$

Each of these is an instance of

$$? : \forall \dots \Rightarrow x \not\prec x \rightarrow \delta x \not\preceq x$$

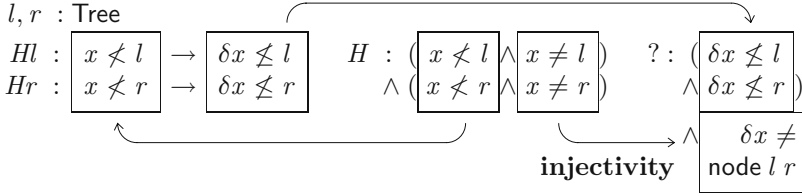
for some variable  $x$  (respectively  $s$  and  $t$ ) and some constructor-form increment<sup>3</sup> of it,  $\delta x$  (either  $(\text{node } x t)$  or  $(\text{node } s x)$ ). The proof is, sad to say, a little crafty: in each case, fix  $x$  and  $\delta x$ , then generalize the bound:

$$? : \forall b \Rightarrow x \not\prec b \rightarrow \delta x \not\preceq b$$

If  $x$  is not below  $b$ , its increment is certainly neither below nor equal to  $b$ . This goes by induction on  $b$ . The base case is trivial, for certainly  $\delta x \neq \text{leaf}$  and there is nothing below  $b$ . In the step case, we introduce hypotheses and

<sup>3</sup> Readers interested in ‘constructor-form increments’ may find [17, 1] useful.

expand definitions selectively. The boxes and arrows show the way the proof fits together.



For a start, we know that  $x$  is not a proper subterm of either  $l$  or  $r$ , so we may certainly deduce that  $\delta x$  is not a subterm of either, and hence not a proper subterm of `node l r`. However, we also know that  $x$  does not equal either  $l$  or  $r$ , hence regardless of whether  $\delta x$  is `node x t` or `node s x`, it does not equal `node l r`.

This construction generalizes readily to inductive families, as soon as we construct *telescopic inequality*. If  $\vec{x} : \Delta$  and  $\vec{y} : \Delta$ , then let  $\vec{x} \neq \vec{y}$  be  $\forall(\vec{x} = \vec{y}) \Rightarrow \forall P \Rightarrow P$ . We may then compare *sequences in Fam*, taking  $\vec{x} \not\prec \vec{y}$  to be  $(\vec{x} \neq)$  **belowFam**  $\vec{y}$  as before, with  $\vec{x} \not\preceq \vec{y}$  being  $\vec{x} \not\prec \vec{y} \wedge \vec{x} \neq \vec{y}$ . This is a suitably heterogeneous notion of ‘not a (proper) subterm’, and it is entirely compatible with the key lemmas—one for each recursive argument  $v_i : \forall \Psi_i \Rightarrow \text{Fam } \vec{r}_i$  of each constructor `con`  $\Delta \vec{v} : \text{Fam } \vec{s}$ .

$$\forall \Psi_i \Rightarrow \forall \vec{b} : \langle \text{Fam} \rangle \Rightarrow \vec{r}_i; (v_i \Psi_i) \not\prec \vec{b} \rightarrow \vec{s}; (\text{con } \Delta \vec{v}) \not\preceq \vec{b}$$

Note that in the higher-order case, an increment steps from a single arbitrary  $x = v_i \Psi_i$  to the node which contains it  $\delta x = \text{con } \Delta \vec{v}$ . The proof of this lemma is again by **famInd**  $\vec{b}$ , and it goes more or less as before. Once again,  $x$  is not a proper subterm of  $b$ ’s subnodes, so  $\delta x$  is not a subterm of  $b$ ; moreover  $\delta x$  is not equal to  $b$  because either  $b$  is made with a constructor other than `con`, or its subnodes are all distinct from  $x$ .

Of course, in the higher-order case—infinately branching trees—it is not generally possible to search mechanically for a cycle. However, the theorem still holds, and any cycle the user can exhibit will deliver an absurdity. In the first-order case, one need merely search the tuple  $\vec{x} \not\prec \vec{t}$  for a proof of  $\vec{x} \neq \vec{x}$ .

## 7 Conclusions and Further Work

We have shown how to construct all the basic apparatus we need for structurally recursive programming with dependent families as proposed by Coquand [7] and implemented in Alf [19]. When pattern matching on an element from a specific branch of an inductive family,  $x : \text{Fam } \vec{t}$ , one may consider only the cases where the  $\vec{t}$  coincide with the indices of a constructor’s return type `con`  $\Delta \vec{v} : \text{Fam } \vec{s}$ .

However, where Alf relied on a syntactic criterion for constructor-guarded recursion and an unspecified external notion of unification, we have constructed all of this technology from the standard induction principles, together with heterogeneous equality. We represent the unification constraints as equational hypotheses

$\vec{s} = \vec{t}$  and reduce them where possible by the procedure given in [20], which is complete for all first-order terms composed of constructors and variables.

We have not shown here how to extend these constructions to *mutual* inductive definitions. These may always be simulated by a single inductive family indexed over a choice of mutual branch, but it is clear that a more direct treatment is desirable. We expect this to be relatively straightforward. The ‘no confusion’ result should extend readily to coinductive data as it relies only on case analysis. We should also seek an internal construction of guarded corecursion, underpinning the criteria in use today [12].

In the more distant future, we should like to see the computational power of our type systems working even harder for us: we have given the general form of our constructions, but they must still be rolled out once for each datatype by a metaprogram which is part of the system’s implementation, manipulating the underlying data structure of terms and types. However convenient one’s tools for such tasks [24], it would still be preferable to perform the constructions once, generically with respect to a universe of inductive families. Research adapting generic functional programming [15] for programs and proofs in Type Theory is showing early promise [3, 2].

Moreover, we might hope to exploit reflection to increase the size of each reasoning step. At the moment our ‘no confusion’ theorem computes how to simplify a single equation. This is used as a component in a unification tactic, but we might hope to reflect the entire unification process in a single theorem, reducing a system of equations to their simplest form.

The main point, however, is that our intuitive expectations of constructors have been confirmed, so we should no longer need to worry about them. The Epigram language and system [25, 23] takes these constructions for granted. We see no reason why the users of other systems should work harder than we do.

## References

1. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride.  $\partial$  for data: derivatives of data structures. *Fundamenta Informaticae*, 2005.
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.
3. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265–269, 2003.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
5. Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.

6. Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First IEEE Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pages 227–236, 1986.
7. Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.
8. Cristina Cornes and Delphine Terrasse. Automating Inversion of Inductive Predicates in Coq. In *Types for Proofs and Programs, '95*, volume 1158 of *LNCS*. Springer-Verlag, 1995.
9. Nicolas G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
10. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
11. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, '94*, volume 996 of *LNCS*, pages 39–59. Springer-Verlag, 1994.
12. Eduardo Giménez. Structural Recursive Definitions in Type Theory. In *Proceedings of ICALP '98*, volume 1443 of *LNCS*. Springer-Verlag, 1998.
13. Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994. Available from <http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-304/>.
14. Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.
15. Ralf Hinze, Johan Jeuring, and Andres Löf. Type-indexed data types. *Science of Computer Programming*, 51:117–151, 2004.
16. Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proc. Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France)*, pages 208–212. IEEE Computer Society Press, 1994.
17. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
18. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
19. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.
20. Conor McBride. Inverting inductively defined relations in LEGO. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, '96*, volume 1512 of *LNCS*, pages 236–253. Springer-Verlag, 1998.
21. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from <http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/>.
22. Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.
23. Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2005+. Revised lecture notes from the International Summer School in Tartu, Estonia.

24. Conor McBride and James McKinna. Functional Pearl: I am not a Number: I am a Free Variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Haskell Workshop 2004, Snowbird, Utah*. ACM, 2004.
25. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
26. Christine Paulin-Mohring. Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation Thesis. Université Claude Bernard (Lyon I), 1996.
27. Jan Smith. The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes. *Journal of Symbolic Logic*, 53(3), 1983.
28. Thomas Streicher. Investigations into intensional type theory. Habilitation Thesis, Ludwig Maximilian Universität, 1993.



# Tactic-Based Optimized Compilation of Functional Programs

Thomas Meyer<sup>1</sup> and Burkhart Wolff<sup>2</sup>

<sup>1</sup> Universität Bremen, Germany

<sup>2</sup> ETH Zürich, Switzerland

**Abstract.** Within a framework of correct code-generation from HOL-specifications, we present a particular instance concerned with the optimized compilation of a lazy language (called *MiniHaskell*) to a strict language (called *MiniML*).

Both languages are defined as shallow embeddings into denotational semantics based on Scott's cpo's, leading to a derivation of the corresponding operational semantics in order to cross-check the basic definitions.

On this basis, translation rules from one language to the other were formally derived in Isabelle/HOL. Particular emphasis is put on the optimized compilation of function applications leading to the side-calculation inferring e.g. strictness of functions.

The derived rules were grouped and set-up as an instance of our generic, tactic-based translator for specifications to code.

## 1 Introduction

The verification of compilers, or at least the verification of compiled code, is known to be notoriously difficult. This problem is still an active research area [3, 4, 12]. In recent tools for formal methods, the problem also re-appears in the form of code-generators for specifications — a subtle error at the very end of a formal development of a software system may be particularly frustrating and damaging for the research field as a whole.

In previous work, we developed a framework for *tactic-based* compilation [5]. The idea is to use a theorem prover itself as a tool to perform source-to-source transformations, controlled by tactic programs, on programming languages embedded into a HOL prover. Since the source-to-source transformations can be derived from the semantics of the program languages embedded into the theorem prover, our approach can guarantee the correctness of the compiled code, provided that the process terminates successfully and yields a representation that consists only of constructs of the target language. Constructed code can be efficient, since our approach can be adopted to optimized compilation techniques, too.

In this paper, we discuss a particular instance of this framework. We present the semantics of two functional languages, a Haskell-like language and an ML-like language for which a *simple* one-to-one translator to SML code is provided. We apply the shallow embedding technique for these languages [1] into standard denotational semantics — this part of our work can be seen as a continuation

of the line of “Winskel is almost right”-papers [8], which formalize proofs of a denotational semantics textbook [11–chapter 9].

As a standard translation, a lazy language can be transformed semantically equivalent via continuation passing style [2] into an eager language. While this compilation is known to produce fairly inefficient code, we also use derived rules for special cases requiring strictness- or definedness analysis. While we admit that the basic techniques are fairly standard in functional compilers, we are not aware of any systematic verification of the underlying reasoning in a theorem prover. Thus, we see here our main contribution.

The plan of the paper is as follows: After a brief outline of the general framework for tactic based compilation and a brief introduction into the used theories for denotational semantics, we discuss the embeddings of MiniHaskell and MiniML into them. These definitions lead to derivations of “classical” textbook operational semantics. In the sequel, we derive transformation rules between these two languages along the lines described by our framework. Then we describe the side-calculus to infer strictness required for optimized compilation; an analogous calculus for definedness is omitted here.

## 2 Background

### 2.1 Concepts and Use of Isabelle/HOL

Isabelle [9] is a generic theorem prover of the LCF prover family; as such, we use the possibility to build programs performing symbolic computations over formulae in a logically safe (conservative) way on top of the logical core engine: this is what our code-generator technically is. Throughout this paper, we will use Isabelle/HOL, the instance for Church’s higher-order logic. Isabelle/HOL offers support for data types, primitive and well-founded recursion, and powerful generic proof engines based on higher-order rewriting which we predominantly use to implement the translation phases of our code-generator.

Isabelle’s type system provides parametric polymorphism enriched by type classes: It is possible to constrain a type variable  $\alpha :: \mathbf{order}$  to specify that an operator  $\_ \leq \_$  must be declared on any  $\alpha$ ; this syntactic concept known from languages such as Haskell is extended in Isabelle by semantic constraints: the operator must additionally fulfill the properties of a partial order.

The proof engine of Isabelle is geared towards rules of the form  $A_1 \Rightarrow (\dots \Rightarrow (A_n \Rightarrow A_{n+1}) \dots)$  which can be interpreted as “from assumptions  $A_1$  to  $A_n$ , infer conclusion  $A_{n+1}$ ”. This corresponds to the textbook notation

$$\frac{A_1 \quad \dots \quad A_n}{A_{n+1}}$$

which we use throughout this paper.

Inside these rules, the meta-quantifier  $\bigwedge$  is used to capture the usual side-constraint “ $x$  must not occur free in the assumptions” for quantifier rules; meta-quantified variables can be considered as “fresh” free variables.

## 2.2 The Framework for Code-Generation

Our generic framework [5] is designed to cope with various executability notions and to provide technical support for them. The following diagram in figure 1 represents the particular instance of the general framework discussed in this paper.

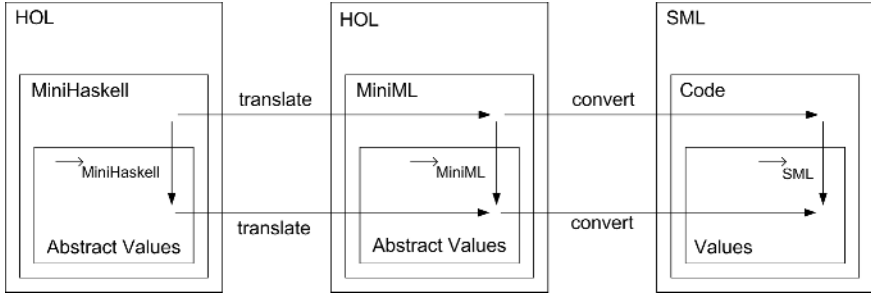


Fig. 1. Basic Concepts

Here, the left block represents the language `MiniHaskell`, the center block the language `MiniML`, which are both presented as conservative shallow embedding into a theory of Scott Domains described in Section 2.3. A subset of both languages are the set of *abstract values*. The embeddings are mirrored by the corresponding terms of a (concrete) programming language, i.e. `SML`, and its subset of (concrete) *values* like e.g. the integers  $1, 2, 3, \dots$ . The first two worlds are connected by the **translate** function, that consists of several tactics that control the translation process by source-to-source translation rules. The latter two worlds are connected by the code-generation function **convert** provided by our framework that is required to be total on the domain of abstract programs.

The three relations  $\rightarrow_{\text{MiniHaskell}}$ ,  $\rightarrow_{\text{MiniML}}$  and  $\rightarrow_{\text{SML}}$  represent the operational semantics of the three languages. We require that they represent partial functions from programs to values. These operational semantics serve as cross-check of our denotational definitions of the language; in particular,  $\rightarrow_{\text{SML}}$  can be compared against an (abstracted) version of the *real SML* semantics [6] in order to validate **convert**. Making these two diagrams commute (while the first commutation is based on formal proofs presented in this paper) constitutes the correctness of our overall translation process.

## 2.3 Denotational Semantics in HOL

The cornerstone of any denotational semantics is its fixpoint theory that gives semantics to systems of (mutual) recursive equations. The well-known Scott-Strachey-approach is based on complete partial orders (*cpo*'s); variants thereof have also been used in standard semantics textbooks such as [11] to give semantics to the languages we discuss here (cf. chapter 9).

Several versions of denotational semantics theories are available for Isabelle [7, 10]. In both, the type class mechanism is used in order to model cpo's, which provide a least element  $\perp$  and completeness on any type belonging to class cpo. This is essentially captured in the theory [10] underlying this work in the axiomatic class definition

```

axclass
  cpo < cpo0
  least       $\perp \leq x$ 
  complete   directed X  $\Rightarrow (\exists b. X \ll b)$ 

```

i.e. completeness means that for any directed set (any non-empty set where two elements have a supremum) there exists a least upper bound.

Moreover, in this type class a number of key concepts such as definedness and strictness of a function and making a function strict are defined:

```

DEF          ::  $\alpha :: \text{cpo0} \Rightarrow \text{bool}$           DEF x  $\equiv x \neq \perp$ 
is_strict   ::  $(\alpha :: \text{cpo0} \Rightarrow \beta :: \text{cpo0}) \Rightarrow \text{bool}$ 
              is_strict f  $\equiv (f \perp = \perp)$ 
strictify   ::  $(\alpha \Rightarrow \beta :: \text{cpo}) \Rightarrow \alpha \perp \Rightarrow \beta$ 
              strictify f x  $\equiv \text{if DEF}(x) \text{ then } f(x) \text{ else } \perp$ 

```

Further, a type constructor can be defined that assigns to each type  $\tau$  a *lifted type*  $\tau_{\perp}$  by disjointly adding the  $\perp$ -element. All types lifted by this type constructor are automatically in the type class cpo but not necessarily vice versa. The function  $\lfloor \_ \rfloor : \alpha \rightarrow \alpha_{\perp}$  denotes the injection, the function  $\lceil \_ \rceil : \alpha_{\perp} \rightarrow \alpha$  its inverse, extended by  $\lceil \perp \rceil = \perp$ .

On cpo's, the usual fixpoint combinator **fix** is defined that is shown to possess the crucial fixpoint property

$$\frac{\text{cont } f}{\text{fix } f = f(\text{fix } f)}$$

for all functions  $f$  that are continuous. Further, there is the usual induction principle for all fixpoints of all types belonging to class cpo:

$$\frac{\text{cont } f \quad \text{adm } P \quad \bigwedge x. P(f \ x)}{P(\text{fix } f)}$$

where the second-order predicate **adm** for *admissibility* captures that a predicate  $P$  holds for a fixpoint if it holds for any approximation of it. **adm** distributes over universal quantification, conjunction and disjunction, but not necessarily over negation. Being defined is an admissible predicate, being total not. As a consequence of induction, we derived a kind of bi-simulation principle:

$$\frac{\text{cont } f \quad \text{cont } f' \quad \bigwedge x. f \ x = f' \ x \quad \bigwedge x. P(f \ x) \quad \text{adm } P}{\text{fix } f = \text{fix } f'}$$

which is the key for the proof of several crucial inference principles over recursive programs to be described in the subsequent sections. If some property  $P$  is invariant through execution of the body  $f$ , then  $P$  can be assumed for the “inner call” when proving the bodies  $f$  and  $f'$  equivalent over them.

### 3 The Semantics of MiniHaskell and MiniML

#### 3.1 The Denotational Semantics of MiniHaskell

Based on the theories of denotational semantics, we define our first contribution — the formal definition of the lazy language MiniHaskell. The types of basic operations like `Bool` were lifted from HOL types

```
types
  Bool = bool⊥   Nat = nat⊥   Unit = unit⊥
```

and basic constants such as `TRUE` or `ONE` are defined accordingly by

```
TRUE :: Bool   TRUE ≡ [True]
ONE  :: Nat    ONE ≡ [1]
```

The core of the MiniHaskell semantics consists of the definitions for the abstraction, application, conditional and the `LET`-construct. As well-known in the literature, an important difference between the denotational theory and the object language has to be made: the abstraction in MiniHaskell is a *value* — a so-called *closure* — and not a function space. Thus, a naive identification of the object language `LAM` with the meta language  $\lambda$  results in a completely wrong model of the operational behaviour: the expression `LAM x. ONE DIV ZERO` should be a *value*, i.e. different from  $\lambda x. 1 \text{ DIV } 0$ , which is just  $\lambda x. \perp$  or just  $\perp$  in the function space. Consequently, the *lifted* function space is used, defined by:

$$\text{types } (\alpha, \beta) \Rightarrow = (\alpha \Rightarrow \beta)_{\perp}$$

which results in the following definitions for the abstraction

```
Lam :: ( $\alpha :: \text{cpo} \Rightarrow \beta :: \text{cpo}$ )  $\Rightarrow$  ( $\alpha \Rightarrow \beta$ )
Lam F ≡ [F]
```

and its inverse, the application

```
 $\triangleright_l :: (\alpha :: \text{cpo} \Rightarrow \beta :: \text{cpo}) \Rightarrow \alpha \Rightarrow \beta$ 
F  $\triangleright_l$  x ≡ [F] x
```

where we may write `LAM x. P x` for `Lam P`. The `LET` construct is just a syntactical shortcut and defined by the application. The remaining definitions of the conditional and the recursor are standard:

```
If :: [Bool,  $\alpha :: \text{cpo}$ ,  $\alpha$ ]  $\Rightarrow$   $\alpha$ 
IF x THEN y ELSE z ≡ case x of
  [v]  $\Rightarrow$  if v then y else z
  |  $\perp$   $\Rightarrow$   $\perp$ 

REC :: ( $\alpha :: \text{cpo} \Rightarrow \alpha$ )  $\Rightarrow$   $\alpha$ 
REC f ≡ fix f
```

The basic operations of MiniHaskell are just strictified versions of the elementary operations of HOL. The paradigmatic example for a 1-ary and a 2-ary function are defined as follows:

```
SUC :: Nat ⇒ Nat
    SUC ≡ strictify(λx. [Suc x])
^<^ :: [Nat, Nat] ⇒ Bool
    (op ^<^) ≡ strictify(λx. strictify(λy. [x<y]))
```

An example for a partial function is DIV:

```
DIV :: [Nat, Nat] ⇒ Nat
    DIV ≡ strictify(λx.
        strictify(λy. if y=0 then ⊥
                    else [x div y]))
```

As top-level constructs, we introduce the following two program definition constructs:

```
VAL  :: [α, α] ⇒ bool
    VAL f E ≡ (f = E)
FUN  :: [α::cpo, α ⇒ α] ⇒ bool
    FUN f F ≡ (f = REC(F)) ∧ cont F
```

This means that a recursive program is representable by the recursor REC of the language MiniHaskell under the condition, that the representing functional F is continuous. The Isabelle syntax engine is set up to parse also mutual recursive function definitions as a combination of fix and pairing. For example, a mutual recursive program in the object language MiniHaskell looks as follows:

```
fun fac x = IF x^=ZERO THEN ONE ELSE x*(fac ▷l (x-ONE))
and add_fac x y = x+fac ▷l y
and suc_fac a = add_fac ▷l ONE ▷l a;
```

Note, that the operators (op +), (op -) and (op \*) are the overloaded (strictified) variants from MiniHaskell.

### 3.2 Lazy Operational Semantics of MiniHaskell

In the following, we derive the operational semantics presented in [11] in order to validate our denotational definitions. The basic concept of this operational semantics is a notion of terms representing values, called *canonical forms*. The judgment  $t \in C_\tau$  states that a term  $t$  is a canonical form of type  $\tau$ . It is defined by the following structural induction on the type  $\tau$ :

Ground type:	$n \in C_{\text{int}} = \{\text{ZERO}, \text{ONE}, \text{TWO}, \dots\}$ and $b \in C_{\text{bool}} = \{\text{TRUE}, \text{FALSE}\}$
Function type:	Closed abstractions are canonical forms, i.e. $(\text{LAM } x. t) \in C_{\tau_1 \rightarrow \tau_2}$ if $t$ is closed

Note, that we can not give an inductive definition for canonical forms since we use a shallow embedding (the types presented above are represented on the

meta-level). Nevertheless, by defining the evaluation relation  $\rightarrow_l$  as equivalent to the logical equality (i.e. evaluation must be correct), we can now derive the rules for the evaluation relation and check that they have the appropriate form  $t \rightarrow_l c$ , where  $t$  is a typeable closed term and  $c$  is a canonical form, meaning  $t$  evaluates to  $c$ . In the following,  $c, c_1, c_2$  and  $c_3$  range over canonical forms:

$$\begin{array}{c}
c \rightarrow_l c \qquad \frac{t_1 \rightarrow_l c_1 \quad t_2 \rightarrow_l c_2}{t_1 \text{ op } t_2 \rightarrow_l c_1 \text{ op } c_2} \\
\\
\frac{t_1 \rightarrow_l \text{TRUE} \quad t_2 \rightarrow_l c_2}{(\text{IF } t_1 \text{ THEN } t_2 \text{ ELSE } t_3) \rightarrow_l c_2} \qquad \frac{t_1 \rightarrow_l \text{FALSE} \quad t_3 \rightarrow_l c_3}{(\text{IF } t_1 \text{ THEN } t_2 \text{ ELSE } t_3) \rightarrow_l c_3} \\
\\
\frac{t_1 \rightarrow_l \text{LAM } x. t \quad t[x := t_2] \rightarrow_l c}{t_1 \triangleright_l t_2 \rightarrow_l c} \qquad \frac{t_2[x := t_1] \rightarrow_l c}{(\text{LET } x = t_1 \text{ IN } t_2 \rightarrow_l c)} \\
\\
\text{REC } y. (\text{LAM } x. t) \rightarrow_l \text{LAM } x. t[y := \text{REC } y. (\text{LAM } x. t)]
\end{array}$$

As can be expected, the rule for canonical forms expresses that canonical forms evaluate to themselves. A key rule is that for the evaluation of applications: the evaluation of an application proceeds by the substitution of the argument into the function body; the treatment of the `LET  $x = t_1$  IN  $t_2$`  is analogously. The rule for recursive definitions unfolds the recursion `REC  $y. (\text{LAM } x. t)$`  once, leading immediately to an abstraction `LAM  $x. t[y := \text{REC } y. (\text{LAM } x. t)]$` , and so a canonical form.

### 3.3 The Denotational Semantics of MiniML

Our semantic interface to the “real” SML target language, the language MiniML, differs with two regards from MiniHaskell:

1. syntactically, all constant symbols in MiniML are followed by a prime, e.g. `ZERO'`, `ONE'`, in order to distinguish them from their counterparts in MiniHaskell. This is for the sake of presentation only.
2. semantically, the two constructs application and `LET` differ from their counterparts in MiniML.

In the sequel, we turn to the semantic issues. In most cases, the semantics of the strict and the lazy constructs are the same. This holds for basic operators like `NOT'` or `SUC'` as well as the abstraction, the conditional and the `REC'` construct. This justifies logical equations such as `NOT'  $\equiv$  NOT` etc.

The crucial difference between the two languages is the strict application. As usual, its denotational definition in MiniML is given by:

$$\begin{array}{l}
\triangleright_s :: (\alpha :: \text{cpo} \Rightarrow \beta :: \text{cpo}) \Rightarrow \alpha \Rightarrow \beta \\
\text{F} \triangleright_s \text{ x} \equiv \text{if } \text{x} = \perp \text{ then } \perp \\
\qquad \text{else if } \text{F} = \perp \text{ then } \perp \text{ else } [\text{F}] \text{ x}
\end{array}$$

The `LET'` construct is defined as usual in terms of abstraction and strict application (enforcing the evaluation of the let-expression prior to the evaluation of its body).

### 3.4 Eager Operational Semantics of MiniML

The rules for the strict evaluation relation  $\rightarrow_s$  is derived analogously to the lazy one  $\rightarrow_l$ . Therefore, we can focus on the differences to MiniHaskell, which are just the rules for the different constructs for the strict application and LET'. In contrast to MiniHaskell, the arguments are first evaluated before performing a substitution:

$$\frac{t_1 \rightarrow_s \text{LAM}' x. t \quad t_2 \rightarrow_s c_2 \quad t[x := c_2] \rightarrow_s c}{t_1 \triangleright_s t_2 \rightarrow_s c}$$

$$\frac{t_1 \rightarrow_s c_1 \quad t_2[x := c_1] \rightarrow_s c}{(\text{LET}' x = t_1 \text{ IN}' t_2) \rightarrow_s c}$$

This concludes our definition and validation of the two languages MiniHaskell and MiniML in terms of a (pre-existing) theory of denotational semantics. In the following, we turn to the semantic translation between these languages by means of derived rules.

## 4 The Semantic Translation

Between the considered languages, the translation of most language constructs is a trivial rewriting due to semantic equivalence. The challenge, however, is the translation of the lazy application to the strict one, and, on the larger scale, the translation of lazy *user-defined* definition constructs to one or more strict versions.

The default solution is well-known and simple: each expression is *delayed* i.e. converted into a closure, and all basic operations were enabled to apply its argument first to the unit-element () in order to *force* the argument closure and to produce an elementary value only when finally needed. Thus, any lazy application can be simulated by an strict one, provided that arguments of applications have been sufficiently delayed.

However, the default solution is fairly inefficient since it delays *any* computation. Therefore, optimizations are mandatory. The principle potentials for such optimizations are

1. the strictness of the function to be applied to an argument (i.e. the argument is used under all possible evaluations) or
2. the definedness of the argument (i.e. delaying is inherently unnecessary).

The concepts discussed above were made precise by a number of combinators which serve either as coding primitive (such as the combinator **delay** and **force**) or as combinators such as **forcify** that represents intermediate states of the translation. We will derive rules that allow to “push” **forcify** combinators throughout a program and thus perform the translation.

In the following, we present these concepts formally. First, we introduce the type constructor **del** for representing delayed, i.e. suspended values:

$$\text{types}$$

$$\alpha \text{ del} = \text{Unit} \Rightarrow \alpha$$



The **delay**-constructor and the corresponding suspension destructor **force** can both be defined completely in terms of our target language MiniML:

$$\begin{aligned} \text{delay} &:: \alpha :: \text{cpo} \Rightarrow \alpha \text{ del} \\ &\quad \text{delay } f \equiv (\text{LAM}' \ x. \ f) \\ \text{force} &:: (\alpha :: \text{cpo}) \text{del} \Rightarrow \alpha \\ &\quad \text{force } f \equiv (f \triangleright_s \text{UNIT}') \end{aligned}$$

Both combinators may remain in final program representations and are treated as primitive by the translation function **convert**.

It turns out that from these definitions the characteristic theorem

$$\text{force } (\text{delay } e) = e$$

can be derived as could be expected.

Now we define the **forcify** combinator that converts a function into its counterpart that deals with delayed values:

$$\begin{aligned} \text{forcify} &:: (\alpha \Rightarrow \beta :: \text{cpo}) \Rightarrow (\alpha \text{ del} \Rightarrow \beta) \\ &\quad \text{forcify } f \equiv \text{LAM}' \ x. \ [f](\text{force } x) \end{aligned}$$

While the **delay** and **force** combinator can be understood as a primitive that can be coded by the converter, **forcify** is a combinator that is uncodable. It is only used internally in the source-to-source translation and has to disappear at the end.

The overall translation process consists of one language translation calculus and three side-calculi — **forcify**-propagation, strictness-reasoning and definedness reasoning, which consist, as mentioned, of derived rules.

#### 4.1 Language Translation Calculus

As mentioned, all but two language constructs have equal semantics can therefore be converted straight-forward by a trivial rewrite rule such as

$$\text{SUC} = \text{SUC}'$$

The key translation rule for the lazy application has the following form:

$$(f \triangleright_l a) = (\text{forcify } f) \triangleright_s (\text{delay } a)$$

This rule states that a lazy application can always be converted into a strict one; the price is the delay of the argument and the necessary forcification of the function of the application. This rule represents the default translation rule, which is — since resulting in inefficient code — avoided whenever possible. The following two rules represent the optimized alternatives of the default scheme: a lazy application is identical with a strict application if its function is strict or if the argument is known to be defined and the function is not the totally undefined one:

$$\frac{\text{is\_strict } f}{(f \triangleright_l a) = (f \triangleright_s a)} \quad \frac{\text{DEF } a \quad \text{DEF } f}{(f \triangleright_l a) = (f \triangleright_s a)}$$

For the LET'-construct, these three cases are analogously. The Isabelle proofs of these rules are not very hard but reveal a number of technicalities that are easily overlooked in paper-and-pencil proofs.

These optimized translation rules lead to side-calculi that attempt to infer the necessary information. One of them, the strictness calculus, will be discussed in the following subsections.

## 4.2 Forcification-Propagation Calculus

In the following, we turn to the key of the default translation to MiniML, the forcification-propagation. The base cases treat identities and constant abstractions as well as basic operators. For the latter, we can assume by construction that they are strict since we only used a particular pattern of their definition built upon `strictify` and `HOL`-functions.

$$\text{forcify } (\text{LAM } x. x) = \text{LAM } x. \text{force } x \quad \text{forcify } (\text{LAM } x. c) = \text{LAM } x. c$$

$$\frac{f \equiv \text{strictify } g}{\text{forcify } (\text{LAM } x. f x) = \text{LAM } x. f (\text{force } x)}$$

$$\frac{\forall f. f \equiv \text{strictify } (\lambda x. \text{strictify } (g x))}{\text{forcify } (\text{LAM } x. f c x) = \text{LAM } x. f c (\text{force } x)}$$

The following rules describe the propagation over the core language constructs for application, abstraction and conditional:

$$\begin{aligned} \text{forcify } (\text{LAM } x. ((f x) \triangleright_l (g x))) = \\ & \text{LAM } x. ( (\text{forcify } (\text{LAM } x. (f x)) \triangleright_l x) \triangleright_l \\ & \quad (\text{forcify } (\text{LAM } x. (g x)) \triangleright_l x) ) \\ \text{forcify } (\text{LAM } x. (\text{LAM } y. (f x y))) = \\ & \text{LAM } x. \text{LAM } y. (\text{forcify}(\text{LAM } x. (f x y)) \triangleright_l x) \\ \text{forcify } (\text{LAM } x. (\text{IF } c x \text{ THEN } f x \text{ ELSE } g x)) = \\ & \text{LAM } x. ( \text{IF } (\text{forcify } (\text{LAM } x. (c x)) \triangleright_l x) \\ & \quad \text{THEN } (\text{forcify } (\text{LAM } x. (f x)) \triangleright_l x) \\ & \quad \text{ELSE } (\text{forcify } (\text{LAM } x. (g x)) \triangleright_l x) ) \end{aligned}$$

Of particular interest is also the rule for the propagation of forcification over the `REC` operator, which allows for the generation of recursive program definitions. In particular, applications like `forcify f` are mapped to the reference  $f'$ , where we assume that for  $f$  there has been the previous statement `fun f x = E` which has been converted to the code-variant `fun f' = forcify (LAM x. E)`. It is automatically proven that this precompiled variant satisfies the property  $(\text{forcify } f) \triangleright_s x = (f' \triangleright_s x)$  which justifies the mapping mentioned above. Thus, “forcified” calls to previously defined functions were mapped to calls of “forcified” definitions.

$$\begin{aligned} \text{forcify } (\text{LAM } x. (\text{REC } (f x))) = \\ \text{LAM } x. \text{REC } (\text{forcify } (\text{LAM } x. (f x)) \triangleright_l x) \end{aligned}$$

For  $n$ -ary functions, analogous rules have to be derived. Moreover, since any function may be strict in the first argument, but not in the second, or vice versa, or non-strict in all arguments, there are  $2^{(n+1)} - 1$  rules for potential forced code variants for direct recursive functions.

### 4.3 Strictness Calculus

As already mentioned, optimized applications require the inference of strictness properties of function bodies. Again, the inference rules follow the cases of our programming language. The base cases treat the identity, the special case of the abstraction yielding  $\perp$  and operations defined upon `strictify`.

$$\text{is\_strict } (\lambda x. x) \quad \text{is\_strict } (\lambda x. \perp)$$

$$\frac{f \equiv \text{strictify } g}{\text{is\_strict } f} \quad \frac{f \equiv \text{strictify } (\lambda x. \text{strictify } (g x))}{\text{is\_strict } (f c)}$$

Note, that the case for the lambda abstraction is omitted since

$$\text{is\_strict } (\lambda x. \text{LAM } y. (E x y))$$

simply does not hold: recall that a closure is a canonical form, hence a value different from  $\perp$ .

Since we suggest a source-to-source translation scheme, the calculus over strictness must cope with terms in which both strict and lazy applications may occur. Therefore, rules for both cases are needed. The inference reduces the applications to semantic functions and substitutes their denotation into it; in the case of the strict application, the argument must be strict in itself:

$$\frac{\text{is\_strict } (\lambda x. [f x] (a x))}{\text{is\_strict } (\lambda x. ((f x) \triangleright_l (a x)))}$$

$$\frac{\text{is\_strict } (\lambda x. [f x] (a x)) \quad \text{is\_strict } (\lambda x. (a x))}{\text{is\_strict } (\lambda x. ((f x) \triangleright_s (a x)))}$$

Note that the computation of the semantic functions  $[f x]$  requires an own (trivial) side-calculus allowing to “push”  $[\_]$  inside; this side-calculus is not presented here.

With respect to the conditional, one gets two cases to establish strictness of the overall construct: either the condition is strict in  $x$  or both branches:

$$\frac{\text{is\_strict } f}{\text{is\_strict } (\lambda x. (\text{IF } (f x) \text{ THEN } (g x) \text{ ELSE } (h x)))}$$

$$\frac{\text{is\_strict } g \quad \text{is\_strict } h}{\text{is\_strict } (\lambda x. (\text{IF } (f x) \text{ THEN } (g x) \text{ ELSE } (h x)))}$$

The most technical proofs of this paper are behind the rules for inferring strictness of recursive schemes and definition constructs. These schemes — which

perform an implicit induction — are consequences of the bi-simulation briefly presented in Section 2.3:

$$\frac{\begin{array}{c} [\text{is\_strict } H] \\ \vdots \\ \text{cont } F \ \wedge \ H. \text{is\_strict } (F \ H) \end{array}}{\text{is\_strict } (\text{REC } F)}$$

This rule performs (for the 1-ary recursive function) a kind of specialized fixpoint induction proof: If we can establish strictness of the body  $F$  provided that a function  $H$  replaced in the recursive call is strict, then the recursor  $\text{REC } F$  yields a function that is strict in its first argument. Note, that for the  $n$ -ary cases similar rules are needed that are omitted here.

## 5 Examples

The calculi are grouped into several sets of rules which were inserted in the Isabelle rewriter. As a result, several tactics are available that perform the translation phases fully automatically.

### 5.1 Example 1

As a first example, we define a function in MiniHaskell whose body consists of a 2-ary lambda abstraction which is strict in its second argument. Its first argument represents an undefined value  $\perp$ :

```
fun f y = (LAM a b. b) ▷l (DIV x ZERO) ▷l y;
```

The first translation phase is able to derive the strictness in the second argument and replaces the second lazy application by a strict one:

```
fun f y = (LAM a b. b) ▷l (DIV x ZERO) ▷s y;
```

The next translation phase replaces the remaining lazy application by our default translation. Recall that a lazy application can always be converted into a strict one by delaying the argument and forcifying the function of the application. Furthermore, a forcification-propagation is performed:

```
fun f y =
  LAM a b. (LAM a. b ▷s delay a) ▷s
           delay (DIV x ZERO) ▷s y;
```

A one-to-one translation is performed by the following translation phase. Each MiniHaskell construct is replaced by its MiniML counterpart yielding a pure MiniML-program:

```
fun' f y =
  LAM' a b. (LAM' a. b ▷s delay a) ▷s
            delay (DIV' x ZERO')
```

The final translation phase performs an optimization by reducing the MiniML-program to the identity:

```
fun' f y = y;
```

## 5.2 Example 2

As a second example, we define the factorial function in MiniHaskell representing a recursive function:

```
fun fac x =
  IF (x ^~=^ ZERO) THEN ONE ELSE x * (fac ▷l (x - ONE));
```

Here, the first translation phase deduces that the function `fac` is strict in its argument and replaces the lazy application in the recursive call by the strict one:

```
fun fac x =
  IF (x ^~=^ ZERO) THEN ONE
  ELSE x * (fac ▷s (x - ONE));
```

Finally, the next phase replaces each MiniHaskell-construct by its corresponding MiniML-counterpart:

```
fun' fac x =
  IF' (EQ' x ZERO')
  THEN' ONE'
  ELSE' TIMES' x (fac ▷s (MINUS' x ONE'));
```

## 6 Conclusion

We address a well-known compilation problem of functional programming. We embed the semantics of both languages into a theory of denotational semantics and derive — as a check of these definitions — the corresponding operational semantics of these languages. The resulting strict semantics can be compared with the semantics of SML [6] and recognized as its abstracted version.

Finally, we derived a couple of rewrite rules that describe the translation of both languages as a source-to-source translation, which is prototypically implemented as a tactic-based compiler finally yielding executable code in SML.

Since the proofs of the translation rules are surprisingly simple (with few exceptions that are interesting in themselves), our approach yields a testbed for the implementation of compilers also for richer languages. Furthermore, it is feasible to develop typical libraries such as lists and compile them with our tactic-based compiler once and for all. Further, our approach may also be relevant to boot-strapping schemes when developing a proven correct compiler.

### 6.1 Further Work

We see the following issues for an extension of our work:

1. *Extending MiniHaskell*: a richer language comprising Cartesian products or lazy data types would help, in particular for the generation of concrete code.
2. *Low level target language*: In principle, our approach can also be applied for the generation of machine-code or JAVA byte-code.

## References

1. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.
2. O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, Dec. 1992.
3. S. Glesner. Using program checking to ensure the correctness of compiler implementations. *JUCS*, 9(3), 2003.
4. G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine and compiler. Technical report, TUM, March 2003.
5. T. Meyer and B. Wolff. Correct code-generation in a generic framework. In M. Aargaard, J. Harrison, and T. Schubert, editors, *TPHOLs 2000: Supplemental Proceedings*, OGI Technical Report CSE 00-009, pages 213–230. Oregon Graduate Institute, Portland, USA, July 2000.
6. R. Milner, M. Tofte, and R. Harper, editors. *The Definition of Standard ML (revised)*. MIT Press, 1997.
7. O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
8. T. Nipkow. Winkler is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.
11. G. Winkler. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.
12. L. Zuck, A. Pnueli, Y. Fang, and B. G. B. A methodology for the translation validation of optimizing compilers. *JUCS*, 9(3), 2003.

# Interfaces as Games, Programs as Strategies

Markus Michelbrink\*

Department of Computer Science,  
University of Wales Swansea,  
Swansea, United Kingdom

[m.michelbrink@swansea.ac.uk](mailto:m.michelbrink@swansea.ac.uk)

<http://www.cs.swan.ac.uk/~csmichel/>

**Abstract.** Peter Hancock and Anton Setzer developed the notion of interface to introduce interactive programming into dependent type theory. We generalise their notion and get an even simpler definition for interfaces. With this definition the relationship of interfaces to games becomes apparent. In fact from a game theoretical point of view interfaces are nothing other than special games. Programs are strategies for these games. There is an obvious notion of refinement which coincides exactly with the intuition. Interfaces together with the refinement relation build a complete lattice. We can define several operators on interfaces: tensor, par, choice, bang etc. Every notion has a dual notion by interchanging the viewpoint of player and opponent. Identifying strategies by some kind of behavioural equivalence we conjecture to receive a linear category. All results so far can be achieved in intensional Martin-Löf Type Theory and are verified in the theorem prover Agda (with the exception of associativity of composition which is only proved on paper until now).

## 1 Introduction

In order to reason about interaction and non-termination in dependent type theory Peter Hancock and Anton Setzer developed the notions of (state dependent) interfaces and interactive programs [8]. Their notion of state independent interface (there is only one state) corresponds to the notion of container of Abbott, Altenkirch, Ghani, McBride [4, 1, 2, 3]. In this paper we propose a generalisation of Hancock/Setzer's notion. A generalised interface or interactive game in our work is given by two sets and two relations between them. With this definition the relationship of interfaces to games becomes apparent. In fact generalised interfaces are nothing other than special games. We can interpret the first set as players, the second as opponents moves and the relations say which moves are allowed after any particular move from the opposite side. Programs for this interfaces are nothing other than strategies for the players.

---

\* Supported by EPSRC grant GR/S30450/01.

We think that our notion of interface is more appropriate for most applications. Stateless networks like the internet seem to be natural application areas for this simplified notion since it does not refer to states. It turns out that under the propositions-as-types interpretation our notion is a generalisation of Hancock/Setzer's notion. States in the definition of Hancock and Setzer can be seen as labels without influence on the behaviour of programs for the interface.

We further propose a simple notion of refinement which corresponds to the notion of linear morphism in the work of Hyvernat/Hancock [7]. Generalised interfaces together with the refinement relation build a complete lattice: Every family of interfaces has a least common refinement and there is an interface such that all members of this family refine this interface and this interface is a refinement for every other interface with this property. A program/strategy on a refinement of an interface gives a strategy on the interface.

There are several ways to build new interfaces from given ones. We introduce the operations tensor, negation and a linear implication in a similar way as in classical game semantics [5, 13]. However whereas in classical game semantics winning strategies are used to compose total strategies, we deal with this problem in a more explicit way, which seems more suitable in a predicative setting. In order to define composition we introduce the notion of fair strategies. A fair strategy is essentially a pair of a strategy and a proof that this strategy plays in both games eventually. By a slight restriction of the notion of interface we are able to define a composition for fair strategies on  $A \multimap B$  and  $B \multimap C$ . The obtained strategy is again fair. We can define fair strategies on  $A \multimap A$  and  $A \otimes (A \multimap B) \multimap B$ . These strategies are versions of copy cat strategies. Identifying strategies by some kind of behavioural equivalence we get a category. We are currently working on a proof in intensional Type Theory that this category is linear.

Unless otherwise stated we work in intensional Martin-Löf Type Theory. However our results can easily be translated into other frameworks. Although our work is carried out in intensional Martin-Löf Type Theory we present our results in an extensional style to improve readability. That means we treat  $B a_0$  and  $B a_1$  as identical types if we have an identity proof  $a_0 \doteq a_1$ , i.e. an inhabitant of the identity type. Further we see  $a_0 \doteq a_1$  and  $a_1 \doteq a_0$  as identical types. We use the following notations:  $t \rightsquigarrow t'$  for  $t$  evaluates to  $t'$ ,  $t \leftrightarrow t'$  for  $t, t'$  evaluate to the same value,  $A$  for the type  $A$  is inhabited,  $id : t \doteq t'$  or  $id : t \doteq_A t'$  for  $id$  is an inhabitant of  $Id A t t'$ . We use the notations  $\Pi(A, B)$ ,  $\Pi(x : A, B x)$ ,  $(x : A) \rightarrow B x$  for the product type and  $\sum(A, B)$ ,  $\sum x : A. B x$  and  $\text{sig } m_0 : A_0 \dots m_n : A_n m_0 \dots m_{n-1}$  for sigma types where in the latter case  $m_0, \dots, m_n$  give access to the components. We denote the canonical elements of this types by  $(a, b)$  or **struct**  $m_0 = a_0; \dots; m_n = a_n$  and switch freely between these notations. The sentential connectives  $\forall, \exists, \wedge, \vee, \Rightarrow$  for this type constructors are used in the standard way to emphasise the reading of types as propositions. We sometimes suppress arguments which can be inferred from other arguments. We use the notations **False** and **True** for the types with zero and one inhabitant, respectively.



## 1.1 Related Work

This paper has many forerunners. In [8] Hancock and Setzer propose a representation of interactive systems in dependent type theory by so called worlds i.e. an interface without a set of states. In [9] the same authors identify interaction systems as coalgebras for certain functors. They investigate the relationship to predicate transformer semantics and the refinement calculus. In [11, 10] it is shown that the introduction rules for weakly final coalgebras for the functors above are a slightly restricted form of guarded induction. In [19] we give a set theoretic model for the type theory enriched by rules for this weakly final coalgebras. Further we define a monadic composition of programs with the possibility to terminate. The author is preparing a paper where state-dependent coalgebras are modelled in intensional type theory. As shown by Hancock/Hyvernat [7] interfaces (interaction structures) seen as predicate transformers give a close connection to formal topology [20]. In fact every interface gives a natural example for a non distributive topology. There is as well a vague connection to the notion of safety and liveness properties of programs [16]. In [14, 15] Hyvernat uses interfaces to give a model of linear logic. Michael Abbott, Thorsten Altenkirch, Neil Ghani and Conor McBride developed the notions of container [4, 1, 2, 3] and its derivative. The derivative of a container is a generalisation of the zipper [12]. The name is motivated by the fact that the derivative behaves like the derivation of a formal power series.

In [6] a related notion of interface is given to treat the question when two software modules are compatible. The authors use pushdown games to model the behaviour of this interfaces.

## 2 Generalised Interfaces

In [8] Hancock and Setzer give the following definition of an Interface: An *interface* is a quadruple  $(S, C, R, n)$  s.t.

- $S : \text{Set}$
- $C : S \rightarrow \text{Set}$
- $R : \prod s : S. C s \rightarrow \text{Set}$
- $n : \prod s : S. \prod c : C s. R s c \rightarrow S$

$S$  is the set of states,  $C s$  the set of commands in state  $s : S$ ,  $R s c$  the set of responses to a command  $c : C s$  in state  $s : S$ , and  $n s c r$  the next state of the system after this interaction.

A *program* for this interface starting in state  $s_0 : S$  is a quadruple  $(A, c, \text{next}, a_0)$  s.t.

- $A : S \rightarrow \text{Set}$
- $c : \prod s : S. A s \rightarrow C s$
- $\text{next} : \prod s : S. \prod a : A s. \prod r : R s (c s a). A (n s (c s a) r)$
- $a_0 : A(s_0)$

$A s$  is the set of programs starting in state  $s$ ,  $c s a$  the command issued by the program  $a : A s$ , and  $\text{next } s a r$  is the program that will be executed, after having obtained for command  $c s a$  the response  $r : R s (c s a)$ . See [8] for further motivations.

Despite the fact that these seem to be quite natural and straightforward notions there are certain drawbacks to this definition. First of all these notions turn out to be technically difficult, particularly if we try to work with them in an intensional setting. What makes working with the interface definition clumsy is that there are too many dependencies. The commands depend on the states, the responses on the commands and the next state on the state, the command and the response. This looks redundant since the information to which state a command belongs should already be given by the command itself etc. Hence the responses should only depend on the command and the next state on the response. This can be achieved by replacing this definition by a more fibration-like (the index-set appears right) definition:

**Definition 1.** *Interface (intermediate)*

*An interface is given by sets  $S, C, R$  and functions  $\text{st} : C \rightarrow S$ ,  $\text{co} : R \rightarrow C$ ,  $\text{next} : R \rightarrow S$ .*

This definition is used in [18] to prove that there is a final coalgebra for the functor induced by the definition of programs above. However in this definition there still appears a set of states  $S$  which may be interpreted as the state of the entire system or the environment. It seems at least questionable if this is an appropriate definition of an interface for the following reasons:

1. An interface for a system should say what can go into the system and what comes out. Depending input and output on the environment leads to unmanageable complex programs.
2. The state of the entire system is often not known and there is no way to refer to it.

If we analyse how the states influence the behaviour of programs in the definition above, then it turns out that regardless of the initial state, the states tell us which commands are available for a program after a certain response. We have as well a direct relationship between commands and responses, which tells us what responses may occur after a certain command. If we generalise this we get the following notion of interface:

**Definition 2.** *A generalised interface  $G$  consist of*

- $C, R : \text{Set}$   
*a set of commands (players moves) and a set of responses (opponents moves),*
- $\llcorner_C : R \rightarrow C \rightarrow \text{Set}$   
*a relation between  $R$  and  $C$ ,*
- $\llcorner_R : C \rightarrow R \rightarrow \text{Set}$   
*a relation between  $C$  and  $R$ .*

*Example 1.* Classical Nim

Let  $n, m : \mathbb{N}$  with  $m < n$  and  $C := R := E_n$  where  $E_n$  the type with canonical elements  $0, \dots, n - 1$ . Let  $\text{diff } e_0 e_1 := (\text{val } e_0) - (\text{val } e_1)$  the modified difference between the values of  $e_0, e_1 : E_n$  and  $e_0 \ll_R e_1 := \Leftrightarrow e_0 \ll_C e_1 := \Leftrightarrow 0 < \text{diff } e_0 e_1 < m$ .

A strategy tells a player his next move and gives a new strategy for every move of the opponent. A sound strategy obeys the laws of the game:

**Definition 3.** A program/strategy is given by

- $X : \text{Set}$   
a set  $X$
- a function  $\text{onestep}$   
giving for every  $x : X$  a command  $\text{command } x : C$  and a function  $\text{next } x : (r : R) \rightarrow c \ll_R r \rightarrow X$ .

A sound strategy is a strategy  $(X, \text{onestep})$  such that:

- $\forall x : X, r : R. \text{command } x \ll_R r \Rightarrow r \ll_C \text{command } (\text{next } x r \_)$ .

*Example 2.* A sound strategy for classical Nim

A well known winning strategy consists of playing multiples of  $m$ :

Let  $X := \sum (e : E_n, m \nmid ((\text{val } e) + 1))$ ,  $\text{command } (e, \_) := e_c$ , where  $e_c$  the unique element with  $\text{val } e_c = (\text{val } e) - (\text{mod } ((\text{val } e) + 1) m)$ . Let  $\text{next } (e, \_) e_r \text{ fit}_r := (e_r, q)$  where  $q$  proves  $m \nmid ((\text{val } e_r) + 1)$ .

The definition above is a generalisation of Hancock/Setzer's notion. We are preparing a paper [17] where this statement is made precise.

### 3 Refinement

There are mainly two ways to understand subsets in type theory. The first is to see a subset of a set  $M$  as  $S : M \rightarrow \text{Set}$ . The idea is that an element  $m$  of  $M$  belongs to the subset  $S$  if  $S m$  is inhabited. The second way is to understand a subset of  $M$  as a function  $f : S \rightarrow M$ . The idea is to interpret  $f : S \rightarrow M$  as the image of  $S$  under  $f$ . There are obvious ways to switch between both conceptions see e.g. the discussion about families and predicates in Michelbrink [18]. We will use the second approach here. A set  $M$  is interpreted as subset of  $M$  by  $\text{id}_M : M \rightarrow M$ .  $f : S \rightarrow M$  is a subset of  $f' : S' \rightarrow M$  iff there is a function  $g : S \rightarrow S'$  with  $f' \circ g = f$  where equality here means pointwise equal. A subset  $g : S' \rightarrow S$  of a subset  $f : S \rightarrow M$  of  $M$  gives a subset  $f \circ g : S' \rightarrow M$  of  $M$  and the subset relation on subsets of  $M$  is the order relation of a complete Heyting algebra.

The idea of refinement is that one side is more restricted in its behaviour whereas the other side has more freedom. If we translate the idea into our setting and decide that the opponent should have more freedom we get the following notion:

**Definition 4.** Given two generalised interfaces  $A = (C_A, R_A, \llcorner_C^A, \llcorner_R^A)$  and  $B = (C_B, R_B, \llcorner_C^B, \llcorner_R^B)$ .  $A$  is a refinement of  $B$  is the type which elements consist of:

– a function

$$\text{ref}_C : C_A \rightarrow C_B,$$

– a function

$$\text{ref}_R : R_B \rightarrow R_A,$$

– and proofs for

$$\text{ref}_R r_b \llcorner_C^A c_a \Rightarrow r_b \llcorner_C^B \text{ref}_C c_a$$

and

$$\text{ref}_C c_a \llcorner_R^B r_b \Rightarrow c_a \llcorner_R^A \text{ref}_R r_b$$

for  $r_b : R_B, c_a : C_A$ .

We write  $A \sqsubseteq B$  for  $A$  is a refinement of  $B$ .

As explicated above  $C_A$  can be seen as subset of  $C_B$  by  $\text{ref}_C$  and  $R_B$  as subset of  $R_A$  by  $\text{ref}_R$  and the weak Galois conditions restrict players and widen opponents possibilities. So the definition is commensurate to our intuition.

**Proposition 1.** If  $A \sqsubseteq B$  and  $(X_A, \text{command}_A, \text{next}_A)$  a sound strategy on  $A$  then there is a sound strategy  $(X_B, \text{command}_B, \text{next}_B)$  on  $B$ .

*Proof.* Let  $X_B := X_A$ ,  $\text{command}_B x := \text{ref}_C(\text{command}_A x)$  and  $\text{next}_B x r_b \text{ rfit}_B := \text{next}_A x (\text{ref}_R r_b) \text{ rfit}_A$  where we obtain  $\text{rfit}_A$  from  $\text{rfit}_B$  by the refinement.  $\square$

**Proposition 2.**  $\sqsubseteq$  is the order relation of a complete lattice.

*Proof:* It is clear that  $\sqsubseteq$  is transitive. Let  $A_i = (C_{A_i}, R_{A_i}, \llcorner_C^{A_i}, \llcorner_R^{A_i})$ ,  $i : I$  be a family of generalised interfaces. Then the infimum  $A$  of  $A_i$ ,  $i : I$  is given by

$$\begin{aligned} R_A &= \sum(I, \lambda i : I. R_{A_i}) \\ C_A &= (i : I) \rightarrow C_{A_i} \\ (i, r) \llcorner_C^A f &:\Leftrightarrow r \llcorner_C^{A_i} f \ i \\ f \llcorner_R^A (i, r) &:\Leftrightarrow f \ i \llcorner_R^{A_i} r \end{aligned}$$

and the supremum  $A$  of  $A_i$ ,  $i : I$  is given by

$$\begin{aligned} R_A &= (i : I) \rightarrow R_{A_i} \\ C_A &= \sum(I, \lambda i : I. C_{A_i}) \\ f \llcorner_C^A (i, c) &:\Leftrightarrow f \ i \llcorner_C^{A_i} c \\ (i, c) \llcorner_R^A f &:\Leftrightarrow c \llcorner_R^{A_i} f \ i. \end{aligned}$$

$\square$

## 4 Operations on Games

There are many different ways to build new games from given ones. We only present the operations needed to form the category we are aiming for. All operations are defined in a way similar to usual game semantics. The dual game  $A^\perp$  (negation, cogame) of  $A$  is defined by changing the rôle of player and opponent. In the tensor game  $A \otimes B$  opponent chooses whether to play in  $A$  or  $B$ .  $A \multimap B$  is defined as  $(A \otimes B^\perp)^\perp$ .

**Definition 5.** – The cogame  $A^\perp$  of a game  $A$  is given by

$$A_C^\perp := A_R, A_R^\perp := A_C, \llcorner_C A^\perp := \llcorner_R^A, \llcorner_R^\perp := \llcorner_C^A.$$

– The tensor  $A \otimes B$  is given by

$$(A \otimes B)_C = A_C \times B_C$$

$$(A \otimes B)_R = (A_R \times B_C) + (A_C \times B_R)$$

$$\text{inl}(r_a, c_b) \llcorner_C^{A \otimes B} (c_a, c'_b) :\Leftrightarrow r_a \llcorner_C^A c_a \quad \wedge \quad c_b \doteq c'_b$$

$$\text{inr}(c_a, r_b) \llcorner_C^{A \otimes B} (c'_a, c_b) :\Leftrightarrow r_b \llcorner_C^B c_b \quad \wedge \quad c_a \doteq c'_a$$

$$(c_a, c'_b) \llcorner_R^{A \otimes B} \text{inl}(r_a, c_b) :\Leftrightarrow c_a \llcorner_R^A r_a \quad \wedge \quad c_b \doteq c'_b$$

$$(c'_a, c_b) \llcorner_R^{A \otimes B} \text{inr}(c_a, r_b) :\Leftrightarrow c_b \llcorner_R^B r_b \quad \wedge \quad c_a \doteq c'_a$$

– Lollipop  $A \multimap B := (A \otimes B^\perp)^\perp$ .

*Note 1.* In the definition of tensor above we have a play in  $A$  and in  $B$  running all the time. There is also a variant of this definition where opponent may start with a play in  $A$  or  $B$  and open a play in the other game later ( $(A \otimes' B)_C = A_C + (A_C \times B_C) + B_C$ ,  $(A \otimes' B)_R = A_R + (A_R \times B_C) + (A_C \times B_R) + B_R$ , ...). We think that this variant is needed to receive a linear category. However for simplicity we work with the definition of tensor above.

*Note 2.* In  $A \multimap B$  player chooses where to play.

**Proposition 3.** ( $\text{CopyCat}_A, \text{command}_A, \text{next}_A$ ) given by

$$\text{CopyCat}_A := A_C + A_R$$

$$\text{command}_A(\text{inl } c_a) := \text{inr}(c_a, c_a) \quad \text{command}_A(\text{inr } r_a) := \text{inl}(r_a, r_a)$$

$$\text{next}_A(\text{inl } c_a)(c_a, r_a)(\text{id}, \text{fit}_r) := \text{inr } r_a \quad \text{next}_A(\text{inr } r_a)(c_a, r_a)(\text{fit}_c, \text{id}) := \text{inl } c_a$$

is a sound strategy on  $A \multimap A$ .

*Proof.* Immediate. □

## 5 Fair Strategies

Our goal is to define a category where the objects are games and the morphisms are sound strategies on  $A \multimap B$ . Given two sound strategies  $\sigma : A \multimap B$  and  $\tau : B \multimap C$  we need to construct a new strategy  $\sigma; \tau : A \multimap C$ . The idea is to play the strategies  $\sigma$  and  $\tau$  against each other in  $B$ . The problem is that this may result in “infinite chattering”: both strategies may play in  $B$  for ever. The usual solution is to use “winning strategies” which implicate that this composition of total strategies is again total. However we prefer to use a more explicit solution. Therefore we demand our strategies to satisfy a certain fairness property, which says that our strategies play in both games eventually. In a constructive setting this means that we not only have to say that our strategy will switch to play in the other game but also when this will happen. And we need to give this information for all possible moves of opponent. This means we have to assign to all states of our strategy  $\sigma$  (elements of the set  $X$ ) two well-founded trees, which say when  $\sigma$  will play the next time in  $A$  or  $B$  respective. We start our construction by defining these well-founded trees. For  $c : (A \multimap B)_C$  let  $ACommand\ c, BCommand\ c : Set$  be defined by

$$ACommand\ (inl\ \_) := BCommand\ (inr\ \_) := True$$

$$ACommand\ (inr\ \_) := BCommand\ (inl\ \_) := False.$$

**Definition 6.** For  $A$  game let

$$\begin{aligned} \text{Beginning } A &:= \text{data stop} \mid \\ &\quad \text{fork}(c : AC)(f : (r : AR) \rightarrow c \ll_R r \rightarrow \text{Beginning } A) \end{aligned}$$

For games  $A, B$ ,  $\sigma = (X, \text{command}, \text{next})$  strategy on  $A \multimap B$ ,  $x : X$  and  $beg : \text{Beginning } A \multimap B$  let

$$\text{maxvalidABeg } beg\ x$$

be

$$BCommand\ (\text{command } x)$$

if  $beg \rightsquigarrow \text{stop}$  and

$$\begin{aligned} &ACommand\ (\text{command } x) \quad \wedge \quad c \doteq (\text{command } x) \quad \wedge \\ &\forall r : (A \multimap B)_R, c \ll_R r \Rightarrow \text{maxvalidABeg } (f\ r\ \_) \ (\text{next } x\ r\ \_) \end{aligned}$$

if  $beg \rightsquigarrow \text{fork } c\ f$ .  $\text{maxvalidBBeg } beg\ x$  is defined similar.

A Beginning is nothing but a well-founded tree which nodes are commands and which branching is given by the responses to each command. A Beginning of  $A \multimap B$  is a maximal play in  $A$  for a strategy  $\sigma$  and  $x : X_\sigma$  if the Beginning is  $\text{stop}$  and  $\sigma$  plays in  $B$  at  $x$  or the top node of the Beginning is the move which  $\sigma$  plays at  $x$ ,  $\sigma$  plays in  $A$  at  $x$  and all subtrees are maximal plays in  $A$  for  $\sigma$  at the corresponding  $x' : X_\sigma$ . A fair strategy is a strategy  $\sigma$  where we can assign a maximal play in  $A$  and a maximal play in  $B$  to every  $x : X_\sigma$ :

**Definition 7.** For games  $A, B$  a fair strategy on  $A \multimap B$  is given by

- a strategy  $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma)$
- and two functions

$$\text{maxABeg}_\sigma, \text{maxBBeg}_\sigma : X \rightarrow (\text{Beginning } A \multimap B)$$

- with

$$\text{maxvalidABeg} (\text{maxABeg}_\sigma x) x$$

and

$$\text{maxvalidBBeg} (\text{maxBBeg}_\sigma x) x$$

*Note 3.* For sound strategies  $\sigma$  exactly one of  $\text{maxABeg}_\sigma x$ ,  $\text{maxBBeg}_\sigma x$  is of the form `stop` whereas the other has the form `fork c f`.

**Corollary 1.**  $(\text{CopyCat}_A, \text{command}_A, \text{next}_A)$  is a fair strategy.

## 6 Composition

The following definition describes the situations when we can run  $\sigma$  against  $\tau$ . We need it to define the set  $X_{\sigma;\tau}$ :

**Definition 8.** For games  $A, B, C$ ,  $c_{ab} : (A \multimap B)_C$ ,  $c_{bc} : (B \multimap C)_C$

$$\text{Link } A \ B \ C \ c_{ab} \ c_{bc}$$

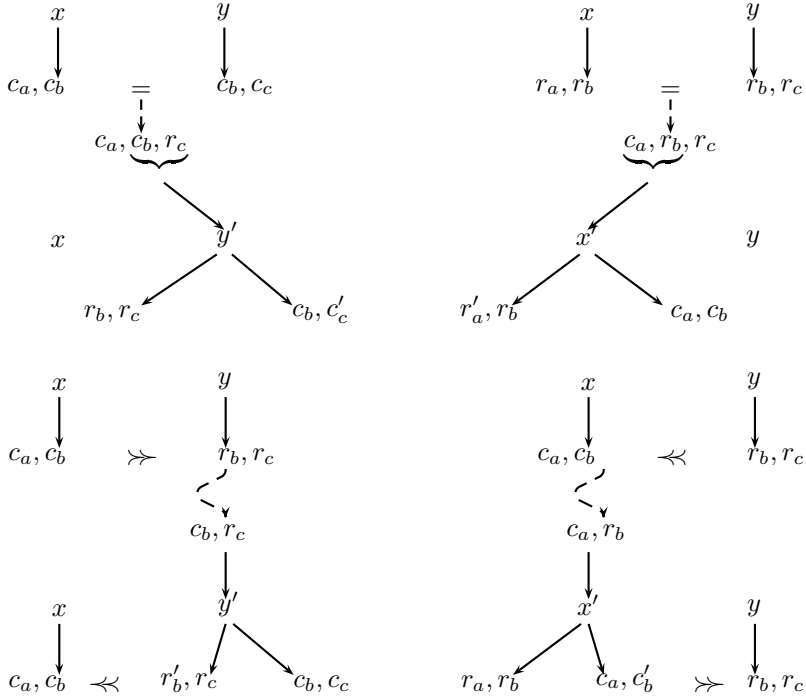
is given by

- $\text{Link } A \ B \ C \ \text{inl}(r_a, r_b) \ \text{inl}(r'_b, r_c) = \text{Id } B_R \ r_b \ r'_b$
- $\text{Link } A \ B \ C \ \text{inl}(r_a, r_b) \ \text{inr}(c_b, c_c) = \text{False}$
- $\text{Link } A \ B \ C \ \text{inr}(c_a, c_b) \ \text{inl}(r_b, r_c) = (c_b \leftarrow_R^B r_b) + (r_b \leftarrow_C^B c_b)$
- $\text{Link } A \ B \ C \ \text{inr}(c_a, c_b) \ \text{inr}(c'_b, c_c) = \text{Id } B_C \ c_b \ c'_b$

**Definition 9.** For fair strategies  $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma) : A \multimap B$  and  $\tau = (X_\tau, \text{command}_\tau, \text{next}_\tau) : B \multimap C$  let  $X_{\sigma;\tau}$  be the set which elements are triples

- $x : X_\sigma$
- $y : X_\tau$
- $\text{link} : \text{Link } A \ B \ C \ (\text{command}_\sigma x) \ (\text{command}_\tau y)$ .

We use the abbreviations  $x \stackrel{\uparrow}{=} y$ ,  $x \leftarrow y$ ,  $x \rightarrow y$  and  $x \stackrel{\downarrow}{=} y$  for  $(x, y, l) : X_{\sigma;\tau}$  according to the shape of  $l$ . The definition above gives the set  $X_{\sigma;\tau}$  of the strategy  $\sigma; \tau := (X_{\sigma;\tau}, \text{onestep}_{\sigma;\tau})$ . It remains to define the function  $\text{onestep}_{\sigma;\tau}$ . This is done by induction on the beginnings  $\text{maxBBeg}_\sigma x$ ,  $\text{maxABeg}_\tau y$  for  $(x, y, l) : X_{\sigma;\tau}$ . The following pictures illustrate the different cases in the definition.



For  $cab : (A \multimap B)_C$ ,  $fit : (c_a, r_b) \ll_C^{A \multimap B} cab$  we define  $compLinkL\ cab\ fit : Link\ cab\ (inl\ (r_b, r_c))$  by

$$\begin{aligned} compLinkL\ (inl\ (r_a, r'_b))\ fit &:= fit.snd \\ compLinkL\ (inr\ (c'_a, c_b))\ fit &:= inr\ fit.fst. \end{aligned}$$

$compLinkR\ cbc\ fit : Link\ (inr\ (c_a, c_b))\ cbc$  is defined analogous for  $cbc : (B \multimap C)_C$ ,  $fit : (c_b, r_c) \ll_C^{B \multimap C} cbc$

**Definition 10.** We define  $onestep_{\sigma;\tau}(x, y, l)$  by cases:

- I.  $command_{\sigma}\ x = inl\ (r_a, r_b)$ .
  - I.1  $command_{\tau}\ y = inl\ (r_b, r_c)$ .

$$\begin{aligned} onestep_{\sigma;\tau}(x, y, l) &:= (inl\ (r_a, r_c), next_{\sigma;\tau}) \\ next_{\sigma;\tau}(x, y, l)\ (c_a, r_c)\ (fit, id) &:= (x', y, l') \end{aligned}$$

where  $x' := next_{\sigma}\ x\ (c_a, r_b)\ (fit, refl\ r_b)$ ,  $l' := compLinkL\ (command_{\sigma}\ x')\ fit'$  &  $fit' : (c_a, r_b) \ll_C^{A \multimap B}\ (command_{\sigma}\ x')$  is given by the soundness proof of  $\sigma$ .

- I.2  $command_{\tau}\ y = inr\ (c_b, c_c)$ .
  - In this case is  $l : False$ .

- II.  $command_{\sigma}\ x = inr\ (c_a, c_b)$ .



II.1  $\text{command}_\tau y = \text{inl}(r_b, r_c)$ .

II.1.a)  $\text{fit}_r : c_b \ll_{\mathbb{R}} r_b$ .

$$\text{onestep}_{\sigma;\tau}(x, y, l) := \text{onestep}_{\sigma;\tau}(x', y, l')$$

where  $x' = \text{next}_\sigma x (c_a, r_b) (\text{fit}_r, \text{refl } c_a)$ ,

$l' := \text{compLinkL}(\text{command}_\sigma x') \text{fit}' \mathcal{E}$

$\text{fit}' : (c_a, r_b) \ll_{\mathbb{C}}^{\mathbb{A} \rightarrow \mathbb{B}} (\text{command}_\sigma x')$  is given by the soundness proof of  $\sigma$ .

II.1.b)  $\text{fit}_c : r_b \ll_{\mathbb{C}} c_b$ . Analog to II.1.a).

II.2  $\text{command}_\tau y = \text{inr}(c_b, c_c)$ . Analog to I.1.

*Note 4.* The definition above is a definition by induction on the maximal plays in  $\mathbb{B}$  assigned to  $x : \mathbb{X}_\sigma$  and  $y : \mathbb{X}_\tau$ . The induction step is in the cases II.1.a) and II.1.b). In II.1.a) the beginning assigned to  $x'$  is smaller than the one assigned to  $x$  whereas in II.1.b) the beginning assigned to  $y'$  is smaller than the one assigned to  $y$ .

**Lemma 1.** *Let  $\sigma : \mathbb{A} \multimap \mathbb{B}$ ,  $\tau : \mathbb{B} \multimap \mathbb{C}$  be fair and sound strategies.*

a) *Let  $\text{command}_\sigma x = \text{inr}(c_a, c_b)$  and  $\text{command}_\tau y = \text{inl}(r_b, r_c)$ .*

*Then  $(c_a, r_c) \ll_{\mathbb{C}} \text{command}_{\sigma;\tau}(x \ll y)$ .*

b) *Let  $\text{command}_\sigma x = \text{inr}(c_a, c_b)$  and  $\text{command}_\tau y = \text{inl}(r_b, r_c)$ .*

*Then  $(c_a, r_c) \ll_{\mathbb{C}} \text{command}_{\sigma;\tau}(x \gg y)$ .*

*Proof.* We prove a) and b) simultaneously by induction on the beginnings of  $x \ll y$ ,  $x \gg y$ .

a) If  $\text{command}_\sigma x' = \text{inl}(r_a, r_b)$  where  $x' = \text{next}_\sigma x r_b \_$  then  $\text{command}_{\sigma;\tau}(x \ll y) = \text{inl}(r_a, r_c)$  with  $c_a \ll r_a$  i.e.  $(c_a, r_c) \ll \text{inl}(r_a, r_c)$ .

If  $\text{command}_\sigma x' = \text{inr}(c_a, c_b)$  where  $x' = \text{next}_\sigma x r_b \_$  then  $\text{command}_{\sigma;\tau}(x \ll y) = \text{command}_{\sigma;\tau}(x' \gg y)$  and by I.H. follows the claim.

b) analog to a). □

**Proposition 4.** *Let  $\sigma : \mathbb{A} \multimap \mathbb{B}$ ,  $\tau : \mathbb{B} \multimap \mathbb{C}$  be fair and sound strategies. Then  $\sigma; \tau : \mathbb{A} \multimap \mathbb{C}$  is sound.*

*Proof.* Let  $(x, y, l) : \mathbb{X}_{\sigma;\tau}$  and  $u\text{auc} : (\mathbb{A} \multimap \mathbb{C})_{\mathbb{R}}$  with  $\text{command}_{\sigma;\tau}(x, y, l) \ll u\text{auc}$ . We must show that  $u\text{auc} \ll \text{command}_{\sigma;\tau}(\text{next}_{\sigma;\tau}(x, y, l) u\text{auc} \_)$ . The proof is by induction on the beginnings of  $x, y$ .

I.  $(x, y, l) = (x \underset{c}{=} y)$ .

Let  $\text{inr}(c_a, c_b) := \text{command}_\sigma x$ ,  $\text{inr}(c_b, c_c) := \text{command}_\tau y$ . Then we have  $\text{inr}(c_a, c_c) = \text{command}_{\sigma;\tau}(x, y, l)$  &  $u\text{auc} = (c_a, r_c)$  with  $c_c \ll r_c$ . Further we have

$$\text{next}_{\sigma;\tau}(x, y, l) u\text{auc} \_ = (x \underset{c}{=} y') \quad (1)$$

$$\text{next}_{\sigma;\tau}(x, y, l) u\text{auc} \_ = (x \ll y') \quad (2)$$

In case (1) we have  $\text{command}_{\sigma;\tau}(x \underset{c}{=} y') = \text{inr}(c_a, c'_c)$  with  $r_c \ll c'_c$  which proves the claim. In case (2) we have

$$\text{command}_{\sigma;\tau}(x \ll y') = \text{command}_{\sigma;\tau}(x' \underset{r}{=} y') \quad \text{or} \quad (3)$$

$$\text{command}_{\sigma;\tau}(x \ll y') = \text{command}_{\sigma;\tau}(x' \gg y') \quad (4)$$

In case (3) we have  $\text{command}_{\sigma;\tau}(x \ll y') = \text{inl}(r_a, r_c)$  with  $c_a \ll r_a$  which proves the claim. In case (4) the claim follows with Lemma 1.

II.  $(x, y, l) = (x \ll y)$ .

Let  $\text{inr}(c_a, c_b) := \text{command}_{\sigma} x$ ,  $\text{inl}(r_b, r_c) := \text{command}_{\tau} y$ . Then we have

$$\begin{aligned} \text{command}_{\sigma;\tau}(x \ll y) &= \text{command}_{\sigma;\tau}(x' \underset{r}{=} y) \quad \text{and} \\ \text{next}_{\sigma;\tau}(x \ll y) \text{uaucl} &= \text{next}_{\sigma;\tau}(x \underset{r}{=} y') \text{uaucl} \quad \text{or} \\ \text{command}_{\sigma;\tau}(x \ll y) &= \text{command}_{\sigma;\tau}(x' \gg y') \quad \text{and} \\ \text{next}_{\sigma;\tau}(x \ll y) \text{uaucl} &= \text{next}_{\sigma;\tau}(x' \gg y) \text{uaucl} \end{aligned}$$

and the claim follows from the induction hypothesis.

III.  $(x, y, l) = (x \gg y)$ . Analog to II.

IV.  $(x, y, l) = (x \underset{r}{=} y)$ . Analog to I. □

**Proposition 5.** *Let  $\sigma : A \multimap B$ ,  $\tau : B \multimap C$  be fair and sound strategies. Then  $\sigma;\tau : A \multimap C$  is fair.*

*Proof.* We only give the construction for  $\text{maxABeg}_{\sigma;\tau}(x, y, l)$ .

The definition is by an outer induction on  $\text{maxABeg}_{\sigma} x$  and an inner induction on  $\text{maxBBeg}_{\sigma} x$  and  $\text{maxABeg}_{\tau} y$ . Note that the later denotes a maximal play in  $B$ .

I.  $\text{command}_{\sigma} x = \text{inl}(r_a, r_b)$ .

I.1.  $\text{command}_{\tau} y = \text{inl}(r_b, r_c)$ .

Then we have  $\text{command}_{\sigma;\tau}(x, y, l) = \text{inl}(r_a, r_c)$ . Let  $(c_a, r_c) : (A \multimap C)_{\mathbb{R}}$  with  $\text{inl}(r_a, r_c) \ll (c_a, r_c)$  i.e.  $r_a \ll^A c_a$ . Since  $\sigma$  plays in  $A$   $\text{maxABeg}_{\sigma} x$  has the shape  $\text{fork} r_a f$ . After receiving  $(c_a, r_c)$  the strategy  $\sigma;\tau$  goes into a state  $(x', y, l')$  where  $x' = \text{next}_{\sigma} x c_a$ . Since  $\text{maxvalidABeg}(f c_a)$   $x'$  the beginning  $\text{maxABeg}_{\sigma;\tau}(x', y, l')$  is already defined and we get a function

$$h : (\text{carc} : (A \multimap C)_{\mathbb{R}}, \text{inl}(r_a, r_c) \ll \text{carc}) \rightarrow \text{Beginning } A \multimap C.$$

$\text{fork}(\text{inl}(r_a, r_c)) h$  gives the desired beginning.

I.2.  $\text{command}_{\tau} y = \text{inr}(c_b, c_c)$ . Then we have  $l : \text{False}$ .

II.  $\text{command}_{\sigma} x = \text{inr}(c_a, c_b)$ .

II.1.  $\text{command}_{\tau} y = \text{inl}(r_b, r_c)$ .

II.1.a)  $c_b \ll r_b$ . Apply the I.H. on  $(x', y, l)$ .

II.1.b)  $c_b \gg r_b$ . Apply the I.H. on  $(x, y', l)$ .

II.2.  $\text{command}_{\tau} y = \text{inr}(c_b, c_c)$ . Then  $\text{maxABeg}_{\sigma;\tau}(x, y, l) := \text{stop}$ . □

## 7 Behaviour

In order to get a category we identify strategies which have the same behaviour. The states  $x : X_\sigma$  of the strategies  $\sigma$  for a game  $A$  build a transition system. The transitions  $x \rightarrow x'$  are given by responses  $r : A_R$  with  $\text{command}_\sigma x \llcorner r$  and  $x' = \text{next}_\sigma r \_$ . This transition system is obviously image finite. This means we can define bisimulation by induction on the natural numbers:

**Definition 11.** Let  $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma)$  and  $\tau = (X_\tau, \text{command}_\tau, \text{next}_\tau)$  strategies on  $A$ .  $x : X_\sigma, y : X_\tau, n : \text{Nat}$  are  $n$ -bisimilar ( $x \sim_n y$ ) iff  $n = 0$  or

$$\text{command}_\sigma x = \text{command}_\tau y$$

and for  $r : A_R$  and  $\text{rfit} : \text{command}_\sigma x \llcorner_R r$

$$\text{next}_\sigma x r \text{ rfit} \sim_{n-1} \text{next}_\tau y r \text{ rfit}.$$

$x : X_\sigma, y : X_\tau$  are bisimilar iff they are  $n$ -bisimilar for all  $n : \text{Nat}$ .

**Definition 12.** Strategies  $\sigma = (X_\sigma, \text{command}_\sigma, \text{next}_\sigma)$  and  $\tau = (X_\tau, \text{command}_\tau, \text{next}_\tau)$  on  $A$  are behavioural equivalent ( $\sigma \approx \tau$ ) iff there are  $f : X_\sigma \rightarrow X_\tau$  and  $g : X_\tau \rightarrow X_\sigma$  such that

$$f x \sim x \quad \text{and} \quad g y \sim y$$

for all  $x : X_\sigma, y : X_\tau$ .

## 8 The Category of Games with Sound and Fair Strategies

**Lemma 2.** Let  $\sigma : A \multimap B$  be fair and sound. Then we have

$$\text{id}_A; \sigma \approx \sigma \approx \sigma; \text{id}_B$$

where  $\text{id}_A := \text{CopyCat}_A$  and  $\text{id}_B := \text{CopyCat}_B$ .

*Proof.* By natural induction follows

$$x \llcorner r_b \sim x' \quad x \underset{c}{=} c_b \sim x \quad x \underset{r}{=} r_b \sim x \quad x \gg r_b \sim x$$

where in the first case  $x' = \text{next}_\sigma x (c_a, r_b) \_$  and  $(c_a, c_b) = \text{command}_\sigma x$ .  $\text{id}_A; \sigma \approx \sigma$  follows analogously.  $\square$

**Lemma 3.** Let  $\sigma : A \multimap B, \tau : B \multimap C, \mu : C \multimap D$  be fair and sound. Then we have

$$(\sigma; \tau); \mu \approx \sigma; (\tau; \mu).$$

*Proof.* We write  $v \rightsquigarrow w$  for onestep  $v \rightsquigarrow$  onestep  $w$ . By definition of composition and reflexivity of  $\sim$  we have

$$\begin{array}{ll}
(x \leftarrow y) \underset{c}{=} z \sim (x^i \underset{c}{=} y^i) \underset{c}{=} z & (x \leftarrow y) \leftarrow z \sim (x^i \underset{c}{=} y^i) \leftarrow z \\
(x \leftarrow y) \succ z \sim (x^i \underset{c}{=} y^i) \succ z & (x \leftarrow y) \underset{r}{=} z \sim (x^{i+1} \underset{r}{=} y^i) \underset{r}{=} z \\
(x \succ y) \underset{c}{=} z \sim (x^i \underset{c}{=} y^{i+1}) \underset{c}{=} z & (x \succ y) \leftarrow z \sim (x^i \underset{c}{=} y^{i+1}) \leftarrow z \\
(x \succ y) \succ z \sim (x^i \underset{c}{=} y^{i+1}) \succ z & (x \succ y) \underset{r}{=} z \sim (x^i \underset{r}{=} y^i) \underset{r}{=} z \\
x \underset{c}{=} (y \leftarrow z) \sim x \underset{c}{=} (y^i \underset{c}{=} z^i) & x \leftarrow (y \leftarrow z) \sim x \leftarrow (y^{i+1} \underset{r}{=} z^i) \\
x \succ (y \leftarrow z) \sim x \succ (y^{i+1} \underset{r}{=} z^i) & x \underset{r}{=} (y \leftarrow z) \sim x \underset{r}{=} (y^{i+1} \underset{r}{=} z^i) \\
x \underset{c}{=} (y \succ z) \sim x \underset{c}{=} (y^i \underset{c}{=} z^{i+1}) & x \leftarrow (y \succ z) \sim x \leftarrow (y^i \underset{r}{=} z^i) \\
x \succ (y \succ z) \sim x \succ (y^i \underset{r}{=} z^i) & x \underset{r}{=} (y \succ z) \sim x \underset{r}{=} (y^i \underset{r}{=} z^i)
\end{array}$$

where  $x \leftarrow y \rightsquigarrow x^i \underset{c}{=} y^i$  in the first three cases,  $x \leftarrow y \rightsquigarrow x^{i+1} \underset{c}{=} y^i$  in the fourth etc.

We prove by natural induction

$$(x \underset{c}{=} y) \underset{c}{=} z \sim x \underset{c}{=} (y \underset{c}{=} z) \quad (1)$$

$$(x \underset{c}{=} y) \leftarrow z \sim x \underset{c}{=} (y \leftarrow z) \text{ for } \text{command}_{\tau;\mu}(y \leftarrow z) = \text{inr } \_ \quad (2)$$

$$(x \underset{c}{=} y) \leftarrow z \sim x \leftarrow (y \leftarrow z) \text{ for } \text{command}_{\tau;\mu}(y \leftarrow z) = \text{inl } \_ \quad (3)$$

$$(x \underset{c}{=} y) \succ z \sim x \underset{c}{=} (y \succ z) \text{ for } \text{command}_{\tau;\mu}(y \succ z) = \text{inr } \_ \quad (4)$$

$$(x \underset{c}{=} y) \succ z \sim x \leftarrow (y \succ z) \text{ for } \text{command}_{\tau;\mu}(y \succ z) = \text{inl } \_ \quad (5)$$

$$x \underset{r}{=} (y \underset{r}{=} z) \sim (x \underset{r}{=} y) \underset{r}{=} z \quad (6)$$

$$x \succ (y \underset{r}{=} z) \sim (x \succ y) \underset{r}{=} z \text{ for } \text{command}_{\sigma;\tau}(x \succ y) = \text{inl } \_ \quad (7)$$

$$x \succ (y \underset{r}{=} z) \sim (x \succ y) \succ z \text{ for } \text{command}_{\sigma;\tau}(x \succ y) = \text{inr } \_ \quad (8)$$

$$x \leftarrow (y \underset{r}{=} z) \sim (x \leftarrow y) \underset{r}{=} z \text{ for } \text{command}_{\sigma;\tau}(x \leftarrow y) = \text{inl } \_ \quad (9)$$

$$x \leftarrow (y \underset{r}{=} z) \sim (x \leftarrow y) \succ z \text{ for } \text{command}_{\sigma;\tau}(x \leftarrow y) = \text{inr } \_ \quad (10)$$

Ad 1. We have

$$\text{inr}(c_a, c_d) := \text{command}_{(\sigma;\tau);\mu}(x \underset{c}{=} y) \underset{c}{=} z = \text{command}_{\sigma;(\tau;\mu)}x \underset{c}{=} (y \underset{c}{=} z).$$

Let  $\text{inr}(c_a, c_d) \leftarrow (c_a, r_d)$  and  $z' := \text{next}_\mu z(c_c, r_d) \_$

I.:  $\text{command}_\mu z' = \text{inr}(c_c, c'_d)$ .

$$\begin{aligned}
\text{next}(x \underset{c}{=} y) \underset{c}{=} z(c_a, r_d) \_ &= (x \underset{c}{=} y) \underset{c}{=} z' \sim_n x \underset{c}{=} (y \underset{c}{=} z') \quad \text{I.H.} \\
&= \text{next } x \underset{c}{=} (y \underset{c}{=} z')(c_a, r_d) \_
\end{aligned}$$

II.:  $\text{command}_\mu z' = \text{inl}(r_c, r_d)$ .

II.a):  $\text{command}_{\tau;\mu}(y \ll z') = \text{inr } \_$

$$\begin{aligned} \text{next}(x \underset{c}{=} y) \underset{c}{=} z(c_a, r_d) \_ &= (x \underset{c}{=} y) \ll z' \sim_n x \underset{c}{=} (y \ll z') \quad \text{I.H.} \\ &= \text{next } x \underset{c}{=} (y \underset{c}{=} z')(c_a, r_d) \_ \end{aligned}$$

II.b):  $\text{command}_{\tau;\mu}(y \ll z') = \text{inl } \_$

$$\begin{aligned} \text{next}(x \underset{c}{=} y) \underset{c}{=} z(c_a, r_d) \_ &= (x \underset{c}{=} y) \ll z' \sim_n x \ll (y \ll z') \quad \text{I.H.} \\ &= \text{next } x \underset{c}{=} (y \underset{c}{=} z')(c_a, r_d) \_ \end{aligned}$$

Ad 6. Analog to 1.

Ad 2.

$$\begin{aligned} (x \underset{c}{=} y) \ll z \rightsquigarrow (x \underset{c}{=} y^i) \underset{c}{=} z^i \sim_{n+1} x \underset{c}{=} (y^i \underset{c}{=} z^i) \quad \text{by 1} \\ \rightsquigarrow x \underset{c}{=} (y \ll z) \end{aligned}$$

where  $y \ll z \rightsquigarrow y^i \underset{c}{=} z^i$ .

Ad 7. Analog to 2.

Ad 3. Let  $\text{inr}(c_a, c_c) := \text{command}(x \underset{c}{=} y)$  and  $z' := \text{next } z(c_c, r_d) \_$ .

I.:  $\text{command } z' = \text{inl } \_$

$$\begin{aligned} (x \underset{c}{=} y) \gg z \rightsquigarrow (x \underset{c}{=} y) \ll z' \sim_{n+1} x \underset{c}{=} (y \ll z') \quad \text{by 2} \\ \rightsquigarrow x \underset{c}{=} (y \gg z). \end{aligned}$$

II.:  $\text{command } z' = \text{inr } \_$

$$\begin{aligned} (x \underset{c}{=} y) \gg z \rightsquigarrow (x \underset{c}{=} y) \underset{c}{=} z' \sim_{n+1} x \underset{c}{=} (y \underset{c}{=} z') \quad \text{by 1} \\ \rightsquigarrow x \underset{c}{=} (y \underset{c}{=} z). \end{aligned}$$

Ad 8. Analog to 3.

Ad 4.

$$\begin{aligned} (x \underset{c}{=} y) \ll z \rightsquigarrow (x \underset{c}{=} y^i) \underset{c}{=} z^{i+1} \sim_{n+1} x \underset{c}{=} (y^i \underset{c}{=} z^{i+1}) \quad \text{by 1} \\ \rightsquigarrow x \underset{c}{=} (y \gg z). \end{aligned}$$

Ad 9. Analog to 4.

Ad 5. Let  $\text{inr}(c_a, c_c) := \text{command}(x \underset{c}{=} y)$  and  $z' := \text{next } z(c_c, r_d) \_$ .

$$\begin{aligned} (x \underset{c}{=} y) \gg z \rightsquigarrow (x \underset{c}{=} y) \ll z' \sim_{n+1} x \ll (y \ll z') \quad \text{by 3} \\ \rightsquigarrow x \ll (y \gg z). \end{aligned}$$

Ad 10. Analogue to 5. □

**Theorem 1.** *Games A together with sound & fair strategies on  $A \multimap B$  and the composition defined above form a category if we identify strategies by the notion of behavioural equivalence above.*

## 9 Conclusion

We introduced a generalisation of Hancock/Setzer's notion of interfaces and investigated its relationship to game semantic. We pointed out how to understand this generalised notion as games. We are going to explain in what sense the resulting notion is indeed a generalisation in a forthcoming paper. We proposed a simple notion of refinement and showed that the refinement relation is the order relation of a complete lattice. We introduced operations negation, tensor and lolipop on these games. We developed the notion of sound and fair strategy for games  $A \multimap B$ . By copying moves from one side to the other we defined a composition for this strategies. We showed that the composed strategy is again sound and fair. By identifying strategies by a notion of behavioural equivalence we received a category with games as objects where the morphisms are sound and fair strategies on  $A \multimap B$ . We conjecture that this category can be modified to yield a model of classical linear logic in intensional type theory, i.e. a linear category with dualising object.

## Acknowledgement

The author thanks Anton Setzer, Peter Hancock and Pierre Hyvernat for fruitful discussions, two anonymous referees for valuable comments and Ken Johnson for proof-reading the article.

## References

1. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of Containers. In Andrew Gordon, editor, *Proceedings of FOSSACS 2003*, volume 2620 of *Lecture Notes in Computer Science*, pages 23–38. Springer-Verlag, 2003.
2. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, TLCA 2003*, volume 2701 of *Lecture notes in Computer Science*, pages 16–30. Springer, 2003.
3. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride.  $\partial$  for data. *Fundamenta Informatica*, 65(1 - 2), 2005.
4. Michael Gordon Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.
5. Samson Abramsky. Semantics of Interaction: an introduction to Game Semantics. In P. Dybjer and A. Pitts, editors, *Proceedings of the 1996 CLiCS Summer School*, pages 1–31. Cambridge University Press, 1997.
6. Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdziński, and Freddy Y.C. Mang. Interface Compatibility Checking for Software Modules. In *CAV 2002*, volume 2404 of *Lecture notes in Computer Science*, pages 428–441. Springer, 2002.
7. Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. In *Proceedings of the second workshop of Formal Topology*, Annals of Pure and Applied Logic, 2004. to appear.

8. Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Proc. of 14th Ann. Conf. of EACSL, CSL'00, Fischbau, Germany, 21–26 Aug 2000*, volume 1862, pages 317–331, Berlin, 2000. Springer-Verlag.
9. Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, 2000. Electronic proceedings.
10. Peter Hancock and Anton Setzer. Guarded induction and weakly final coalgebras in dependent type theory (extended version). In T. Altenkirch, M. Hofmann, and J. Hughes, editors, *Dependently typed programming. Dagstuhl Seminar Proceedings 04381, Dagstuhl, Germany, 2005*.
11. Peter Hancock and Anton Setzer. Interactive programs and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From sets and types to topology and analysis: Towards practicable foundations for constructive mathematics*, Oxford Logic Guides, pages 115 – 136. Oxford University Press, 2005.
12. G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549 – 554, 1997.
13. Martin Hyland. Game Semantics. In P. Dybjer and A. Pitts, editors, *Proceedings of the 1996 CLiCS Summer School*, pages 131–184. Cambridge University Press, 1997.
14. Pierre Hyvernat. Predicate transformers and linear logic: yet another denotational model. In *18th International Workshop CSL 2004*, volume 3210 of *Lecture notes in Computer Science*. Springer, 2004.
15. Pierre Hyvernat. Synchronous Games, Simulations and  $\lambda$ -calculus. In Dan R. Ghica and Guy McCusker, editors, *Games for Logic and Programming Languages, GaLoP (ETAPS 2005)*, pages 1–15, April 2005.
16. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.
17. Markus Michelbrink. A generalisation of Hancock-Setzer Interfaces. Draft. Available via <http://www.cs.swan.ac.uk/~csmichel/>, 2005.
18. Markus Michelbrink. Interfaces as functors, programs as coalgebras - a final coalgebra theorem in intensional type theory. Submitted. Available via <http://www.cs.swan.ac.uk/~csmichel/>, 2005.
19. Markus Michelbrink and Anton Setzer. State-dependent IO-monads in type theory. In Lars Birkedal, editor, *Proceedings of the 10th Conference on Category Theory in Computer Science (CTCS 2004)*, volume 122 of *Electronic Notes in Theoretical Computer Science*, pages 127–146. Elsevier, 2005.
20. Giovanni Sambin. The Basic Picture, a structure for topology (the Basic Picture, I), 2001.

# $\lambda Z$ : Zermelo’s Set Theory as a PTS with 4 Sorts

Alexandre Miquel

PPS & Université Paris 7,  
175 rue du Chevaleret, 75013 Paris

**Abstract.** We introduce a pure type system (PTS)  $\lambda Z$  with four sorts and show that this PTS captures the proof-theoretic strength of Zermelo’s set theory. For that, we show that the embedding of the language of set theory into  $\lambda Z$  via the ‘sets as pointed graphs’ translation makes  $\lambda Z$  a conservative extension of  $IZ + AFA + TC$  (intuitionistic Zermelo’s set theory plus Aczel’s antifoundation axiom plus the axiom of transitive closure)—a theory which is equiconsistent to Zermelo’s. The proof of conservativity is achieved by defining a retraction from  $\lambda Z$  to a (skolemised version of) Zermelo’s set theory and by showing that both transformations commute via the axioms AFA and TC.

## 1 Introduction

Modern proof assistants based on the Curry Howard correspondence—such as Agda, Coq, Nuprl or Plastic—basically implement a well-known pure type system [7, 3] (PTS) enriched with many extensions such as inductive data-types and recursive definitions of functions. Traditionally, the proof-theoretic strength of the implemented formalisms is estimated via the sets-in-types and types-in-sets encodings [13, 2], that respectively give a lower and an upper bound of the proof-theoretic strength of the system, expressed as a variant of set theory.

Surprisingly, very little is known about the proof-theoretic strength of the underlying PTSs themselves. The main reason is that the framework of PTSs lacks the inductive data-types that are crucial in the definition of the traditional sets-in-types encoding based on Aczel’s  $W$ -trees. Another reason is that there is currently no simple set-theoretic interpretation of type-theoretic universes that does not rely on the existence of large cardinals—an assumption which is definitely too strong to give a reasonable upper bound of a PTS.

The aim of this paper is to initiate a more systematic study of the proof-theoretic strength of the subsystems of the Calculus of Constructions with universes ( $CC_\omega$ ) following the correspondence with extensions of Zermelo’s set theory that was outlined in the author’s thesis [11]. In this direction, we present a first result by extracting a sub-PTS of the Calculus of Constructions with universes—the system  $\lambda Z$  presented in section 2—that captures the proof-theoretic strength of Zermelo’s set theory (without the Foundation Axiom). Moreover, we show that through the sets-as-pointed-graphs encoding (which is recalled in section 4) the PTS  $\lambda Z$  appears to be a conservative extension of a



very natural extension of Intuitionistic Zermelo's set theory, namely, the system  $IZ + AFA + TC$  whose classical version as been already considered in [5], and which is clearly equiconsistent to  $Z$ .

Finally, let us mention that the crucial ingredient of the equiconsistency proof presented in this paper does not come from the type-theoretic side, but from the set-theoretic side. As we shall see in section 3, introducing an explicitly Skolemised version of Zermelo's set theory reveals some unexpected closure properties of this system that are fruitfully exploited in the definition of the types-in-sets interpretation presented in section 5.

## 2 The PTS λZ

In this section, we assume the reader has some familiarity with the theory of PTS (see [7, 3]).

### 2.1 The PTS Presentation

**Definition 1** ( $\lambda Z$ ). —  $\lambda Z$  is the PTS whose set of sorts  $\mathcal{S}$ , whose set of axioms  $\mathcal{A} \subset \mathcal{S}^2$  and whose set of rules  $\mathcal{R} \subset \mathcal{S}^3$  are given by

$$\begin{aligned} \mathcal{S} &= \{*; \square_1; \square_2; \square_3\}, \\ \mathcal{A} &= \{(* : \square_1); (\square_1 : \square_2); (\square_2 : \square_3)\}, \\ \mathcal{R} &= \{(*, *, *); (\square_i, *, *) \mid i \in \{1, 2, 3\}\} \cup \{(\square_i, \square_j, \square_{\max(i,j)}) \mid i, j \in \{1, 2\}\}. \end{aligned}$$

By construction, the PTS  $\lambda Z$  is a sub-system of the calculus of constructions with universes ( $CC_\omega$ ), and actually, a subsystem of system  $F\omega$  with universes ( $F\omega^2$ , the non-dependent fragment of  $CC_\omega$ ) which is the PTS defined by:

$$\begin{aligned} \mathcal{S}_{F\omega^2} &= \{*; \square_i \mid i \geq 1\}, \\ \mathcal{A}_{F\omega^2} &= \{(* : \square_0); (\square_i : \square_{i+1}) \mid i \geq 1\}, \\ \mathcal{R}_{F\omega^2} &= \{(*, *, *); (\square_i, *, *); (\square_i, \square_j, \square_{\max(i,j)}) \mid i, j \geq 1\}. \end{aligned}$$

Moreover, if we write  $F\omega.n$  (for  $n \geq 1$ ) the PTS obtained by restricting  $F\omega^2$  to the set of sorts  $\{*; \square_i \mid 1 \leq i \leq n\}$ , then we have the inclusions:

$$F\omega.2 \subset \lambda Z \subset F\omega.3 \subset \dots \subset F\omega^2 \subset CC_\omega$$

Intuitively, the PTS  $\lambda Z$  extends  $F\omega.2$  with a sort  $\square_3$ , an axiom  $\square_2 : \square_3$  and a unique rule  $(\square_3, *, *)$ , whereas  $F\omega.3$  completes the extension by adding all the 'missing rules'  $(\square_3, \square_i, \square_3)$  and  $(\square_i, \square_3, \square_3)$  for  $i \in \{1; 2; 3\}$ .

As for any PTS,  $\lambda Z$  enjoys many good properties, such as substitutivity and subject-reduction [7] as well as the property of uniqueness of types up to  $\beta$ -conversion (since  $\lambda Z$  is a functional PTS).

From the inclusions  $\lambda Z \subset F\omega^2 \subset CC_\omega$  we immediately get [9, 10]:

**Fact 1 (Strong Normalisation).** — *All the well-typed term of  $\lambda Z$  are strongly normalisable terms.*

It is important to notice that this result will not be used in the following, for that the conservativity result we will present purely relies on syntactic codings (that involve straightforward conversion steps on the type-theoretic side). On the other hand, using the normalisation result above—which seems to require much more proof-theoretic strength than the consistency of Zermelo’s<sup>1</sup>—would dramatically weaken the interest of our relative consistency proof.

## 2.2 Stratified Presentation of $F\omega^2$

As for the systems of Barendregt’s cube, the PTS  $F\omega^2$  (and its subsystems) can be given a stratified presentation which syntactically distinguishes the terms whose type has type  $\square_i$ —that represent mathematical objects—from the terms whose type has type  $*$ —that represent mathematical proofs.

Formally, we say that in a given context  $\Gamma$ , a term  $M$  of type  $T$  is an *object term* if  $\Gamma \vdash T : \square_i$  for some  $i \geq 1$ , and a *proof term* if  $\Gamma \vdash T : *$ . Notice that *propositions*—that is, terms of type  $*$ —are a special case of object terms. In the rest of this presentation, we use capital letters  $M, N, T, U, A, B$ , etc. to denote object terms (and more specifically:  $T, U$  for types and  $A, B$  for propositions) whereas lowercase letters  $t, u$ , etc. are reserved for proof terms.

Dependent and non-dependent products are stratified according to their formation rule as follows:

- Non-dependent products formed according to the rule  $(*, *, *)$ , which express logical implication, are written  $A \Rightarrow B$ . Notice that non-dependent products are the only products that can be formed by this rule, since  $\lambda Z$  is a non-dependent logical PTS (following the terminology of [4]).
- Dependent products formed according to the rule  $(\square_i, *, *)$ , which express universal quantification, are written  $\forall x : T. A$ .
- Dependent products formed according to the rule  $(\square_i, \square_i, \square_i)$ , which express dependent function spaces, are still written  $\Pi x : T. U$  (or simply  $T \rightarrow U$  in the non-dependent case, when  $x \notin FV(U)$ ).

The stratified presentation of system  $F\omega^2$  is given in table 1, and the corresponding (stratified) typing rules are recalled in table 2.

**Proposition 1 (Stratification Equivalence).** — *The well-typed terms of system  $F\omega^2$  are exactly the object terms and proof terms that can be expressed in the syntax given in table 1 and type-checked using the rules of table 2.*

## 2.3 The Stratified Presentation of $\lambda Z$

In the stratified setting, system  $\lambda Z$  naturally appears as the subsystem of  $F\omega^2$  which is obtained:

<sup>1</sup> We conjecture that the (*strong*) *normalisation* of  $\lambda Z$  has the same proof-theoretic strength as (the consistency of)  $\text{IZ}$  plus one Zermelo-universe. This has to be compared to the formalisms  $\text{CC}$  and  $F\omega$ , whose strong normalisation properties have exactly the same proof-theoretic strength as higher-order arithmetic ( $\text{HA}\omega$ ) but whose consistency can be proved within Heyting arithmetic ( $\text{HA}$ ).

**Table 1.** The stratified presentation of  $F\omega^2$

<b>Object terms</b>	$M, N, T, U, A, B$	$::=$	$x \mid \lambda x:T.M \mid MN$ $\mid \Pi x:T.U \mid * \mid \Box_i \ (i \geq 1)$ $\mid A \Rightarrow B \mid \forall x:T.A$
<b>Proof terms</b>	$t, u$	$::=$	$\xi$ $\mid \lambda \xi^A.t \mid tu$ $\mid \lambda x:T.t \mid tM$

**Table 2.** Typing rules of  $F\omega^2$

<u>Context formation</u>	
$\overline{\Box} \vdash$	$\frac{\Gamma \vdash T : \Box_i}{\Gamma, x : T \vdash}$ $\frac{\Gamma \vdash A : *}{\Gamma, \xi : A \vdash}$
<u>Object terms</u>	
$\frac{\Gamma \vdash}{\Gamma \vdash x : T} \ (x:T) \in \Gamma$	$\frac{\Gamma \vdash}{\Gamma \vdash * : \Box_1}$
$\frac{\Gamma \vdash}{\Gamma \vdash \Box_i : \Box_{i+1}}$	$\frac{\Gamma \vdash}{\Gamma \vdash \Box_i : \Box_{i+1}}$
$\frac{\Gamma \vdash \Pi x:T.U : \Box_i \quad \Gamma, x:T \vdash M : U}{\Gamma \vdash \lambda x:T.M : \Pi x:T.U}$	$\frac{\Gamma \vdash M : \Pi x:T.U \quad \Gamma \vdash N : T}{\Gamma \vdash MN : U\{x := N\}}$
$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : *}{\Gamma \vdash A \Rightarrow B : *}$	$\frac{\Gamma, x:T \vdash A : *}{\Gamma \vdash \forall x:T.A : *}$
$\frac{\Gamma \vdash T : \Box_i \quad \Gamma, x:T \vdash U : \Box_j}{\Gamma \vdash \Pi x:T.U : \Box_{\max(i,j)}}$	$\frac{\Gamma \vdash M : T \quad \Gamma \vdash T' : \Box_i}{\Gamma \vdash M : T'} \ T' =_{\beta} T$
<u>Proof terms</u>	
$\frac{\Gamma \vdash}{\Gamma \vdash \xi : A} \ (\xi:T) \in \Gamma$	$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A' : *}{\Gamma \vdash t : A'} \ A' =_{\beta} A$
$\frac{\Gamma, \xi : A \vdash t : B}{\Gamma \vdash \lambda \xi : A.t : A \Rightarrow B}$	$\frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$
$\frac{\Gamma, x : T \vdash t : A}{\Gamma \vdash \lambda x : T.t : \forall x : T.A}$	$\frac{\Gamma \vdash t : \forall x : T.A \quad \Gamma \vdash N : T}{\Gamma \vdash tN : A\{x := N\}}$

- By restricting the set of sorts to the initial segment  $\mathcal{S} = \{*, \square_1, \square_2, \square_3\}$  of  $\mathcal{S}_{F,\omega^2}$  and the set  $\mathcal{A}$  of axioms accordingly.
- By restricting the formation rule of dependent function spaces to the rules of the form  $(\square_i, \square_j, \square_{\max(i,j)})$  for  $i, j \in \{1; 2\}$ .

On the other hand, system  $\lambda Z$  does not further restrict the rules  $(\square_i, *, *)$  that are responsible for the formation of universal quantification  $\forall x : T . A$ , and that can be used at any index  $i \in \{1; 2; 3\}$ .

To understand the structure of  $\lambda Z$ , let us explain the meaning of each universe  $\square_i$  (for  $i \in \{1; 2; 3\}$ ) and of each formation rule  $(\square_i, *, *)$  in terms of the notions they will correspond to via our translation to set theory:

1. The first universe  $\square_1$ —that contains no provably infinite data-type<sup>2</sup>—has to be thought as the universe of *finite data-types*. Technically, the presence of a first universe below the universe  $\square_2$  of sets (see below) is needed to justify the existence of a provably infinite data-type in  $\square_2$ , and plays the very same role as the axiom of infinity in set theory. In particular, universal quantifications  $\forall x : T . A(x)$  formed by the rule  $(\square_1, *, *)$  roughly correspond to finite quantifications  $\forall x < t A(x)$  (where  $t \in \omega$ ) in set theory.
2. The universe  $\square_2$  has to be thought as the universe of sets, or, more precisely, as the universe of the *carriers* of the pointed graphs that we will use to represent sets. Thus, universal quantifications  $\forall x : T . A(x)$  formed by the rule  $(\square_2, *, *)$  correspond to bounded quantifications  $\forall x \in t A(x)$  in set theory.
3. The sort  $\square_3$  is a top sort whose only inhabitant is the universe  $\square_2$ —which is due to the absence of formation rules of the form  $(s_1, s_2, \square_3)$ . Technically this sort is needed to type-check the construction  $\forall x : \square_2 . A(x)$ —the only form of universal quantification induced by the rule  $(\square_3, *, *)$ —that corresponds to the unbounded quantification  $\forall x A(x)$  in set theory.

The aim of this paper is to formalise the correspondence depicted above to turn it into a result of proof-theoretic equivalence.

### 3 Zermelo’s Set Theory

#### 3.1 The Core Language

Zermelo’s set theory (Z) is the classical first-order theory whose language is built from the two binary relations  $x = y$  (*equality*) and  $x \in y$  (*membership*)

**Formulæ**      $\phi, \psi ::= \top \mid \perp \mid x = y \mid x \in y$   
                    $\mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \forall x \psi \mid \exists x \psi$

---

<sup>2</sup> This will be a consequence of the soundness of the translation  $(\_)^\dagger$  defined in section 5.

and whose axioms are given in table 3, using the following shorthands:

$$\begin{aligned}
 \neg\phi &\equiv \phi \Rightarrow \perp & \phi \Leftrightarrow \psi &\equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\
 x \notin y &\equiv \neg(x \in y) & x \subset y &\equiv \forall z (z \in x \Rightarrow z \in y) \\
 \text{Zero}(x) &\equiv \forall z (z \notin x) & \text{Succ}(x, y) &\equiv \forall z [z \in y \Leftrightarrow z \in x \vee z = x] \\
 \text{Nat}(n) &\equiv \forall a [\forall x (\text{Zero}(x) \Rightarrow x \in a) \wedge \\
 &\quad \forall x \forall y (x \in a \wedge \text{Succ}(x, y) \Rightarrow y \in a) \Rightarrow n \in a]
 \end{aligned}$$

(Notice that in this presentation of Z, there is no Axiom of Foundation.)

**Table 3.** Axioms of Zermelo's set theory

<u>Equality axioms</u>	
(REFLEXIVITY)	$\forall x (x = x)$
(SYMMETRY)	$\forall x \forall y (x = y \Rightarrow y = x)$
(TRANSITIVITY)	$\forall x \forall y \forall z (x = y \wedge y = z \Rightarrow x = z)$
(MEM-COMPAT-L)	$\forall x \forall y \forall z (x = y \wedge y \in z \Rightarrow x \in z)$
(MEM-COMPAT-R)	$\forall x \forall y \forall z (x \in y \wedge y = z \Rightarrow x \in z)$
<u>Zermelo's axioms</u>	
(EXTENSIONALITY)	$\forall a \forall b [\forall x (x \in a \Leftrightarrow x \in b) \Rightarrow a = b]$
(PAIRING)	$\forall a_1 \forall a_2 \exists b \forall x [x \in b \Leftrightarrow x = a_1 \vee x = a_2]$
(COMPREHENSION)	$\forall x_1 \dots \forall x_n \forall a \exists b \forall x [x \in b \Leftrightarrow x \in a \wedge \phi]$ for any formula $\phi$ such that $FV(\phi) \subset \{x_1; \dots; x_n; x\}$ .
(POWERSSET)	$\forall a \exists b \forall x [x \in b \Leftrightarrow x \subset a]$
(UNION)	$\forall a \exists b \forall x [x \in b \Leftrightarrow \exists y (y \in a \wedge x \in y)]$
(INFINITY)	$\exists a \forall x [x \in a \Leftrightarrow \text{Nat}(x)]$

Intuitionistic Zermelo's set theory (IZ) is the theory based on the same language and axioms as Z, but in which reasoning is done in intuitionistic logic. As shown by [6], there is a double negation translation which maps (classically) provable formulæ of Z to (intuitionistically) provable formulæ of IZ, so that both theories IZ and Z are actually equiconsistent.

In what follows, we will mainly work in IZ.

### 3.2 Skolemising Z

The main drawback of the traditional presentation of set theory is the lack of notations to express objects (i.e. sets). To define the 'retraction' of section 5,

we first need to enrich—in a conservative way—the term algebra of set theory (that only contains variables) with notations to express the unordered pairs, the powersets, the unions, the set of natural numbers and all the sets defined by using the comprehension scheme.

Formally, we introduce a system called  $Z^{\text{sk}}$ , whose terms and formulæ are mutually defined by:

<b>Terms</b>	$t, u ::= x \mid \omega \mid \{t_1; t_2\} \mid \mathfrak{P}(t) \mid \bigcup t \mid \{x \in t \mid \phi\}$
<b>Formulæ</b>	$\phi, \psi ::= t = u \mid t \in u \mid \top \mid \perp$ $\mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \forall x \phi \mid \exists x \phi$

(Free and bound occurrences of variables are defined as expected, keeping in mind that the construction  $\{x \in t \mid \phi\}$  binds all the free occurrences of the variable  $x$  in  $\phi$ , but none of the free occurrences of  $x$  in  $t$ . The notions of substitutions  $t\{x := u\}$  and  $A\{x := u\}$  are defined accordingly.)

Although  $Z^{\text{sk}}$  is not based on a first-order language, the underlying notions of sequent, inference rule and derivation are defined as in first-order theories. The axioms of  $Z^{\text{sk}}$  are the same as in  $Z$ , except that the existential axioms of Zermelo’s system (table 3) are replaced by their Skolemized forms (table 4).

The intuitionistic fragment of  $Z^{\text{sk}}$  is written  $IZ^{\text{sk}}$ .

**Table 4.** Skolemised axioms of  $Z^{\text{sk}}$

(PAIRING <sup>sk</sup> )	$\forall a_1 \forall a_2 \forall x [x \in \{a_1; a_2\} \Leftrightarrow x = a_1 \vee x = a_2]$
(COMPREHENSION <sup>sk</sup> )	$\forall x_1 \dots \forall x_n \forall a \forall x [x \in \{z \in a \mid \phi\} \Leftrightarrow x \in a \wedge \phi\{z := x\}]$ for any formula $\phi$ such that $FV(\phi) \subset \{x_1; \dots; x_n; z\}$ .
(POWERSET <sup>sk</sup> )	$\forall a \forall x [x \in \mathfrak{P}(a) \Leftrightarrow x \subset a]$
(UNION <sup>sk</sup> )	$\forall a \forall x [x \in \bigcup a \Leftrightarrow \exists y (y \in a \wedge x \in y)]$
(INFINITY <sup>sk</sup> )	$\forall x [x \in \omega \Leftrightarrow \text{Nat}(x)]$

The theory  $(I)Z^{\text{sk}}$  is clearly an extension of  $(I)Z$ ,<sup>3</sup> in the sense that for any formula  $\phi$  of set theory,  $(I)Z \vdash \phi$  entails  $(I)Z^{\text{sk}} \vdash \phi$ .

From the axioms of table 4, we easily check that the function symbols  $\{-; \cdot\}$ ,  $\mathfrak{P}(\_)$  and  $\bigcup \_$  are compatible with equality (in  $IZ^{\text{sk}}$ ) as well as the construction  $\{x \in t \mid \phi\}$  in the sense that:

$$\begin{array}{l} IZ^{\text{sk}} \vdash \forall x_1 \dots \forall x_n \forall a \forall a' [a = a' \Rightarrow \{x \in a \mid \phi\} = \{x \in a' \mid \phi\}] \\ IZ^{\text{sk}} \vdash \forall x_1 \dots \forall x_n \forall a [\forall x (\phi \Leftrightarrow \phi') \Rightarrow \{x \in a \mid \phi\} = \{x \in a \mid \phi'\}] \end{array}$$

<sup>3</sup> The shorthand  $(I)Z$  reads: “ $Z$  (resp.  $IZ$ )”. And similarly for  $(I)Z^{\text{sk}}$ .

(for all formulæ  $\phi, \phi'$  of  $Z^{\text{sk}}$  such that  $FV(\phi) \cup FV(\phi') \subset \{x_1; \dots; x_n; x\}$ ), from which we deduce that Leibniz principle holds, both for terms and formulæ:

**Proposition 2 (Leibniz Principle).** — *For any term  $t$  and for any formula  $\phi$  of the language of  $Z^{\text{sk}}$ :*

$$\begin{aligned} \text{IZ}^{\text{sk}} \quad \vdash \quad x_1 = x_2 &\Rightarrow t\{x := x_1\} = t\{x := x_2\} \\ \text{IZ}^{\text{sk}} \quad \vdash \quad x_1 = x_2 &\Rightarrow \phi\{x := x_1\} \Leftrightarrow \phi\{x := x_2\} \end{aligned}$$

*Proof.* This result is proved by mutual induction on  $t$  and  $\phi$ . □

In  $Z^{\text{sk}}$  (and, actually, in  $\text{IZ}^{\text{sk}}$ ), most standard mathematical notations such as  $\emptyset$  (empty set),  $x \cup y$  (union),  $x \cap y$  (intersection),  $x \setminus y$  (difference),  $f(x)$  (function application),  $\langle x, y \rangle$  (ordered pair),  $B^A$  (function space), etc. are easily definable as macros in the enriched term algebra.

We now have to ensure that  $(\text{I})Z^{\text{sk}}$  is a conservative extension of  $(\text{I})Z$ .

### 3.3 The Deskolemisation Procedure

The proof of conservativity of  $(\text{I})Z^{\text{sk}}$  w.r.t.  $(\text{I})Z$  relies on a deskolemisation procedure that is achieved by two transformations:

- A transformation on *terms*, which maps each pair  $(t, z)$  formed by a term  $t$  of  $Z^{\text{sk}}$  and a variable  $z$  to a formula of set theory written  $z \in^\circ t$ ;<sup>4</sup>
- A transformation on *formulæ*, which maps each formula  $\phi$  of  $Z^{\text{sk}}$  to a formula of set theory written  $\phi^\circ$ .

Both transformations are defined by mutual induction on  $t$  and  $\phi$  from the deskolemisation equations given in table 5.

This process of deskolemisation preserves the meaning of terms and formulæ in  $\text{IZ}^{\text{sk}}$  in the sense that:

**Proposition 3 (Translation Equivalence).** — *For all terms  $t$  and formulæ  $\phi$  of the language of  $Z^{\text{sk}}$ , one has:*

$$\text{IZ}^{\text{sk}} \quad \vdash \quad (z \in^\circ t) \Leftrightarrow z \in t \quad \text{and} \quad \text{IZ}^{\text{sk}} \quad \vdash \quad \phi^\circ \Leftrightarrow \phi$$

Moreover, if  $\phi$  is expressed in the core language  $(=, \in)$  of set theory, then:

$$\text{IZ} \quad \vdash \quad \phi^\circ \Leftrightarrow \phi.$$

*Proof.* The first two items are proved by mutual induction on  $t$  and  $\phi$ . Last item is proved by induction on  $\phi$ . □

Furthermore, we can show:

**Proposition 4 (Soundness of Deskolemisation).** — *If a closed formula  $\phi$  is a theorem of  $(\text{I})Z^{\text{sk}}$ , then  $\phi^\circ$  is a theorem of  $(\text{I})Z$ .*

---

<sup>4</sup> Notice the conceptual similarity between the design of the deskolemisation procedure for terms  $(z \in^\circ t)$  and the notions of realisability and forcing  $(t \Vdash \phi)$ .

**Table 5.** Deskolemisation equations for terms and formulæ of  $Z^{\text{sk}}$

$z \in^\circ x$	$\equiv z \in x$																						
$z \in^\circ \omega$	$\equiv \text{Nat}(z)$																						
$z \in^\circ \{t_1; t_2\}$	$\equiv (z = t_1)^\circ \vee (z = t_2)^\circ$																						
$z \in^\circ \mathfrak{P}(t)$	$\equiv \forall x (x \in z \Rightarrow x \in^\circ t)$																						
$z \in^\circ \bigcup t$	$\equiv \exists y (y \in^\circ t \wedge z \in y)$																						
$z \in^\circ \{x \in t \mid \phi\}$	$\equiv z \in^\circ t \wedge \phi^\circ \{x := z\}$																						
<table style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td style="padding: 5px;"><math>(t = u)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \forall z (z \in^\circ t \Leftrightarrow z \in^\circ u)</math></td> <td style="padding: 5px;"><math>(\phi \wedge \psi)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \phi^\circ \wedge \psi^\circ</math></td> </tr> <tr> <td style="padding: 5px;"><math>(t \in u)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \exists x ((z = t)^\circ \wedge z \in^\circ u)</math></td> <td style="padding: 5px;"><math>(\phi \vee \psi)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \phi^\circ \vee \psi^\circ</math></td> </tr> <tr> <td></td> <td></td> <td style="padding: 5px;"><math>(\phi \Rightarrow \psi)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \phi^\circ \Rightarrow \psi^\circ</math></td> </tr> <tr> <td style="padding: 5px;"><math>\top^\circ</math></td> <td style="padding: 5px;"><math>\equiv \top</math></td> <td style="padding: 5px;"><math>(\forall x \phi)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \forall x \phi^\circ</math></td> </tr> <tr> <td style="padding: 5px;"><math>\perp^\circ</math></td> <td style="padding: 5px;"><math>\equiv \perp</math></td> <td style="padding: 5px;"><math>(\exists x \phi)^\circ</math></td> <td style="padding: 5px;"><math>\equiv \exists x \phi^\circ</math></td> </tr> </tbody> </table>				$(t = u)^\circ$	$\equiv \forall z (z \in^\circ t \Leftrightarrow z \in^\circ u)$	$(\phi \wedge \psi)^\circ$	$\equiv \phi^\circ \wedge \psi^\circ$	$(t \in u)^\circ$	$\equiv \exists x ((z = t)^\circ \wedge z \in^\circ u)$	$(\phi \vee \psi)^\circ$	$\equiv \phi^\circ \vee \psi^\circ$			$(\phi \Rightarrow \psi)^\circ$	$\equiv \phi^\circ \Rightarrow \psi^\circ$	$\top^\circ$	$\equiv \top$	$(\forall x \phi)^\circ$	$\equiv \forall x \phi^\circ$	$\perp^\circ$	$\equiv \perp$	$(\exists x \phi)^\circ$	$\equiv \exists x \phi^\circ$
$(t = u)^\circ$	$\equiv \forall z (z \in^\circ t \Leftrightarrow z \in^\circ u)$	$(\phi \wedge \psi)^\circ$	$\equiv \phi^\circ \wedge \psi^\circ$																				
$(t \in u)^\circ$	$\equiv \exists x ((z = t)^\circ \wedge z \in^\circ u)$	$(\phi \vee \psi)^\circ$	$\equiv \phi^\circ \vee \psi^\circ$																				
		$(\phi \Rightarrow \psi)^\circ$	$\equiv \phi^\circ \Rightarrow \psi^\circ$																				
$\top^\circ$	$\equiv \top$	$(\forall x \phi)^\circ$	$\equiv \forall x \phi^\circ$																				
$\perp^\circ$	$\equiv \perp$	$(\exists x \phi)^\circ$	$\equiv \exists x \phi^\circ$																				

From Prop. 3 (last equivalence) and Prop. 4 we easily deduce:

**Proposition 5 (Conservativity).** — *The theory  $(I)Z^{\text{sk}}$  is a conservative extension of  $(I)Z$ .*

*Proof.* See appendix A.

### 3.4 A Weak Form of Replacement in Zermelo’s System

Historically, one of the motivations of Fraenkel and Skolem to introduce the replacement scheme in set theory

$$(\text{REPLACEMENT}) \quad \forall a [\forall x \in a \exists !y \phi(x, y) \Rightarrow \exists b \forall x \in a \exists y \in b \phi(x, y)]$$

(which fills the gap between Z and ZF) was to justify the notation  $\{t(x) \mid x \in u\}$  which expresses the image of the set  $u$  by the functional relation  $x \mapsto t(x)$ .

Surprisingly, the study of  $Z^{\text{sk}}$  reveals that the justification of the notation  $\{t(x) \mid x \in u\}$  does not need any extension of Zermelo’s system *when the term  $t(x)$  is expressed in the term language of  $Z^{\text{sk}}$ .*

The reason is that for any term  $u$  of  $Z^{\text{sk}}$  and for any term  $t(x)$  of  $Z^{\text{sk}}$  that possibly depends on a variable  $x$ , we can define a term written  $B(t(x), x \in u)$  which uniformly bounds  $t(x)$  when  $x$  ranges over  $u$ . Formally, such a term can be defined by structural induction on  $t(x)$  as follows:

$$\begin{aligned}
 B(x, x \in u) &= u \\
 B(y, x \in u) &= \mathfrak{P}(y) \quad (\text{if } y \neq x) \\
 B(\omega, x \in u) &= \mathfrak{P}(\omega) \\
 B(\{t_1; t_2\}, x \in u) &= \mathfrak{P}(B(t_1, x \in u) \cup B(t_2, x \in u)) \\
 B(\mathfrak{P}(t), x \in u) &= \mathfrak{P}(\mathfrak{P}(\bigcup B(t, x \in u))) \\
 B(\bigcup t, x \in u) &= \mathfrak{P}(\bigcup \bigcup B(t, x \in u)) \\
 B(\{y \in t \mid \phi\}, x \in u) &= \mathfrak{P}(\bigcup B(t, x \in u))
 \end{aligned}$$



**Lemma 1.** — *For all terms  $t(x)$  and  $u$  of  $Z^{\text{sk}}$  such that  $x \notin FV(u)$ :*

$$IZ^{\text{sk}} \vdash \forall x [x \in u \Rightarrow t(x) \in \mathbf{B}(t(x), x \in u)].$$

*Proof.* By induction on  $t(x)$ . □

Setting  $\{t(x) \mid x \in u\} \equiv \{y \in \mathbf{B}(t(x), x \in u) \mid \exists x (x \in u \wedge y = t(x))\}$  we easily check that:

**Proposition 6.** — *For all terms  $t$  and  $u$  such that  $x \notin FV(u)$  and  $y \notin FV(t)$ :*

$$IZ^{\text{sk}} \vdash \forall y [y \in \{t \mid x \in u\} \Leftrightarrow \exists x (x \in u \wedge y = t)].$$

An important consequence of this result is that we can now define in the language of  $Z^{\text{sk}}$  both the notation for function abstraction and the notation for generalised Cartesian product that are crucial ingredients for any translation of type theory in set theory:

$$\begin{aligned} \lambda x \in t. u(x) &\equiv \{ \langle x, u(x) \rangle \mid x \in t \} \\ \prod_{x \in t} u(x) &\equiv \left\{ f \in \left( \bigcup \{u(x) \mid x \in t\} \right)^t \mid \forall x (x \in t \Rightarrow f(x) \in u(x)) \right\} \end{aligned}$$

(Remember that these notations are not macros, but that they denote the result of complex transformations in the term language of  $Z^{\text{sk}}$ .)

## 4 Sets as Pointed Graphs

In this section, we present the translation  $(\_)^*$  of IZ into λZ using the representation of sets as pointed graphs [1]. This translation is basically the one presented by the author in [11, 12], except that:

- The target formalism λZ is slightly weaker than the formalism (i.e.  $F\omega.3$ ) in which this translation was originally presented.
- We also prove the soundness of two additional axioms that are crucial to achieve the conservativity result (theorem 2) of section 5, namely: the anti-foundation axiom (AFA) and the axiom of the transitive closure (TC).

Before defining the translation, let us first recall the basic notions of the theory of pointed graphs (presented in set theory) that are needed to introduce the axiom of anti-foundation.

### 4.1 Pointed Graphs and Anti-foundation

In set theory, a *pointed graph* is a triple  $\langle X, R, r \rangle$  formed by an arbitrary set  $X$  (the *carrier*) equipped with a binary relation  $R \subset (X \times X)$  (the *edge relation*) and a distinguished element  $r \in X$  (the *root*).

Given a binary relation  $R$  (on any set), we say that a function  $\phi$  (whose domain is written  $D_\phi$ ) *decorates*  $R$  if for all  $x \in D_\phi$  and for any set  $z$ , the relation  $z \in \phi(x)$

holds iff there exists  $x' \in D_\phi$  such that  $z = \phi(x')$  and  $\langle x', x \rangle \in R$ . Finally, we say that a pointed graph  $\langle X, R, r \rangle$  *pictures* a set  $x$  when there exists a decoration  $\phi$  of  $R$  such that  $r \in D_\phi$  and  $\phi(r) = x$ .

Formally, the relations  $\text{PGraph}(G)$  ( $G$  is a pointed graph'),  $\text{Decor}(\phi, R)$  ('the function  $\phi$  decorates  $R$ ') and  $\text{Pict}(G, x)$  ( $G$  pictures  $x$ ') are defined by:<sup>5</sup>

$$\text{PGraph}(G) \equiv \exists X \exists R \exists r [G = \langle X, R, r \rangle \wedge R \subset (X \times X) \wedge r \in X]$$

$$\text{Decor}(\phi, R) \equiv \forall x \in D_\phi \forall z [z \in \phi(x) \Leftrightarrow \exists x' \in D_\phi (z = \phi(x') \wedge \langle x', x \rangle \in R)]$$

$$\text{Pict}(G, x) \equiv \exists X \exists R \exists r \exists \phi [G = \langle X, R, r \rangle \wedge \text{function}(\phi) \wedge \text{Decor}(\phi, R) \wedge r \in D_\phi \wedge x = \phi(r)]$$

In ZF, it is easy to show that any pointed graph  $G = \langle X, R, r \rangle$  whose root  $r$  is accessible<sup>6</sup> w.r.t. the relation  $R$  pictures a unique set.<sup>7</sup>

In presence of the axiom of foundation [8], this result cannot be extended further, for the relation  $\text{Pict}(\langle X, R, r \rangle, x)$  automatically implies the accessibility of the root  $r$  w.r.t. the relation  $R$ .

The axiom of anti-foundation (AFA) refutes the axiom of foundation by extending the latter result of existence and uniqueness to all the pointed graphs:

$$\text{(AFA)} \quad \forall G [\text{PGraph}(G) \Rightarrow \exists! x \text{Pict}(G, x)]$$

Notice that this axiom has two parts: the existence part that allows to build arbitrarily non well-founded sets (for instance, a set  $x$  such that  $x = \{x\}$ ), and the uniqueness part that allows to prove equalities between non-wellfounded sets (for instance, that any two sets  $x$  and  $y$  such that  $x = \{x\}$  and  $y = \{y\}$  are equal).

### 4.2 The Axiom of Transitive Closure

In ZF it is easy to associate to each set  $x$  a pointed graph  $\langle X, R, r \rangle$  that pictures  $x$ —a *representation* of  $x$ —simply by taking  $X = \text{Cl}(\{x\})$  the transitive closure of the singleton  $\{x\}$ ,  $R = \{(y', y) \in X \mid y' \in y\}$  and  $r = x$ .

Unfortunately, the latter construction relies on the existence of a transitive closure, which is not provable in Z [5]. For this reason, we consider the extension of Zermelo's system with the following axiom

$$\text{(TC)} \quad \forall a \exists b [a \subset b \wedge \forall x (x \in b \Rightarrow x \subset b)].$$

<sup>5</sup> Formally, these definitions are expressed in the language of  $Z^{\text{sk}}$ , but in what follows we will consider them as definitions expressed in the ordinary language of set theory, implicitly using the deskolemisation procedure presented in section 3.

<sup>6</sup> The accessibility predicate  $\text{Acc}_R(x)$  is inductively defined on  $X$  by the unique clause: if  $\text{Acc}_R(y)$  for all  $y \in X$  such that  $\langle y, x \rangle \in R$ , then  $\text{Acc}_R(x)$ .

<sup>7</sup> Actually, the proof does not rely on classical principles and can be done in  $\text{IZF}_R$  (intuitionistic ZF with replacement). In IZ, only the uniqueness is provable.

that expresses that any set is included in a transitive set. From this axiom it is easy to derive the expected representation property

$$(REPR) \quad \forall x \exists G (\text{PGraph}(G) \wedge \text{Pict}(G, x))$$

using the construction described above.<sup>8</sup>

### 4.3 The Translation of IZ into λZ

To each variable  $x$  of set theory we associate in λZ three object term variables written  $\bar{x}$  (the carrier),  $\tilde{x}$  (the relation) and  $\dot{x}$  (the root), with types

$$\bar{x} : \square_2, \quad \tilde{x} : \bar{x} \rightarrow \bar{x} \rightarrow *, \quad \dot{x} : \bar{x},$$

that are intended to represent the set  $x$  as a pointed graph  $(\bar{x}, \tilde{x}, \dot{x})$  in λZ. We also assume that for any pair  $x$  and  $y$  of distinct variables of set theory, the variables  $\bar{x}, \tilde{x}, \dot{x}, \bar{y}, \tilde{y}, \dot{y}$  are pairwise distinct.

Given a finite set  $X$  of variables of set theory, we denote by  $\Gamma_X$  the well-formed context of λZ given by

$$\Gamma_X = \bigcup_{x \in X} [\bar{x} : \square_2; \tilde{x} : \bar{x} \rightarrow \bar{x} \rightarrow *, \dot{x} : \bar{x}]$$

(here, the union refers to a concatenation of contexts whose order is irrelevant).

Given two variables  $x$  and  $y$  of set theory, the relation  $x = y$  that expresses the extensional equality of the sets  $x$  and  $y$  is interpreted in λZ as the *bisimilarity* of the pointed graphs  $\langle \bar{x}, \tilde{x}, \dot{x} \rangle$  and  $\langle \bar{y}, \tilde{y}, \dot{y} \rangle$ , namely, as the proposition written  $\langle \bar{x}, \tilde{x}, \dot{x} \rangle \approx \langle \bar{y}, \tilde{y}, \dot{y} \rangle$  and defined by

$$\begin{aligned} \langle \bar{x}, \tilde{x}, \dot{x} \rangle \approx \langle \bar{y}, \tilde{y}, \dot{y} \rangle &\equiv \\ \exists r : (\bar{x} \rightarrow \bar{y} \rightarrow *) &. \\ [ R \dot{x} \dot{y} & \\ \forall \alpha, \alpha' : \bar{x}. \forall \beta : \bar{y}. (\tilde{x} \alpha' \alpha \wedge r \alpha \beta \Rightarrow \exists \beta' : \bar{y}. (\tilde{y} \beta' \beta \wedge r \alpha' \beta')) \wedge & \\ \forall \beta, \beta' : \bar{y}. \forall \alpha : \bar{x}. (\tilde{y} \beta' \beta \wedge r \alpha \beta \Rightarrow \exists \alpha' : \bar{x}. (\tilde{x} \alpha' \alpha \wedge r \alpha' \beta')) ] & \wedge \end{aligned}$$

To each formula  $\phi$  of set theory we associate a proposition  $\phi^*$  of λZ by setting

$$\begin{aligned} (x = y)^* &\equiv \langle \bar{x}, \tilde{x}, \dot{x} \rangle \approx \langle \bar{y}, \tilde{y}, \dot{y} \rangle \\ (x \in y)^* &\equiv \exists z : \bar{y}. (\tilde{y} z \dot{y} \wedge \langle \bar{x}, \tilde{x}, \dot{x} \rangle \approx \langle \bar{y}, \tilde{y}, z \rangle) \\ (\phi \wedge \psi)^* &\equiv \phi^* \wedge \psi^* \quad (\top)^* \equiv \top \\ (\phi \vee \psi)^* &\equiv \phi^* \vee \psi^* \quad (\perp)^* \equiv \perp \\ (\phi \Rightarrow \psi)^* &\equiv \phi^* \Rightarrow \psi^* \\ (\forall x \phi)^* &\equiv \forall \bar{x} : \square_2. \forall \tilde{x} : (\bar{x} \rightarrow \bar{x} \rightarrow *). \forall \dot{x} : \bar{x}. \phi^* \\ (\exists x \phi)^* &\equiv \exists \bar{x} : \square_2. \exists \tilde{x} : (\bar{x} \rightarrow \bar{x} \rightarrow *). \exists \dot{x} : \bar{x}. \phi^* \end{aligned}$$

It is easy to check that  $FV(\phi^*) = \bigcup_{x \in FV(x)} \{\bar{x}; \tilde{x}; \dot{x}\}$  and that  $\Gamma_{FV(\phi)} \vdash \phi^* : *$ .

**Theorem 1 (Soundness).** — *For all formulæ  $\phi$  of set theory such that IZ + AFA + TC  $\vdash \phi$ , there is a proof-term  $t$  such that  $\Gamma_{FV(\phi)} \vdash t : \phi^*$ .*

*Proof.* See appendix B.

<sup>8</sup> The proposition (REPR) is actually equivalent to (TC) in IZ + AFA.

## 5 Retracting $\lambda Z$ in $Z^{\text{sk}} + \text{AFA}$

We now define a converse translation  $(\_)^\dagger$  from  $\lambda Z$  to  $\text{IZ}^{\text{sk}} + \text{AFA}$ , using the standard types-in-sets interpretation [13, 2]. Notice that here, we only need anti-foundation (AFA) to justify the existence of the set  $\text{HF}$  of hereditarily finite sets (a.k.a.  $V_\omega$ ) [8], which is used to interpret the sort  $\square_1$ .<sup>9</sup>

### 5.1 The Translation $M \mapsto M^\dagger$

Raw object terms of  $\lambda Z$  (cf table 1) are translated into terms of  $Z^{\text{sk}}$  as follows:

$$\begin{aligned}
 x^\dagger &\equiv x \\
 *^\dagger &\equiv \mathfrak{P}(\{\bullet\}) \\
 \square_1^\dagger &\equiv \text{HF} \\
 \square_2, \square_3 &\text{ no translation} \\
 (\Pi x : T . U)^\dagger &\equiv \prod_{x \in T^\dagger} U^\dagger \\
 (\lambda x : T . M)^\dagger &\equiv \lambda x \in T^\dagger . M^\dagger \\
 (MN)^\dagger &\equiv M^\dagger(N^\dagger) \\
 (A \Rightarrow B)^\dagger &\equiv \{\pi \in \{\bullet\} \mid \bullet \in A^\dagger \Rightarrow \bullet \in B^\dagger\} \\
 \forall x : \square_2 . A &\equiv \{\pi \in \{\bullet\} \mid \forall x (\bullet \in A^\dagger)\} \\
 \forall x : T . A &\equiv \{\pi \in \{\bullet\} \mid \forall x (x \in T^\dagger \Rightarrow \bullet \in A^\dagger)\}
 \end{aligned}$$

This translation is partial, and associates no term to the sorts  $\square_2$  and  $\square_3$ . (Notice that  $FV(M^\dagger) = FV(M)$  whenever  $M^\dagger$  is defined.)

Propositions are interpreted in a proof-irrelevant way, as subsets of a singleton  $\{\bullet\}$ , where  $\bullet$  is any closed term of  $Z^{\text{sk}}$ . We use here the standard trick by which any (intuitionistic) formula  $\phi$  of  $Z^{\text{sk}}$  can be encoded as a subset  $\hat{\phi} \subset \{\bullet\}$  by setting  $\hat{\phi} = \{\pi \in \{\bullet\} \mid \phi\}$  whereas any subset  $p \subset \{\bullet\}$  naturally decodes to the formula  $\bullet \in p$ . This correspondence between propositions and subsets of  $\{\bullet\}$  is one-to-one,<sup>10</sup> in this sense that the equivalence  $\phi \Leftrightarrow (\bullet \in \hat{\phi})$  is provable in  $\text{IZ}^{\text{sk}}$  (for all formulæ  $\phi$ ), as well as the implication  $p \subset \{\bullet\} \Rightarrow (p = \widehat{\bullet \in p})$ .

Up to this coding trick, the different kinds of universal quantifications are interpreted exactly as outlined in subsection 2.3. In particular, universal quantifications of the form  $\forall x : \square_2 \dots$  are treated in a separate case, using unbounded quantification of set theory.

Contexts of  $\lambda Z$  are translated as formulæ of  $Z^{\text{sk}}$  as follows:

$$\begin{aligned}
 (\square)^\dagger &\equiv \top \\
 (\Gamma; x : \square_2)^\dagger &\equiv \Gamma^\dagger \\
 (\Gamma; x : T)^\dagger &\equiv \Gamma^\dagger \wedge (x \in T^\dagger) \quad (\text{if } T \neq \square_2) \\
 (\Gamma, \xi : A)^\dagger &\equiv \Gamma^\dagger \wedge (\bullet \in A^\dagger)
 \end{aligned}$$

<sup>9</sup> Remember that the existence of the set  $\text{HF}$  of all hereditarily finite sets cannot be justified in  $\text{IZ}$  ([8], p. 238, exercise 10). This is no more the case if we extend the system with AFA, in which case  $\text{HF}$  can be reconstructed as the reification of a universal (and denumerable) pointed graph whose root points to all the finite trees.

<sup>10</sup> Classically, this is even more obvious since  $\mathfrak{P}(\{\bullet\}) = \{\emptyset; \{\bullet\}\}$ .

Notice that  $FV(\Gamma^*) \subset FV(\Gamma)$ . In particular, proof-term variables are systematically erased (as well as object term variables declared with type  $\square_2$ .)

**Proposition 7 (Soundness).** — *For any derivable judgment of λZ of the form  $\Gamma \vdash M : T$  where  $M$  and  $T$  are object terms such that  $T$  is neither  $\square_2$  nor  $\square_3$ , one has:*

$$\text{IZ}^{\text{sk}} \vdash \Gamma^\dagger \Rightarrow M^\dagger \in T^\dagger$$

*Proof.* By induction on the derivation of  $\Gamma \vdash M : T$ .

**Proposition 8 (Soundness).** — *For any derivable judgment of λZ of the form  $\Gamma \vdash t : A$  where  $t$  is a proof-term and  $A$  and object term, one has:*

$$\text{IZ}^{\text{sk}} \vdash \Gamma^\dagger \Rightarrow \bullet \in A^\dagger$$

*Proof.* By induction on the derivation of  $\Gamma \vdash t : A$ .

Since the equality  $(\Pi x : * . x)^\dagger = \emptyset$  is easily provable in IZ, the latter proposition implies that the translation  $M \mapsto M^\dagger$  transforms any inconsistency of λZ (given as a closed proof-term of  $\Pi x : * . x$ ) into an inconsistency of IZ + AFA (expressed as a proof of  $\bullet \in \emptyset$ ). Combining this with theorem 1 we get:

**Proposition 9 (Equiconsistency).** — *The theories IZ + AFA + TC and λZ are equiconsistent.*

However, this result of equiconsistency can be refined as a result of conservativity by studying the composition of the translations  $(\_)^*$  and  $(\_)^\dagger$ .

## 5.2 Composing Both Translations

The translation  $(\_)^*$  from IZ to λZ rephrases each formula of set theory in graph-theoretic terms by replacing each variable  $x$  (of set theory) by three variables  $\bar{x} : \square_2$ ,  $\tilde{x} : \bar{x} \rightarrow \bar{x} \rightarrow *$  and  $\hat{x} : \bar{x}$  that denote a pointed graph representing  $x$ .

Via the translation  $(\_)^\dagger$ , each type-theoretic pointed graph  $(X, R, r)$  becomes in turn a set-theoretic pointed graph  $\langle X^\dagger, R^\dagger, r^\dagger \rangle$ , up to this (minor) difference that the edge relation  $R^\dagger$  is not given as a subset of  $X^\dagger \times X^\dagger$ , but as an element of the function space  $X^\dagger \rightarrow X^\dagger \rightarrow \mathfrak{P}(\{\bullet\})$  which is clearly isomorphic to the set  $\mathfrak{P}(X^\dagger \times X^\dagger)$  using the same coding trick as before. (For the sake of clarity, both sets  $X \rightarrow X \rightarrow \mathfrak{P}(\{\bullet\})$  and  $\mathfrak{P}(X \times X)$  will be identified in what follows.)

Consequently, the composition  $(\_)^{*\dagger}$  of both translations is nothing but the graph-theoretic rephrasing of non-well founded set theory into set theory itself. Using the anti-foundation axiom AFA together with TC we easily close the diagram as follows:

**Proposition 10 (Composition).** — *Let  $\phi$  be a formula of IZ with free variables  $x_1, \dots, x_n$ . If  $y_1, \dots, y_n$  are variables such that the variables  $x_1, \dots, x_n, \bar{y}_1, \dots, \bar{y}_n, \tilde{y}_1, \dots, \tilde{y}_n$  and  $\hat{y}_1, \dots, \hat{y}_n$  are pairwise distinct, then:*

$$\text{IZ} + \text{AFA} + \text{TC} \vdash \left( \bigwedge_{i=1}^n \text{Pict}((\bar{y}_i, \tilde{y}_i, \hat{y}_i), x_i) \right) \Rightarrow (\phi \Leftrightarrow \bullet \in \phi\{\bar{x} := \bar{y}\}^{*\dagger})$$

*Proof.* By induction on the formula  $\phi$ . AFA and TC are used to treat the case of atomic formulæ  $x = y$  and  $x \in y$  as well as quantifiers  $\forall x \psi$  and  $\exists x \psi$ .  $\square$

When  $\phi$  is a closed formula, the equivalence  $\phi \Leftrightarrow \bullet \in \phi^{*\dagger}$  is thus provable in  $\text{IZ} + \text{AFA} + \text{TC}$ . Consequently:

**Theorem 2 (Conservativity).** — *Via the embedding  $\phi \mapsto \phi^*$ ,  $\lambda Z$  is a conservative extension of  $\text{IZ} + \text{AFA} + \text{TC}$ .*

## References

1. P. Aczel. Non well-founded sets. *Center for the Study of Language and Information*, 1988.
2. Peter Aczel. On relating type theories and set theories. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *TYPES*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1998.
3. Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier and MIT Press, 2001.
4. Thierry Coquand and Hugo Herbelin. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, 4(1):77–88, 1994.
5. Olivier Esser and Roland Hinnion. Antifoundation and transitive closure in the system of Zermelo. *Notre Dame Journal of Formal Logic*, 40(2):197–205, 1999.
6. H. Friedman. Some applications of Kleene’s methods for intuitionistic systems. In *Cambridge Summer School in Mathematical Logic*, volume 337 of *Springer Lecture Notes in Mathematics*, pages 113–170. Springer-Verlag, 1973.
7. J.H. Geuvers and M.J. Nederhof. A modular proof of strong normalization for the calculus of constructions. In *Journal of Functional Programming*, volume 1,2(1991), pages 155–189, 1991.
8. J.-L. Krivine. *Théorie des ensembles*. Cassini, 1998.
9. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
10. Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *TYPES*, volume 1512 of *Lecture Notes in Computer Science*, pages 254–276. Springer, 1996.
11. A. Miquel. *Le calcul des constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, 2001.
12. Alexandre Miquel. A strongly normalising Curry-Howard correspondence for IZF set theory. In Matthias Baaz and Johann A. Makowsky, editors, *CSL’03*, volume 2803 of *Lecture Notes in Computer Science*, pages 441–454. Springer, 2003.
13. Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 530–346. Springer, 1997.

## Appendix :

### A Soundness of Deskolemisation (from $\text{IZ}^{\text{sk}}$ to $\text{IZ}$ )

The proof of the soundness of the deskolemisation procedure (Prop. 4) actually involves several intermediate steps that we briefly sketch here.

**Fact 2.** — For all terms  $t$  and formulæ  $\phi$  of the language of  $Z^{\text{sk}}$ , one has  $FV(z \in^\circ t) = FV(t) \cup \{z\}$  and  $FV(\phi^\circ) = FV(\phi)$ .

*Proof.* By mutual induction on  $t$  and  $\phi$ . □

To prove that the deskolemisation procedure transforms each theorem  $\phi$  of  $(I)Z^{\text{sk}}$  into a theorem  $\phi^\circ$  of  $(I)Z$ , we first check that each term of the extended language  $Z^{\text{sk}}$  corresponds to a set whose existence can be proved in  $Z$ :

**Lemma 2 (Collection).** — For each term  $t$  of  $Z^{\text{sk}}$ , one has:

$$IZ \vdash \exists x \forall z [z \in x \Leftrightarrow z \in^\circ t] \quad (x \text{ and } z \text{ fresh})$$

*Proof.* By structural induction on  $t$ , using the corresponding existential axiom of Zermelo's system for each syntactic construct of the term algebra of  $Z^{\text{sk}}$ . □

**Lemma 3.** — For each axiom  $\phi$  of  $Z^{\text{sk}}$ , one has:  $IZ \vdash \phi^\circ$ .

Notice that for all axioms  $\phi$  of  $Z^{\text{sk}}$ , the proof of  $\phi^\circ$  only relies on the equality axioms and the axiom of extensionality. In the deskolemisation process, Zermelo's existential axioms actually play their role in the deduction rules that involve a substitution, and whose translation relies on the following lemma:

**Lemma 4 (Substitutivity).** — For all formulæ  $\phi$  and for all terms  $t$  and  $u$  of  $Z^{\text{sk}}$  one has the equivalences:

1.  $IZ \vdash y \in^\circ t\{x := u\} \Leftrightarrow \exists x [y \in^\circ t \wedge \forall z (z \in x \Leftrightarrow z \in^\circ u)] \quad (y \neq x)$
2.  $IZ \vdash (\phi\{x := u\})^\circ \Leftrightarrow \exists x [\phi^\circ \wedge \forall z (z \in x \Leftrightarrow z \in^\circ u)]$

*Proof.* We first prove by mutual induction on  $t$  and  $\phi$  that:

1.  $IZ \vdash \forall x [\forall z (z \in x \Leftrightarrow z \in^\circ u) \Rightarrow \forall z (z \in^\circ t\{x := u\} \Leftrightarrow z \in^\circ t)]$
2.  $IZ \vdash \forall x [\forall z (z \in x \Leftrightarrow z \in^\circ u) \Rightarrow (\phi\{x := u\})^\circ \Leftrightarrow \phi^\circ]$

We then conclude that the desired equivalences hold by using lemma 2, whose proof relies on Zermelo's existential axioms. □

**Lemma 5 (Deskolemisation of a Derivation).** — Let  $A$  be a formula and  $\Gamma$  a list of formulæ both expressed in the language of  $Z^{\text{sk}}$ . If  $\Gamma \vdash A$  is classically (resp. intuitionistically) derivable, then there exists a list  $\Delta$  of axioms of  $Z$  such that  $\Delta, \Gamma^\circ \vdash A^\circ$  is classically (resp. intuitionistically) derivable.

*Proof.* By induction on the derivation of  $\Gamma \vdash A$ . The only interesting cases correspond to the rules  $\forall$ -elim and  $\exists$ -intro, whose translation rely on lemma 4. □

From lemmas 3 and 5 it is then clear that:

**Proposition 11 (Soundness of Deskolemisation).** — If a closed formula  $\phi$  is a theorem of  $(I)Z^{\text{sk}}$ , then  $\phi^\circ$  is a theorem of  $(I)Z$ .

## B Soundness of the Translation $\phi \mapsto \phi^*$

The translation  $(\_)^*$  from  $\text{IZ} (+ \text{AFA} + \text{TC})$  into  $\lambda\text{Z}$  depicted in section 4 is actually a fragment of a translation of  $\text{IZ}^{\text{sk}} (+ \text{AFA} + \text{TC})$  into  $\lambda\text{Z}$ , in which the pointed graphs associated to sets are explicitly built from the terms of  $\text{Z}^{\text{sk}}$ .

Formally, this translation maps

- Each variable  $x$  of set theory to three variables  $\bar{x}$ ,  $\tilde{x}$  and  $\dot{x}$  of  $\lambda\text{Z}$ , declared (in this order) as follows:  $\bar{x} : \square_2$ ,  $\tilde{x} : \bar{x} \rightarrow \bar{x} \rightarrow *$ ,  $\dot{x} : \bar{x}$ .
- Each formula  $\phi$  of the language of  $\text{Z}^{\text{sk}}$  to a term  $\phi$  of  $\lambda\text{Z}$  of type  $*$  in the typing context associated to the free variables of  $\phi$ .
- Each term  $t$  of the language of  $\text{Z}^{\text{sk}}$  to three terms  $t^{\bar{*}} : \square_2$ ,  $t^{\tilde{*}} : t^{\bar{*}} \rightarrow t^{\bar{*}} \rightarrow *$  and  $t^{\dot{*}} : t^{\bar{*}}$  of  $\lambda\text{Z}$  (in the typing context associated to the free variables of  $t$ ) that respectively represent the carrier, the edge relation and the root of the pointed graph that represents the set denoted by  $t$  in  $\lambda\text{Z}$ .

### B.1 Logic and Data Types

The formal definition of the translation relies on the usual second-order encodings of connectives, existential quantifier and Leibniz equality in  $\lambda\text{Z}$ :

$$\begin{aligned}
 \perp &\equiv \forall \gamma : *. \gamma & \top &\equiv \forall \gamma : *. (\gamma \Rightarrow \gamma) \\
 A \wedge B &\equiv \forall \gamma : *. ((A \Rightarrow B \Rightarrow \gamma) \Rightarrow \gamma) \\
 A \vee B &\equiv \forall \gamma : *. ((A \Rightarrow \gamma) \Rightarrow (B \Rightarrow \gamma) \Rightarrow \gamma) \\
 \exists x : T. A(x) &\equiv \forall \gamma : *. (\forall x : T. (A(x) \Rightarrow \gamma) \Rightarrow \gamma) \\
 x =_T y &\equiv \forall \gamma : (T \rightarrow *). (\gamma x \Rightarrow \gamma y)
 \end{aligned}$$

Given two types  $X, Y : \square_2$  we define two data-types  $\text{opt}(X) : \square_2$  ('pseudo option type') and  $\text{sum}(X, Y) : \square_2$  ('pseudo union type') as follows:

$$\begin{aligned}
 \text{opt}(X) &: \square_2 && \equiv (X \rightarrow *) \rightarrow * \\
 \text{some}(X, x) &: \text{opt}(X) && \equiv \lambda f : (X \rightarrow *). f \ x && (x : X) \\
 \text{none}(X) &: \text{opt}(X) && \equiv \lambda f : (X \rightarrow *). \perp \\
 \text{sum}(X, Y) &: \square_2 && \equiv (X \rightarrow *) \rightarrow (Y \rightarrow *) \rightarrow * \\
 \text{inl}(X, Y, x) &: \text{sum}(X, Y) && \equiv \lambda f : (X \rightarrow *). \lambda g : (Y \rightarrow *). f \ x && (x : X) \\
 \text{inr}(X, Y, y) &: \text{sum}(X, Y) && \equiv \lambda f : (X \rightarrow *). \lambda g : (Y \rightarrow *). g \ y && (y : Y) \\
 \text{out}(X, Y) &: \text{sum}(X, Y) && \equiv \lambda f : (X \rightarrow *). \lambda g : (Y \rightarrow *). \perp
 \end{aligned}$$

It can be shown [11] that the constructions  $\text{some}(X, x)$  and  $\text{none}(X)$  (for the data type  $\text{opt}(X)$ ) and the constructions  $\text{inl}(X, Y, x)$ ,  $\text{inr}(X, Y, y)$  and  $\text{out}(X, Y)$  (for the data type  $\text{sum}(X, Y)$ ) behave as constructors in the sense that they enjoy the expected properties of injectivity and non-confusion. On the other hand, these data types (that actually contain much more values than the ones introduced by the constructors) have no associated elimination principle.

The type  $\text{nat}$  of Church numerals is easily constructed in  $\square_2$  as shown below. As usual, this definition is accompanied with a relativisation predicate  $\text{wf\_nat}(n)$



which captures the induction strength. We also introduce the definition of large and strict ordering on natural numbers:

$$\begin{aligned}
 \mathbf{nat} &\equiv \Pi Z : \square_1 . (Z \rightarrow (Z \rightarrow Z) \rightarrow Z) \\
 0 &\equiv \lambda Z : \square_1 . \lambda z : Z . \lambda f : (Z \rightarrow Z) . z \\
 S(n) &\equiv \lambda Z : \square_1 . \lambda z : Z . \lambda f : (Z \rightarrow Z) . f (n \ Z \ z \ f) \\
 n \leq m &\equiv \forall P : (\mathbf{nat} \rightarrow *) . (P \ n \ \Rightarrow \ \forall z : \mathbf{nat} . (P \ z \ \Rightarrow \ P \ S(z)) \ \Rightarrow \ P \ m) \\
 n < m &\equiv S(n) \leq m \qquad \qquad \qquad \mathbf{wf\_nat}(n) = 0 \leq n
 \end{aligned}$$

(assuming  $n, m : \mathbf{nat}$ ). This encoding is sound w.r.t. all principles of Heyting arithmetic provided all quantifications on the type  $\mathbf{nat}$  are relativised to the class defined by the predicate  $\mathbf{wf\_nat}$ .

### B.2 Translation of Terms and Formulæ

The four transformations  $\phi \mapsto \phi^*$  (proposition),  $t \mapsto t^*$  (carrier),  $t \mapsto t^{\bar{*}}$  (edge relation) and  $t \mapsto t^{\dot{*}}$  (root) are defined by mutual induction on  $\phi$  and  $t$ .

*Translation of Formulæ.* The translation  $\phi \mapsto \phi^*$  is defined by:

$$\begin{aligned}
 (t = u)^* &\equiv (t^{\bar{*}}, t^{\dot{*}}, t^{\dot{*}}) \approx (u^{\bar{*}}, u^{\dot{*}}, u^{\dot{*}}) \\
 (t \in u)^* &\equiv \exists \beta : u^{\bar{*}} . (u^{\dot{*}} \beta \ u^{\dot{*}} \wedge (t^{\bar{*}}, t^{\dot{*}}, t^{\dot{*}}) \approx (u^{\bar{*}}, u^{\dot{*}}, \beta)) \\
 (\phi \diamond \psi)^* &\equiv \phi^* \diamond \psi^* \qquad U^* \equiv U \qquad (\diamond = \wedge, \vee, \Rightarrow \quad U = \top, \perp) \\
 (Qx \ \phi)^* &\equiv Q\bar{x} : \square_2 . Q\tilde{x} : (\bar{x} \rightarrow \bar{x} \rightarrow *) . Q\dot{x} : \bar{x} . \phi^* \qquad (Q = \forall, \exists)
 \end{aligned}$$

(where  $\approx$  denotes the type-theoretic expression of the bisimilarity relation, that has been already given in subsection 4.3).

*Translation of Terms.* The translations  $t \mapsto t^*$ ,  $t \mapsto t^{\bar{*}}$  and  $t \mapsto t^{\dot{*}}$  are defined in table 6. For the sake of clarity, we omit type parameters  $X$  and  $Y$  in the constructors `some`, `none`, `inl`, `inr`, `out` and Leibniz equality ‘=’.

### B.3 Soundness of the Axioms of $\mathbf{IZ}^{\text{sk}}$

**Lemma 6.** — *For each axiom  $\phi$  of  $\mathbf{IZ}^{\text{sk}}$ , the proposition  $\phi^* : *$  has a closed proof-term in  $\lambda Z$ .*

The proof essentially proceeds as in [11], except that the target formalism  $\lambda Z$  is slightly weaker than  $F\omega.3$ , in which the translation was originally presented. Technically, the difference appears with the comprehension scheme, whose translation in  $F\omega.3$  benefits from the possibility of encoding class abstraction (using the rule  $(\square_3, \square_1, \square_3)$ ) and class quantification (using rule  $(\square_3, *, *)$ ) so that comprehension can be expressed as a single proposition.<sup>11</sup> In  $\lambda Z$  however, class abstraction is not possible anymore, and each instance of the comprehension scheme has to be translated separately.

<sup>11</sup> The main reason is that  $F\omega.3$  ( $\supseteq \lambda Z$ ) actually captures the strength of  $\mathbf{IZ}\omega$  (intuitionistic higher-order Zermelo's set theory).

**Table 6.** Translation of the terms of  $Z^{\text{sk}}$  in  $\lambda Z$ 


---



---

<u>Variables</u>	
$x^{\bar{*}} \equiv \bar{x}$ ,	$x^{*} \equiv \tilde{x}$ and $x^{\dot{*}} \equiv \dot{x}$
<u>Set of von Neumann numerals</u>	
$\omega^{\bar{*}} \equiv \text{opt}(\text{nat})$ ,	$\omega^{*} \equiv \text{none}$
$\omega^{\dot{*}} \equiv \lambda\beta', \beta: \text{opt}(\text{nat}).$	
	$\exists n', n: \text{nat}. (\text{wf\_nat}(n') \wedge \beta' = \text{some}(n') \wedge$ $\text{wf\_nat}(n) \wedge \beta = \text{some}(n) \wedge n' < n)$ $\vee \exists n': \text{nat}. (\text{wf\_nat}(n') \wedge \beta' = \text{some}(n') \wedge \beta = \text{none})$
<u>Unordered pair</u>	
$\{t_1; t_2\}^{\bar{*}} \equiv \text{sum}(t_1^{\bar{*}}, t_2^{\bar{*}})$ ,	$\{t_1; t_2\}^{*} \equiv \text{out}$
$\{t_1; t_2\}^{\dot{*}} \equiv \lambda\beta', \beta: \text{sum}(t_1^{\dot{*}}, t_2^{\dot{*}}).$	
	$\exists \alpha', \alpha: t_1^{\dot{*}}. (\beta' = \text{inl}(\alpha') \wedge \beta = \text{inl}(\alpha) \wedge t_1^{\dot{*}} \alpha' \alpha)$ $\vee \exists \alpha', \alpha: t_2^{\dot{*}}. (\beta' = \text{inr}(\alpha') \wedge \beta = \text{inr}(\alpha) \wedge t_2^{\dot{*}} \alpha' \alpha)$ $\vee (\beta' = \text{inl}(t_1^{\dot{*}}) \wedge \beta = \text{out})$ $\vee (\beta' = \text{inr}(t_2^{\dot{*}}) \wedge \beta = \text{out})$
<u>Powerset</u>	
$(\mathfrak{P}(t))^{\bar{*}} \equiv \text{sum}(t^{\bar{*}}, t^{\bar{*}} \rightarrow *)$ ,	$(\mathfrak{P}(t))^{*} \equiv \text{out}$
$(\mathfrak{P}(t))^{\dot{*}} \equiv \lambda\beta', \beta: \text{sum}(t^{\dot{*}}, t^{\dot{*}} \rightarrow *).$	
	$\exists \alpha', \alpha: t^{\dot{*}}. (\beta' = \text{inl}(\alpha') \wedge \beta = \text{inl}(\alpha) \wedge t^{\dot{*}} \alpha' \alpha)$ $\vee \exists \alpha: t^{\dot{*}}. \exists p: (t^{\dot{*}} \rightarrow *). (\beta' = \text{inl}(\alpha) \wedge \beta = \text{inr}(p) \wedge t^{\dot{*}} \alpha t^{\dot{*}} \wedge p \alpha)$ $\vee \exists p: (t^{\dot{*}} \rightarrow *). (\beta' = \text{inr}(p) \wedge \beta = \text{out})$
<u>Union</u>	
$(\bigcup t)^{\bar{*}} \equiv \text{opt}(t^{\bar{*}})$ ,	$(\bigcup t)^{*} \equiv \text{none}$
$(\bigcup t)^{\dot{*}} \equiv \lambda\beta', \beta: \text{opt}(t^{\dot{*}}).$	
	$\exists \alpha', \alpha: t^{\dot{*}}. (\beta' = \text{some}(\alpha') \wedge \beta = \text{some}(\alpha) \wedge t^{\dot{*}} \alpha' \alpha)$ $\vee \exists \alpha', \alpha: t^{\dot{*}}. (\beta' = \text{some}(\alpha') \wedge \beta = \text{none} \wedge t^{\dot{*}} \alpha' \alpha \wedge t^{\dot{*}} \alpha t^{\dot{*}})$
<u>Comprehension</u>	
$\{x \in t \mid \phi\}^{\bar{*}} \equiv \text{opt}(t^{\bar{*}})$ ,	$\{x \in t \mid \phi\}^{*} \equiv \text{none}$
$\{x \in t \mid \phi\}^{\dot{*}} \equiv \lambda\beta', \beta: \text{opt}(t^{\dot{*}}).$	
	$\exists \alpha', \alpha: t^{\dot{*}}. (\beta' = \text{some}(\alpha') \wedge \beta = \text{some}(\alpha) \wedge t^{\dot{*}} \alpha' \alpha)$ $\vee \exists \alpha: t^{\dot{*}}. (\beta' = \text{some}(\alpha) \wedge \beta = \text{none} \wedge$ $t^{\dot{*}} \alpha t^{\dot{*}} \wedge \phi^* \{ \bar{x} := t^{\dot{*}}; \tilde{x} := t^{\dot{*}}; \dot{x} := \alpha \})$

---



---

### B.4 Soundness of Anti-foundation

The soundness of AFA in λZ is an exercise of decoding a pointed graph structure from the type-theoretic representation of a set-theoretic pointed graph.

Let us assume that  $(X, R, r)$  is a type-theoretic pointed graph that represents a set-theoretic pointed graph, that is, three terms

$$X : \square_2, \quad R : X \rightarrow X \rightarrow * \quad \text{and} \quad r : X$$

of λZ such that the proposition  $\text{PGraph}^*(X, R, r)$  is provable in λZ, where  $\text{PGraph}$  is the set-theoretic predicate defined by

$$\text{PGraph}(x) \equiv \exists x_1 \exists x_2 \exists x_3 [G = \langle x_1, x_2, x_3 \rangle \wedge x_2 \subset (x_1 \times x_1) \wedge x_3 \in x_1]$$

From the assumption  $\text{PGraph}^*(X, R, r)$ , we can easily extract three pointed graphs  $(X_1, R_1, r_1)$ ,  $(X_2, R_2, r_2)$  and  $(X_3, R_3, r_3)$  representing the three components  $x_1$ ,  $x_2$  and  $x_3$  of the set-theoretic triple represented by  $(X, R, r)$ . In particular, we know that  $x_3 \in x_1$  so that there is a vertex  $\alpha_0 : X_1$  such that the pointed graphs  $(X_3, R_3, r_3)$  and  $(X_1, R_1, \alpha_0)$  are bisimilar.

We now have to build in λZ a pointed graph  $(Y, S, s)$  that represents the set pictured by the set-theoretic pointed graph whose carrier, edge relation and root are represented by the pointed graphs  $(X_1, R_1, r_1)$ ,  $(X_2, R_2, r_2)$  and  $(X_3, R_3, r_3)$ . This pointed graph  $(Y, S, s)$  is constructed from the pointed graph  $(X_1, R_1, r_1)$  by changing the edge relation and root as follows:

$$\begin{aligned} Y &\equiv X_1 & s &\equiv \alpha \\ S &\equiv \lambda \alpha', \alpha : X_1. \quad R_1 \alpha' r_1 \wedge R_1 \alpha r_1 \wedge \\ &\quad \text{Rel}^*((X_1, R_1, \alpha'), (X_1, R_1, \alpha), (X_2, R_2, r_2)) \end{aligned}$$

where  $\text{Rel}(x, y, z)$  is the set-theoretic formula  $\langle x, y \rangle \in z$ . We then check that  $\text{Pict}^*((X, R, r), (Y, S, s))$  holds in λZ, and that any pointed graph  $(Y', S', s')$  such that  $\text{Pict}^*((X, R, r), (Y', S', s'))$  is bisimilar to  $(Y, S, s)$ . (The proof is the type-theoretic transposition of the validity proof of AFA presented in [1].)

### B.5 Soundness of Transitive Closure

The transitive closure of a pointed graph  $(X, R, r)$  is represented in λZ as the pointed graph  $(Y, S, s)$  whose components are given by:

$$\begin{aligned} Y &\equiv \text{opt}(X) & s &\equiv \text{none} \\ S &\equiv \lambda \beta', \beta : Y. \\ &\quad \exists \alpha', \alpha : X. (\beta' = \text{some}(\alpha') \wedge \beta = \text{some}(\alpha) \wedge R \alpha' \alpha) \\ &\quad \vee \exists \alpha : X. (\beta' = \text{some}(\alpha) \wedge \beta = \text{none} \wedge R^+ \alpha r) \end{aligned}$$

where  $R^+$  denotes the transitive closure of  $R$  (expressed in λZ), namely, the binary relation on  $X$  defined by

$$\begin{aligned} R^+ &\equiv \lambda \alpha_1, \alpha_2 : X. \forall r : (X \rightarrow X \rightarrow *). \\ &\quad [\forall \alpha', \alpha : X. (R \alpha' \alpha \Rightarrow r \alpha' \alpha) \wedge \\ &\quad \forall \alpha'', \alpha', \alpha : X. (r \alpha'' \alpha' \Rightarrow r \alpha' \alpha \Rightarrow r \alpha'' \alpha) \\ &\quad \Rightarrow r \alpha_1 \alpha_2]. \end{aligned}$$

# Exploring the Regular Tree Types

Peter Morris, Thorsten Altenkirch, and Conor McBride

School of Computer Science and Information Technology,  
University of Nottingham

**Abstract.** In this paper we use the Epigram language to define the universe of regular tree types—closed under empty, unit, sum, product and least fixpoint. We then present a generic decision procedure for Epigram’s in-built equality at each type, taking a complementary approach to that of Benke, Dybjer and Jansson [7]. We also give a generic definition of map, taking our inspiration from Jansson and Jeuring [21]. Finally, we equip the regular universe with the partial derivative which can be interpreted functionally as Huet’s notion of ‘zipper’, as suggested by McBride in [27] and implemented (without the fixpoint case) in Generic Haskell by Hinze, Jeuring and Löh [18]. We aim to show through these examples that generic programming can be ordinary programming in a dependently typed language.

## 1 Introduction

This paper is about generic programming [6] in the dependently typed functional language Epigram [29, 30]. Generic programming allows programmers to explain how a single algorithm can be instantiated for a variety of datatypes, by computation over each datatype’s structure. In particular, we construct the universe of regular tree types—the datatypes closed under empty, unit, sum, product and least fixpoint. We define a de Bruijn indexed syntax [14] for these types, but we do not interpret this syntax via a recursive function: rather we give the elements for a given type as an *inductive family* [16]. It is Epigram’s support for dependent pattern matching [13] which makes this approach practicable.

The universe of regular tree types is small compared to others we might imagine [4, 7], but it is rich in structure. We exploit some of that structure in our programs: Epigram’s standard equality is decidable for every regular tree type; every regular tree type constructor has a notion of functorial ‘map’; we also give the formal derivative of each type expression, including fixpoints, and acquire the related notion of one-hole context or ‘zipper’ [20]. In the last example McBride’s observation [27], given its explanation in [3], has finally become a program.

### 1.1 What Is a Universe?

The notion of a *universe* in Type Theory was introduced by Per Martin-Löf [26, 34] as a means to abstract over specific collections of types. A universe is given by a type  $U : \star$  of *codes* representing just the types in the collection, and a function  $T : U \rightarrow \star$  which interprets each code as a type. A standard example

is a universe of *finite* types—each type may be coded by a natural number representing its size. We can declare the natural numbers in Epigram as follows

$$\underline{\text{data}} \frac{}{\text{Nat} : \star} \quad \underline{\text{where}} \frac{}{\text{zero} : \text{Nat}} \quad \frac{n : \text{Nat}}{\text{suc } n : \text{Nat}}$$

One way to interpret each **Nat** as a finite type is to write a recursive function which calculates a type of the right size, given an empty type **Zero**, a unit type **One** and disjoint unions  $S + T$

$$\underline{\text{let}} \frac{n : \text{Nat}}{\text{fin } n : \star} \quad \text{fin } n \leftarrow \underline{\text{rec}} \ n$$

$$\text{fin } \text{zero} \Rightarrow \text{Zero}$$

$$\text{fin } (\text{suc } n) \Rightarrow \text{One} + \text{fin } n$$

Another way is to define directly an *inductive family* [16] of finite types:

$$\underline{\text{data}} \frac{n : \text{Nat}}{\text{Fin } n : \star} \quad \underline{\text{where}} \frac{}{\text{fz} : \text{Fin } (\text{suc } n)} \quad \frac{i : \text{Fin } n}{\text{fs } i : \text{Fin } (\text{suc } n)}$$

**Fin**  $n$  gives a coding of the set  $\{0, \dots, n - 1\}$ . **Fin zero** is uninhabited because no constructor targets it; **Fin (suc  $n$ )** has one more element than **Fin  $n$** . Below we tabulate the first few types in this family: we show the ‘ $n$ ’ arguments to **fz** and **fs**—usually left implicit—as subscripts, and write in decimal to save space.

Fin 0	Fin 1	Fin 2	Fin 3	Fin 4	...
	fz <sub>0</sub>	fz <sub>1</sub>	fz <sub>2</sub>	fz <sub>3</sub>	...
		fs <sub>1</sub> fz <sub>0</sub>	fs <sub>2</sub> fz <sub>1</sub>	fs <sub>3</sub> fz <sub>2</sub>	...
			fs <sub>2</sub> (fs <sub>1</sub> fz <sub>0</sub> )	fs <sub>3</sub> (fs <sub>2</sub> fz <sub>1</sub> )	...
				fs <sub>3</sub> (fs <sub>2</sub> (fs <sub>1</sub> fz <sub>0</sub> ))	...
					...

In either presentation, **Nat** acts as a syntax for the finite types which we then equip with a semantics via **fin** or **Fin**. Let us emphasize that **Nat** is an ordinary datatype, and hence operations such as **plus** can be used to equip the finite universe with structure: **Fin (plus  $m$   $n$ )** is isomorphic to **Fin  $m$  + Fin  $n$** . Universe constructions express generic programming for collections of datatypes [6, 18, 21] in terms of ordinary programming with their codes.

The notion of universe brings an extra degree of freedom and of precision to the business of generic programming. By their nature compiler extensions such as Generic Haskell [10] support the extension of generic operations to the whole of Haskell’s type system, but we are free to construct a continuum, from large universes which support basic functionality to small, highly structured universes which support highly advanced functionality. Benke, Dybjer and Jansson provide a good introduction to this continuum in [7]. In fact every family of types, whether inductive like **Fin** or computational like **fin**, yields a universe.

## 1.2 From Finite Types to Regular Tree Types

The finite types are closed under ‘arithmetic’ type constructors such as empty, unit, sum and product. If we also add list formation, we leave the finite universe and acquire the *regular expression types*. We can code these (with respect to an alphabet of size  $n$ ) by the following syntax

$$\begin{array}{c} \text{data } \frac{n : \mathbf{Nat}}{\mathbf{Rex } n : \star} \text{ where } \frac{}{\mathbf{fail}, \mathbf{nil}, \mathbf{dot} : \mathbf{Rex } n} \quad \frac{i : \mathbf{Fin } n}{\mathbf{only } i : \mathbf{Rex } n} \\ \frac{S, T : \mathbf{Rex } n}{S \text{ or } T, S \text{ then } T : \mathbf{Rex } n} \quad \frac{R : \mathbf{Rex } n}{R \text{ star} : \mathbf{Rex } n} \end{array}$$

So for instance we translate the regular expression

$$(A + BC)^* \subseteq \{A, B, C\}^*$$

into the code

$$((\mathbf{only } \mathbf{fz}) \text{ or } ((\mathbf{only } (\mathbf{fs } \mathbf{fz})) \text{ then } (\mathbf{only } (\mathbf{fs } (\mathbf{fs } \mathbf{fz})))))) \mathbf{star} : \mathbf{Rex } 3$$

From each regular expression in the syntax, we may then compute a type which represents the words which match it.

$$\begin{array}{l} \text{let } \frac{R : \mathbf{Rex } n}{\mathbf{Words}_n R : \star} \quad \mathbf{Words}_n R \leftarrow \mathbf{rec } R \\ \mathbf{Words}_n \mathbf{fail} \quad \Rightarrow \mathbf{Zero} \\ \mathbf{Words}_n \mathbf{nil} \quad \Rightarrow \mathbf{One} \\ \mathbf{Words}_n \mathbf{dot} \quad \Rightarrow \mathbf{Fin } n \\ \mathbf{Words}_n (\mathbf{only } i) \quad \Rightarrow \mathbf{Single } i \\ \mathbf{Words}_n (S \text{ or } T) \quad \Rightarrow \mathbf{Words}_n S + \mathbf{Words}_n T \\ \mathbf{Words}_n (S \text{ then } T) \quad \Rightarrow \mathbf{Words}_n S \times \mathbf{Words}_n T \\ \mathbf{Words}_n (R \text{ star}) \quad \Rightarrow \mathbf{List } (\mathbf{Words}_n R) \end{array}$$

Some example **Words** of the expression above would be represented thus:

$$\begin{array}{l} BCA \mapsto [\mathbf{inr } (\mathbf{single } (\mathbf{fs } \mathbf{fz}); \mathbf{single } (\mathbf{fs } (\mathbf{fs } \mathbf{fz}))), \mathbf{inl}(\mathbf{single } \mathbf{fz})] \\ A \mapsto [\mathbf{inl } (\mathbf{single } \mathbf{fz})] \\ \varepsilon \mapsto [] \end{array}$$

This universe, like the finite types, has much algebraic structure to expose, and there is plenty of *ad hoc* work devoted to it, motivated by applications to document structure [19].

Moving just a little further, we can generalise from *lists* to *trees* by replacing **star** with a binding operator  $\mu$  which indicates the least fixpoint of an algebraic type expression. Closing under  $\mu$  gives us the universe of *regular tree types*. In effect, we acquire the first-order fragment of the datatypes found in Standard ML [32]. These include the string-like structures such as the natural numbers,  $\mu N. 1 + N$  and the lists of  $A$ 's,  $\mu L. 1 + A \times L$ , but also branching structures such as binary trees  $\mu T. 1 + T \times T$ . Nesting  $\mu$  yields structures like the finitely

branching trees, whose nodes carry lists of subtrees,  $\mu F. \mu L. 1 + F \times L$ . It is this class of types together with the structures and algorithms they support, which we shall study in this paper.

Of course, there are plenty more universes to visit. Altenkirch and McBride construct the *nested datatypes*, allowing non-uniform type constructors to be defined by recursion [4]. Benke, Dybjer and Jansson construct the *indexed inductive definitions* [7, 17], their motivation being that these structures are effectively those of the Agda system [12] with which they work.

### 1.3 Programming in Epigram

Epigram [29, 30] is a functional programming language with an interactive editor, incrementally typechecking source code containing *sheds*, `[ · · · ]`, whose contents are free text which remains unchecked. It supports programming with inductive families in a pattern matching style, as proposed by Thierry Coquand [13] and first implemented in the Alf system [25].

However, Epigram programs elaborate into a version of Luo’s UTT [23]. This is a more spartan and more clearly defined theory than that of Alf, equipping inductive families only with the induction principles which directly reflect their constructors. In this respect, Epigram more closely resembles its direct ancestor, Lego [24], and also to some extent the Coq system [11]. The design criteria for a good high-level programming language and a good low-level core often pull in opposite directions, hence we separate them. At present, neither Agda nor Coq directly supports pattern matching with inductive families—hand-coding our constructions in these systems would be possible but unnecessarily painful.

Epigram’s data structures are introduced by declaring their applied formation rules and constructors in a natural deduction style. Argument declarations may be omitted where inferrable by unification from their usage—for example, in our declarations of `Fin`’s constructors, `fz` and `fs`, there is no need to declare `n : Nat`. The resemblance between constructor declarations and typing rules is no accident. We intend to encourage the view of an inductive family as a universe capturing a small type system—and that is exactly how we work in this paper.

Epigram functions are given a type signature, also in the natural deduction style, then developed in a *decision tree* structure, shown here by indentation and representing a hierarchical analysis of the task it performs. Each node in a decision tree has a left-hand side which shows the information available, in the form of the *patterns* into which the arguments have been analysed, and a right-hand side which explains how to proceed in that case. The latter may take one of three forms:

- $\Rightarrow t$  the function *returns*  $t$ , an expression of the appropriate type, constructed over the pattern variables on the left;
- $\Leftarrow e$  the function’s analysis is refined by  $e$ , an *eliminator* expression, characterising some scheme of case analysis or recursion, giving rise to a bunch of subnodes with more informative left-hand sides;

|  $w$  the subnodes' left-hand sides are then extended *with* the value of  $w$ , some intermediate computation, in an extra column: this may then be analysed in addition to the function's original arguments.

In effect, Epigram gives a programming notation to some constructions which are more familiar as tactics in proof systems:  $\Rightarrow$  corresponds to Coq's **exact** and | resembles **generalize**;  $\Leftarrow$  is McBride's elimination tactic [28]. McBride and McKinna give a thorough treatment of Epigram elaboration in [30], and begin to explore the flexibility of the  $\Leftarrow$  construct. In this paper, however, we shall need only the standard constructor-guarded recursion operators **rec**  $x$ , which we make explicit, and the standard constructor case analysis operators **case**  $x$ , which we leave implicit whenever their presence is directly inferable from the resulting constructor patterns. In general, we are only explicit about case analysis when its results are empty:

$$\text{let } \frac{x : \text{Fin zero}}{\text{impossible } x : \text{Zero}} \quad \text{impossible } x \Leftarrow \text{case } x$$

Case analyses in Epigram, as in Alf, are constrained by the requirement in each branch that the indices of the scrutinee—**zero** for  $x : \text{Fin zero}$  above—coincide with those of the constructor pattern in question—above,  $(\text{suc } n)$  in both cases. When they concern constructor symbols, these constraints are automatically simplified by first-order unification: impossible cases are dismissed, as above, and the possible cases are simplified. The  $\Leftarrow$  construct thus generalises Alf's dependent constructor matching 'in software'.

Before we start work in earnest, we must own up to the notational liberties we have taken in this paper which the current implementation of Epigram does not support. At present, neither the |  $w$  notation, nor the suppression of obvious  $\Leftarrow \text{case } \dots$  nodes has been implemented: both omissions have simple but verbose workarounds—expanding the programs here would shed more heat than light. More trivially, we work in ASCII rather than  $\LaTeX$  and have only prefix operators thus far—the notation we use here is cosmetically more advanced.

## 2 The Universe of Regular Tree Types

We define the codes for the regular tree types as follows:

$$\begin{aligned} &\text{data } \frac{n : \text{Nat}}{\text{Reg } n : \star} \\ &\text{where } \frac{}{\text{'Z'} : \text{Reg } (\text{suc } n)} \quad \frac{T : \text{Reg } n}{\text{'wk'} T : \text{Reg } (\text{suc } n)} \quad \frac{S : \text{Reg } n \quad T : \text{Reg } (\text{suc } n)}{\text{'let'} S T : \text{Reg } n} \\ &\frac{}{\text{'0'}, \text{'1'} : \text{Reg } n} \quad \frac{S, T : \text{Reg } n}{S \text{'+' } T, S \text{'\times'} T : \text{Reg } n} \quad \frac{F : \text{Reg } (\text{suc } n)}{\text{'\mu'} F : \text{Reg } n} \end{aligned}$$

This is syntax-with-binding in de Bruijn style—the numeric index gives the number of free type variables available in the expression. The 'Z' refers to the



most local variable (de Bruijn index zero), where there is one; the weakening constructor ‘wk’, read backwards, discards the top variable, allowing access to the others. We can thus define an embedding from  $\text{Fin } n$  to the representation of variables in  $\text{Reg } n$

$$\text{let} \frac{X : \text{Fin } n}{\text{‘var’ } X : \text{Reg } n} \quad \begin{array}{l} \text{‘var’ } X \Leftarrow \text{rec } X \\ \text{‘var’ } \text{fz} \Rightarrow \text{‘Z’} \\ \text{‘var’ } (\text{fs } X) \Rightarrow \text{‘wk’ } (\text{‘var’ } X) \end{array}$$

Both ‘ $\mu$ ’ (least fixpoint) and ‘let’ (local definition) bind a variable. The latter clearly introduces redundancy, as does the applicability of ‘wk’ to expressions other than variables. We could have chosen a redundancy free representation, making ‘var’ a constructor and dropping ‘Z’, ‘wk’ and ‘let’. Such a syntax could be equipped with a renaming functor and a substitution monad as in [5] and we should need this equipment *and proofs of its properties* to do our work. Definition and weakening replace just enough of the behavior of substitution for us to avoid this extra effort.

A similar choice presents itself when we come to interpret this syntax. It seems natural to interpret only the *closed* type expressions—the elements of  $\text{Reg zero}$ —substituting whenever we go under a ‘ $\mu$ ’ or ‘let’ binder. Some simple operations, such as our generic equality, become even simpler if we take this choice, but other operations, such as ‘map’, require us to work with properties of substitution. We choose to sidestep substitution in the usual way, interpreting *open* expressions over an environment. We construct our environments carefully to support the way we shall use them: they are *telescopic* [15] in the sense that each new variable is bound to an expression over the previous variables.

$$\text{data} \frac{n : \text{Nat}}{\text{Tel } n : \star} \quad \text{where} \frac{}{\varepsilon : \text{Tel zero}} \quad \frac{ts : \text{Tel } n \quad t : \text{Reg } n}{ts : t : \text{Tel } (\text{suc } n)}$$

We can now interpret every type expression without having to rename de Bruijn indicies at run time to account for the new context or substituting out to a closed expression. Notice that the inductive structure of  $\llbracket - \rrbracket^-$  is *not* the inductive structure of  $\text{Reg}$ —the size of an element is not bounded by the size of its type.

$$\begin{array}{l} \text{data} \frac{\Gamma : \text{Tel } n \quad T : \text{Reg } n}{\llbracket T \rrbracket^\Gamma : \star} \\ \\ \text{where} \frac{t : \llbracket T \rrbracket^\Gamma}{\text{top } t : \llbracket \text{‘Z’} \rrbracket^{\Gamma : T}} \quad \frac{t : \llbracket T \rrbracket^\Gamma}{\text{pop } t : \llbracket \text{‘wk’} \rrbracket^{\Gamma : S}} \quad \frac{t : \llbracket T \rrbracket^{\Gamma : S}}{\text{def } t : \llbracket \text{‘let’} \rrbracket^{\Gamma : S} T} \\ \\ \frac{s : \llbracket S \rrbracket^\Gamma}{\text{inl } s : \llbracket S \text{ ‘+’} \rrbracket^\Gamma} \quad \frac{t : \llbracket T \rrbracket^\Gamma}{\text{inr } t : \llbracket S \text{ ‘+’} \rrbracket^\Gamma} \\ \\ \frac{}{\text{void} : \llbracket \text{‘1’} \rrbracket^\Gamma} \quad \frac{s : \llbracket S \rrbracket^\Gamma \quad t : \llbracket T \rrbracket^\Gamma \quad x : \llbracket F \rrbracket^{\Gamma : (\mu^F F)}}{\text{pair } s \ t : \llbracket S \text{ ‘x’} \rrbracket^\Gamma} \quad \frac{}{\text{in } x : \llbracket \text{‘}\mu \text{’} \rrbracket^{\Gamma : F}} \end{array}$$

The telescopic environments behave as we promised. The rule for ‘Z’ interprets the top type  $T$  over the remaining  $\Gamma$ —but how did it get there? Either from

a ‘let’ or a ‘μ’ extending  $\Gamma$  with a type which is defined over it! The rule for ‘wk’ just pops the environment. Most interesting is the definition of **in**, which uses the environment to expand the fixpoint—let us show how this behaves by coding the natural numbers:

$$\begin{array}{l} \underline{\text{let}} \frac{}{\text{‘Nat’} : \text{Reg } n} \quad \text{‘Nat’} \Rightarrow \text{‘}\mu\text{’} (\text{‘1’} \text{ ‘+’} \text{ ‘Z’}) \\ \\ \underline{\text{let}} \frac{}{\text{ze} : \llbracket \text{‘Nat’} \rrbracket^{\Gamma}} \quad \text{ze} \Rightarrow \text{in (inl void)} \qquad \underline{\text{let}} \frac{n : \llbracket \text{‘Nat’} \rrbracket^{\Gamma}}{\text{su } n : \llbracket \text{‘Nat’} \rrbracket^{\Gamma}} \quad \text{su } n \Rightarrow \text{in (inr (top } n))} \end{array}$$

The **in** constructor places a recursive copy of ‘Nat’ on top of the telescope, which **su** invokes via **top**. We can program with ‘Nat’ quite easily:

$$\underline{\text{let}} \frac{m, n : \llbracket \text{‘Nat’} \rrbracket^{\Gamma}}{\text{plus } m \ n : \llbracket \text{‘Nat’} \rrbracket^{\Gamma}} \quad \begin{array}{l} \text{plus} \quad m \quad n \leftarrow \text{rec } m \\ \text{plus} \quad (\text{in (inl void)}) \quad n \Rightarrow n \\ \text{plus} \quad (\text{in (inr (top } m))) \quad n \Rightarrow \text{su (plus } m \ n) \end{array}$$

Note that the patterns on the left correspond to **ze** and **su**. These cases are exhaustive—all the other constructors target types which conflict with the definition of ‘Nat’, so Epigram dismisses them automatically. The recursive structure of the whole  $\llbracket - \rrbracket^-$  family, thus specializes to that of  $\llbracket \text{‘Nat’} \rrbracket^{\Gamma}$ .

Our recognizably inductive presentation contrasts with Benke, Dybjer and Jansson’s recursive definitions of the *functor* given by each code in a universe, whose least fixpoint is in turn the coded type. Whilst in all of their examples, it is clear to *us* that the computed functors are strictly positive and give rise to inductive types, they make no apparent attempt to explain this to Agda.

The space efficiency of  $\llbracket - \rrbracket^-$  is a serious concern: on the face of it, each data constructor takes an environment and perhaps some type expressions as arguments. Even if sharing is preserved, this is particularly wasteful. Fortunately, as Brady has shown [8, 9], there is no need to duplicate in the data any information extractable from the type indices, so all of the  $\Gamma$ ’s,  $S$ ’s and  $T$ ’s vanish, even from the open representation we need for partial evaluation in the typechecker.

Further, Brady’s work suggests that we can also remove constructor tags where these are determined by indices. In our case, this means that only elements of sums need to be tagged **inl** or **inr**, as each of the other type formers has at most one data former. Hence there is no need for an extra layer of indirection inside each **top**, **pop**, **def** or **in**. There is no reason why our explicit definition and weakening has to lead to a space penalty.

### 3 Deciding Equality

Every regular tree type can be given a decidable equality in a systematic way. In this section, we express that system as a program. Equality as a *Boolean test* has been a standard example of generic programming from PolyP onwards [21].

Benke, Dybjer and Jansson replay this construction in Agda [7] and, moreover, they prove generically that what is being tested really behaves like equality, in that it is reflexive and substitutive. We take a slightly different approach, given that Epigram has a built in equality type [28] which is reflexive by construction, and substitutive by case analysis:

$$\frac{a : A \quad b : B}{a=b : \star} \quad \frac{}{\text{refl} : a=a}$$

Rather than proving a Boolean test correct, we can exploit directly the type of *decisions*, which packs up either a proof or a refutation for a given proposition:

$$\text{data } \frac{P : \star}{\text{Decision } P : \star} \text{ where } \frac{y : P}{\text{yes } y : \text{Decision } P} \quad \frac{n : P \rightarrow \text{Zero}}{\text{no } n : \text{Decision } P}$$

To decide the equality of  $x$  and  $y$ , and know that we have done so, we must show how to compute an inhabitant of `Decision (x=y)`. We can get most of the way by analysing each element and inspecting the results of the recursive calls on corresponding subterms—see 1.

Again, dependent pattern matching ensures that we need only consider elements which have the same type. Fundamentally, all inequalities boil down to the fact that `inl` and `inr` are different: `inl s = inr t` is an empty type, so case analysis leaves no branches. We have left `[]`s for most of the cases where we must show that recursive inequality of subterms breaks equality for the whole terms. Each of these proofs require an auxiliary definition all of which follow the same pattern. The proof for `in` is given below.

$$\text{let } \frac{x, y : \llbracket F \rrbracket^{\Gamma z : (\mu' F)} \quad n : (x=y) \rightarrow \text{Zero} \quad q : (\text{in } x = \text{in } y)}{\text{notEqIn}_{x y} n q : \text{Zero}} \\ \text{notEqIn}_{x x} n \text{ refl} \Rightarrow n \text{ refl}$$

## 4 Type Constructors and Generic Map

We can represent type constructors in our universe by type expressions with parameters, much as one does when one defined a polymorphic data structure in a functional programming language. For example, we can have

$$\text{let } \overline{\text{List}} : \text{Reg} (\text{suc } n) \quad \text{List} \Rightarrow \mu' ('1' '+' ('wk' 'Z' '×' 'Z'))$$

We can then create specific instances of polymorphic structures by capturing the free parameter with `let`—the type of lists of natural numbers would then be coded by `let Nat List`. We can also develop polymorphic operations by working with open type expressions over a nonempty environment:

$\underline{\text{let}} \frac{x, y : \llbracket T \rrbracket^\Gamma}{\text{decEq } x \ y : \text{Decision } (x=y)}$			
<b>decEq</b> $x$ $y$	$\Leftarrow \text{rec } x$		
<b>decEq</b> (def $x$ ) (def $y$ )	<b>decEq</b> $x \ y$		
<b>decEq</b> (def $x$ ) (def $x$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (def $x$ ) (def $y$ )	no $n$	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> (top $x$ ) (top $y$ )	<b>decEq</b> $x \ y$		
<b>decEq</b> (top $x$ ) (top $x$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (top $x$ ) (top $y$ )	no $n$	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> (pop $x$ ) (pop $y$ )	<b>decEq</b> $x \ y$		
<b>decEq</b> (pop $x$ ) (pop $x$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (pop $x$ ) (pop $y$ )	no $n$	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> void void	$\Rightarrow$ yes refl		
<b>decEq</b> (inl $sx$ ) (inl $sy$ )	<b>decEq</b> $sx \ sy$		
<b>decEq</b> (inl $s$ ) (inl $s$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (inl $sx$ ) (inl $sy$ )	no $sn$	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> (inl $sx$ ) (inr $ty$ )	$\Rightarrow$ no ( $\lambda q \Leftarrow \text{case } q$ )		
<b>decEq</b> (inr $tx$ ) (inl $sy$ )	$\Rightarrow$ no ( $\lambda q \Leftarrow \text{case } q$ )		
<b>decEq</b> (inr $tx$ ) (inr $ty$ )	<b>decEq</b> $tx \ ty$		
<b>decEq</b> (inr $t$ ) (inr $t$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (inr $tx$ ) (inr $ty$ )	no $tn$	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> (pair $sx \ tx$ ) (pair $sy \ ty$ )	<b>decEq</b> $sx \ sy$		
<b>decEq</b> (pair $s \ tx$ ) (pair $s \ ty$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (pair $s \ t$ ) (pair $s \ t$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (pair $s \ tx$ ) (pair $s \ ty$ )	yes refl	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> (pair $sx \ tx$ ) (pair $sy \ ty$ )	no $sn$	$\Rightarrow$ no <span style="background-color: yellow;">[]</span>	
<b>decEq</b> (in $x$ ) (in $y$ )	<b>decEq</b> $x \ y$		
<b>decEq</b> (in $x$ ) (in $x$ )	yes refl	$\Rightarrow$ yes refl	
<b>decEq</b> (in $x$ ) (in $y$ )	no $n$	$\Rightarrow$ no ( <b>notEqIn</b> $n$ )	

Fig. 1. Decidable Equality

$\underline{\text{let}} \frac{}{\text{nil} : \llbracket \text{'List'} \rrbracket^\Gamma}$	<b>nil</b>	$\Rightarrow$ in (inl void)
$\underline{\text{let}} \frac{a : \llbracket \text{'Z'} \rrbracket^\Gamma \quad as : \llbracket \text{'List'} \rrbracket^\Gamma}{\text{cons } a \ as : \llbracket \text{'List'} \rrbracket^\Gamma}$	<b>cons</b> $a \ as$	$\Rightarrow$ in (inr (pair (pop $a$ ) (top $as$ )))
$\underline{\text{let}} \frac{as, bs : \llbracket \text{'List'} \rrbracket^\Gamma}{\text{append } as \ bs : \llbracket \text{'List'} \rrbracket^\Gamma}$	<b>append</b>	$bs \Leftarrow \text{rec } as$
<b>append</b> (in (inl void))	$bs \Rightarrow bs$	
<b>append</b> (in (inr (pair (pop $a$ ) (top $as$ ))))	$bs \Rightarrow$ <b>cons</b> $a$ ( <b>append</b> $as \ bs$ )	

Of course, to apply these polymorphic operations in specific cases, one must strip and apply the **def** constructor.

Let us now develop a generic polymorphic operation—functorial mapping. Suppose we have two environments  $\Gamma$  and  $\Delta$  interpreting the free type variables

in an expression  $T$  (**List**, for example). If we can translate between the values in the corresponding types in  $\Gamma$  and  $\Delta$ , then we can map between  $\llbracket T \rrbracket^\Gamma$  and  $\llbracket T \rrbracket^\Delta$ , preserving the structure due to  $T$ , but translating the data corresponding to the free type variables. Here, the fact that we represent the *syntax* of type expressions makes this task easy.

Let us define morphisms between environments and then show how to map them across polymorphic type expressions. We are careful to ensure that we can readily extend a morphism *uniformly* when we go under a binder.

$$\begin{array}{c} \underline{\text{data}} \frac{\Gamma, \Delta : \text{Tel } n}{\text{Morph } \Gamma \ \Delta : \star} \quad \underline{\text{where}} \quad \frac{}{\text{mld} : \text{Morph } \Gamma \ \Gamma} \\ \frac{\phi : \text{Morph } \Gamma \ \Delta \quad f : \llbracket S \rrbracket^\Gamma \rightarrow \llbracket T \rrbracket^\Delta}{\text{mFun } \phi \ f : \text{Morph } (\Gamma : S) (\Delta : T)} \\ \frac{\phi : \text{Morph } \Gamma \ \Delta}{\text{mMap } \phi : \text{Morph } (\Gamma : T) (\Delta : T)} \end{array}$$

We can now write our generic **gMap** operator by structural recursion on data. Each time we go under a binder, we extend the morphism with **mMap**, explaining that the type variable at that point is local. When we reach a variable, we look up the appropriate translation, using **gMap** to interpret **mMap**. In the case of the identity morphism, the environments are known to coincide, so no further traversal is necessary.

$$\underline{\text{let}} \quad \frac{\phi : \text{Morph } \Gamma \ \Delta \quad x : \llbracket T \rrbracket^\Gamma}{\text{gMap } \phi \ x : \llbracket T \rrbracket^\Delta}$$

<b>gMap</b>	$\phi$	$x$	$\Leftarrow \text{rec } x$
<b>gMap</b>	$\phi$	(def $x$ )	$\Rightarrow \text{def } (\text{gMap } (\text{mMap } \phi) \ x)$
<b>gMap</b>	<b>mld</b>	(top $x$ )	$\Rightarrow \text{top } x$
<b>gMap</b>	(mFun $\phi \ f$ )	(top $x$ )	$\Rightarrow \text{top } (f \ x)$
<b>gMap</b>	(mMap $\phi$ )	(top $x$ )	$\Rightarrow \text{top } (\text{gMap } \phi \ x)$
<b>gMap</b>	<b>mld</b>	(pop $x$ )	$\Rightarrow \text{pop } x$
<b>gMap</b>	(mFun $\phi \ f$ )	(pop $x$ )	$\Rightarrow \text{pop } (\text{gMap } \phi \ x)$
<b>gMap</b>	(mMap $\phi$ )	(pop $x$ )	$\Rightarrow \text{pop } (\text{gMap } \phi \ x)$
<b>gMap</b>	$\phi$	(inl $x$ )	$\Rightarrow \text{inl } (\text{gMap } \phi \ x)$
<b>gMap</b>	$\phi$	(inr $x$ )	$\Rightarrow \text{inr } (\text{gMap } \phi \ x)$
<b>gMap</b>	$\phi$	<b>void</b>	$\Rightarrow \text{void}$
<b>gMap</b>	$\phi$	(pair $x \ y$ )	$\Rightarrow \text{pair } (\text{gMap } \phi \ x) \ (\text{gMap } \phi \ y)$
<b>gMap</b>	$\phi$	(in $x$ )	$\Rightarrow \text{in } (\text{gMap } (\text{mMap } \phi) \ x)$

Instantiating **gMap** for our **List** example is straightforward

$$\underline{\text{let}} \quad \frac{f : \llbracket S \rrbracket^\Gamma \rightarrow \llbracket T \rrbracket^\Gamma \quad \text{as} : \llbracket \text{'let' } S \ \text{'List'} \rrbracket^\Gamma}{\text{list } f \ \text{as} : \llbracket \text{'let' } T \ \text{'List'} \rrbracket^\Gamma} \\ \text{list } f \ (\text{def } \text{as}) \Rightarrow \text{def } (\text{gMap } (\text{mFun mld } f) \ \text{as})$$

Is this functorial mapping? An easy induction on  $x$  shows that

$$\mathbf{gMap} \mathbf{mld} x = x$$

but what about composition? Composition may be defined as follows

$$\mathbf{let} \frac{\phi : \mathbf{Morph} \Delta \Theta : \star \quad \psi : \mathbf{Morph} \Gamma \Delta : \star}{\phi \circ \psi : \mathbf{Morph} \Gamma \Theta}$$

$$\begin{aligned} \phi \circ \psi &\leftarrow \mathbf{rec} \phi \\ \mathbf{mld} \quad \circ \psi &\quad \Rightarrow \psi \\ \mathbf{mFun} \phi f \circ \mathbf{mld} &\quad \Rightarrow \mathbf{mFun} \phi f \\ \mathbf{mFun} \phi f \circ \mathbf{mFun} \psi g &\Rightarrow \mathbf{mFun} (\phi \circ \psi) (f \cdot g) \\ \mathbf{mFun} \phi f \circ \mathbf{mMap} \psi &\Rightarrow \mathbf{mFun} (\phi \circ \psi) (f \cdot \mathbf{gMap} \psi) \\ \mathbf{mMap} \phi \circ \mathbf{mld} &\quad \Rightarrow \mathbf{mMap} \phi \\ \mathbf{mMap} \phi \circ \mathbf{mFun} \psi g &\Rightarrow \mathbf{mFun} (\phi \circ \psi) (\mathbf{gMap} \phi \cdot g) \\ \mathbf{mMap} \phi \circ \mathbf{mMap} \psi &\Rightarrow \mathbf{mMap} (\phi \circ \psi) \end{aligned}$$

Another easy induction on  $x$  then shows that

$$\mathbf{gMap} (\phi \circ \psi) x = (\mathbf{gMap} \phi \cdot \mathbf{gMap} \psi) x$$

## 5 The Derivative and the Zipper

Formal differentiation of algebraic expressions was one of the first functional programs ever to be written in pattern matching style and executed on a computer [31]. Thirty-five years later we can run it again, but with a new meaning. As McBride observed [27], differentiating a regular tree type  $T$  with respect to a free variable  $X$  computes the type of *one-hole contexts* for a value from  $X$  in a value from  $T$ . The explanation of the derivative as coding for the linear part of a polymorphic function space between containers can be found in [3]. Here we show how this works out as code:

$$\mathbf{let} \frac{X : \mathbf{Fin} n \quad T : \mathbf{Reg} n}{\partial X T : \mathbf{Reg} n}$$

$$\begin{aligned} \partial X T &\leftarrow \mathbf{rec} T \\ \partial \mathbf{fz} \quad \mathbf{Z} &\Rightarrow \mathbf{1} \\ \partial (\mathbf{fs} X) \quad \mathbf{Z} &\Rightarrow \mathbf{0} \\ \partial \mathbf{fz} \quad (\mathbf{wk} T) &\Rightarrow \mathbf{0} \\ \partial (\mathbf{fs} X) \quad (\mathbf{wk} T) &\Rightarrow \mathbf{wk} (\partial X T) \\ \partial X \quad (\mathbf{let} S T) &\Rightarrow \mathbf{let} S (\partial (\mathbf{fs} X) T) \\ &\quad \mathbf{+} \mathbf{let} S (\partial \mathbf{fz} T) \quad \mathbf{\times} \partial X S \\ \partial X \quad \mathbf{0} &\Rightarrow \mathbf{0} \\ \partial X \quad \mathbf{1} &\Rightarrow \mathbf{0} \\ \partial X \quad (S \mathbf{+} T) &\Rightarrow \partial X S \mathbf{+} \partial X T \\ \partial X \quad (S \mathbf{\times} T) &\Rightarrow \partial X S \mathbf{\times} T \mathbf{+} S \mathbf{\times} \partial X T \\ \partial X \quad (\mu F) &\Rightarrow \mu (\mathbf{1} \mathbf{+} \mathbf{Z} \mathbf{\times} \mathbf{wk} (\mathbf{let} (\mu F) (\partial \mathbf{fz} F))) \\ &\quad \mathbf{\times} \mathbf{let} (\mu F) (\partial (\mathbf{fs} X) F) \end{aligned}$$

Rules we learned from Leibniz take on a direct visual intuition: ‘an  $S \text{ ‘} \vdash \text{’ } T$  with a hole’ is either ‘an  $S$  with a hole’ or ‘a  $T$  with a hole’; ‘an  $S \text{ ‘} \times \text{’ } T$  with a hole’ is either ‘an  $S$  with a hole and a  $T$ ’ or ‘an  $S$  and a  $T$  with a hole’. The *chain rule* for ‘let’ must account for each  $\text{fs } X$  directly in  $T$  as well as each  $X$  sitting inside an  $S$  via a  $\text{fz}$  in  $T$ —this notion of derivative is thus partial on the *free* variables and *total* on the bound variables. McBride added a new rule, inspired by Huet [20]—a one hole context inside an inductively defined container consists of a ‘zipper’ which wraps up the node where the hole is. Let us define a ‘zipper’:

$$\text{let } \frac{F : \text{Reg}(\text{suc } n)}{\text{‘Zipper’ } F : \text{Reg } n}$$

$$\text{‘Zipper’ } F \Rightarrow \mu' ('1' \text{ ‘} \vdash \text{’ } 'Z' \text{ ‘} \times \text{’ } \text{‘wk’} ('let' (\mu' F) (\partial \text{fz } F)))$$

A ‘Zipper’  $F$  is thus a stack of steps, each giving the context for a ‘Z’ inside an  $F$ , and hence a recursive subtree inside a  $\mu' F$ . With this definition, we effectively have that  $\partial X (\mu' F)$  is a node with a hole and a ‘Zipper’  $F$ .

Notice that it is our access to the full syntax of type expressions which enables us to differentiate types with multiple parameters, and hence local definitions and fixpoints. By contrast, programmers in Generic Haskell have no access to these syntactic details. Often this is a convenience, but here it restricts the treatment given by Hinze, Jeurings and Löh [18] to polynomials in one variable.

Let us now show how to plug a ‘var’  $X$  into a  $\partial X T$ , and a  $\mu' F$  into a ‘Zipper’  $F$ . It is not hard to see that these two tasks are mutually recursive. We shall therefore need to dodge the problem that Epigram does not currently support mutually recursive functions. We do this in the obvious way, by turning the mutual definition into the definition of a *family*. First, we define the family of ‘pluggers’ for a type of contexts  $C$  with a hole type  $H$ , yielding output in  $O$ ,

$$\text{data } \frac{C, H, O : \text{Reg } n}{\text{Plugger } C H O : \star} \text{ where } \frac{X : \text{Fin } n \quad T : \text{Reg } n}{X \multimap T : \text{Plugger} (\partial X T) (\text{‘var’ } X) T}$$

$$\frac{F : \text{Reg}(\text{suc } n)}{\odot F : \text{Plugger} (\text{‘Zipper’ } F) (\mu' F) (\mu' F)}$$

and then we explain how to interpret pluggers as operators, by recursion over the context—as long as we consume the context, we are free to ‘change mode’ when we need to. We start like this, by recursion on the task, then case analysis on the plugger:

$$\text{let } \frac{p : \text{Plugger } C H O \quad c : \llbracket C \rrbracket^T \quad h : \llbracket H \rrbracket^T}{c \langle p \rangle h : \llbracket O \rrbracket^T} \quad \begin{array}{l} c \langle p \rangle h \\ c \langle X \multimap T \rangle h \\ c \langle \odot F \rangle h \end{array} \quad \begin{array}{l} \Leftarrow \text{rec } c \\ \boxed{\quad} \\ \boxed{\quad} \end{array}$$

Now we can develop the two branches as if we were writing a mutual definition. We implement  $\langle X \multimap T \rangle$  as follows:

<code>void</code>	$\langle \text{fz} \multimap \text{'Z'} \rangle$	$h$	$\Rightarrow h$
<code>c</code>	$\langle \text{fs } X \multimap \text{'Z'} \rangle$	$h$	$\Leftarrow \text{case } c$
<code>c</code>	$\langle \text{fz} \multimap \text{'wk'} T \rangle$	$h$	$\Leftarrow \text{case } c$
<code>pop c</code>	$\langle \text{fs } X \multimap \text{'wk'} T \rangle$	$\text{pop } h$	$\Rightarrow \text{pop } (c \langle X \multimap T \rangle h)$
<code>inl (def tc)</code>	$\langle X \multimap \text{'let'} S T \rangle$	$h$	$\Rightarrow \text{def } (tc \langle \text{fs } X \multimap T \rangle \text{pop } h)$
<code>inr (pair (def tc) sc)</code>	$\langle X \multimap \text{'let'} S T \rangle$	$h$	$\Rightarrow \text{def } (tc \langle \text{fz} \multimap T \rangle$ $\text{top } (sc \langle X \multimap S \rangle h))$
<code>c</code>	$\langle X \multimap \text{'0'} \rangle$	$h$	$\Leftarrow \text{case } c$
<code>c</code>	$\langle X \multimap \text{'1'} \rangle$	$h$	$\Leftarrow \text{case } c$
<code>inl sc</code>	$\langle X \multimap S \text{'+' } T \rangle$	$h$	$\Rightarrow \text{inl } (sc \langle X \multimap S \rangle h)$
<code>inr tc</code>	$\langle X \multimap S \text{'+' } T \rangle$	$h$	$\Rightarrow \text{inr } (tc \langle X \multimap T \rangle h)$
<code>inl (pair sc t)</code>	$\langle X \multimap S \text{'\times'} T \rangle$	$h$	$\Rightarrow \text{pair } (sc \langle X \multimap S \rangle h) t$
<code>inr (pair s tc)</code>	$\langle X \multimap S \text{'\times'} T \rangle$	$h$	$\Rightarrow \text{pair } s (tc \langle X \multimap T \rangle h)$
<code>pair ff (def fc)</code>	$\langle X \multimap \text{'\mu'} F \rangle$	$h$	$\Rightarrow \text{ff } \langle \odot F \rangle$ $\text{in } (fc \langle \text{fs } X \multimap F \rangle \text{pop } h)$

Meanwhile, ‘zipping out’ iterates ‘plugging in’ tail recursively:

<code>in (inl void)</code>	$\langle \odot F \rangle$	$h \Rightarrow h$
<code>in (inr (pair (top ff) (pop (def fc))))</code>	$\langle \odot F \rangle$	$h \Rightarrow \text{ff } \langle \odot F \rangle$ $\text{in } (fc \langle \text{'Z'} \multimap F \rangle \text{top } h)$

This may look like a complicated definition, but we had some help to write it. The Epigram system calculates all the context types, not us: we just apply case analysis repeatedly on the contexts until the subcontexts appear. The only real choice we must make is whether  $\langle \odot F \rangle$  should read its context as ‘hole-to-root’ or ‘root-to-hole’. Here, following Huet, we choose the former, shrinking the context and growing the subtree as we follow the path.

## 6 Conclusions and Further Work

In this paper, we have constructed the universe of regular tree types, closed under polynomials and least fixpoints; we equipped its syntax with an inductively defined semantics. By dependent pattern matching and structural recursion on data, we implemented a generic decision procedure for equalities on regular types and functorial mapping. We equipped the regular tree types with their differential structure, generalising ‘the zipper’. We have given a tractable coding to these generic tasks without the assistance of any peculiar extensions to Epigram. Ordinary programming suffices to get us this far, and while it is inevitably harder work than *using* tools dedicated to a specific universe, it is undeniably less work than *making* those tools. A dependently typed language allows a flexible approach to programming with universes of many characters, large and small.

Perhaps we should remark on the technology which makes this approach practicable—dependent pattern matching with inductive families of datatypes. We have quietly exploited multiple layers of dependency, with equality types indexed by data from interpretations indexed by telescopes and type expressions



indexed by numbers, for example, and we have not had to lift a finger to push the pattern matching through. This kind of deep dependency takes us well beyond the familiar world of inductive relations indexed by simply typed data, but it is nothing to be frightened of, given suitable tools.

There is a great deal of work yet to do. Whilst we have programmed generically in a small and *ad hoc* universe, we have not developed generic programming for *Epigram*. We should pursue the agenda set by Pfeifer and Rueß [33] to acquire generic programs and proofs for the types we *use*, not just those we *model*.

This is even more vital for dependently typed programming than it is for ‘ordinary’ functional programming because we tend to tailor data structures more closely to the specific properties we need for a given task. We might have sized lists, sorted lists, telescopes or transitive closures where once we just had lists—the extra detail may be just what we need for a particular problem, but it should not come at the expense of rebuilding the list library for each variation. Generic programming can potentially help us in two ways. We can seek to develop operations which work generically for all list-like types, or all concrete syntaxes, or whatever classes of structure we can characterise. We can also seek to roll out structure such as sizing or sortedness across a broad universe of datatypes.

Correspondingly, we need a representation of data structures which directly and compositionally describes inductive families in *Epigram*, in much the way that indexed induction-recursion [17] gives an account of data structures in *Agda*. A promising approach is based on the uniform representation of strictly positive structures as *containers* [1, 2]. These extend readily to dependent structures, and are closed under a fixed grammar of combinators including least and greatest fixpoints. We need the system to automate the quotation of data structures in this grammar and the maps in and out of their container form—much the way Generic Haskell [10] relates each Haskell datatype with the ‘structure types’ over which generic programs actually compute. Subsets of this general grammar then give us codes for smaller universes with more specific structure. If we can standardise our reflection of data structures in this way, then we really shall have reduced generic programming to ordinary programming.

## References

1. Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.
2. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.
3. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride.  $\partial$  for data: derivatives of data structures. *Fundamenta Informaticae*, 65(1,2):1–128, March 2005.
4. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.

5. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.
6. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming—An Introduction. In S. Doaitse Sweierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming, Third International Summer School (AFP '98); Braga, Portugal*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1998.
7. Marcín Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265–269, 2003.
8. Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
9. Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.
10. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löf, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.
11. L'Équipe Coq. The Coq Proof Assistant Reference Manual. <http://pauillac.inria.fr/coq/doc/main.html>, 2001.
12. Catarina Coquand and Thierry Coquand. Structured Type Theory. In *Workshop on Logical Frameworks and Metalanguages*, 1999.
13. Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.
14. Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.
15. Nicolas G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
16. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
17. Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.
18. Ralf Hinze, Johan Jeuring, and Andres Löf. Type-indexed data types. *Science of Computer Programming*, 51:117–151, 2004.
19. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. In *Proceedings of the International Conference on Functional Programming*, 2000.
20. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
21. Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97*, pages 470–482. ACM, 1997.
22. Gottfried Leibniz. Nova methodus pro maximis et minimis, itemque tangentibus, qua nec irrationales quantitates moratur. *Acta eruditorum*, 1684.
23. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
24. Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

25. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.
26. Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.
27. Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at <http://www.dur.ac.uk/c.t.mcbride/diff.ps>, 2001.
28. Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of LNCS. Springer-Verlag, 2002.
29. Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2005+. Revised lecture notes from the International Summer School in Tartu, Estonia.
30. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
31. Fred McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.
32. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, revised edition*. MIT Press, 1997.
33. Holger Pfeifer and Harald Rueß. Polytypic Proof Construction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, , and L. Théry, editors, *Proc. 12th Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 55–72. Springer-Verlag, September 1999.
34. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990.

# On Constructive Existence

Michel Parigot

Équipe “Preuves, Programmes, Systèmes”, CNRS - Université Paris 7,  
Case 7024 - 2, place Jussieu, 75251 Paris Cedex 05  
parigot@pps.jussieu.fr

**Abstract.** Baaz and Fermueller gave in 2003 an original characterization of constructive existence in classical logic [2]. In this note, we give a simple proof of this result based on cut-elimination in sequent calculus. The interest of this proof besides its simplicity is that it allows in particular to generalize the result to other logics enjoying cut-elimination. We also briefly discuss the significance of the characterization itself.

## 1 Introduction

In classical logic, proving an existential statement doesn't ensure having a witness of this existence. The problem of constructive existence is precisely to ensure in one way or another that one can get a witness.

The problem of constructive existence can be stated at the level of the logical framework. Intuitionistic logic is then a possible answer. The intuitionistic logical framework ensures that if we have proved  $\exists xA(x)$ , then we have a witness  $t$  and a proof of  $A(t)$ . But this approach reaches its limit when one works inside theories: one can have  $T \vdash_i \exists xA(x)$  without having a term  $t$  such that  $T \vdash_i A(t)$ .

The problem can also be stated within any given framework, classical logic in particular. The problem is then: is there a general method to *prove* inside the given logical framework that *there exists a term  $t$  such that  $T \vdash A[t]$* , without exhibiting such a term  $t$ ?

M. Baaz and C. Fermueller gave an answer to that problem in [2]. Their characterization of constructive existence uses a syntactic translation of formulas which can be seen as a degenerated notion of realizability.

Consider a first order language  $\mathcal{L}$  without equality and function symbols (this later restriction is only there for simplification and can be removed without any problem). Capital letters  $A, B, C, D$  will always stand for formulas of language  $\mathcal{L}$  and  $T$  for a theory in this language. Symbols  $\vdash_c$  and  $\vdash_i$  denotes respectively provability in classical and intuitionistic logic.

To each  $n$ -ary predicate symbol  $P$  of  $\mathcal{L}$ , we associate a  $n + 1$ -ary predicate symbol  $P^*$  and we define the language  $\mathcal{L}^*$  as  $\{P^* \mid P \in \mathcal{L}\}$ .

The realizability translation is a translation of the formulas of  $\mathcal{L}$  into formulas of  $\mathcal{L}^*$  defined as follows.

**Definition 1.** For each formula  $A$  of the language  $\mathcal{L}$ , a formula  $x \Vdash A$  of the language  $\mathcal{L}^*$  is defined as follows:

$$\begin{aligned} x \Vdash P(t_1, \dots, t_n) &\equiv P^*(t_1, \dots, t_n, x) && \text{for } P \text{ an } n\text{-ary predicate symbol;} \\ x \Vdash C \ c \ D &\equiv (x \Vdash C) \ c \ (x \Vdash D) && \text{for } C, D \text{ formulas and } c \text{ a connective;} \\ x \Vdash Qy \ B &\equiv Qy \ (x \Vdash B) && \text{for } B \text{ a formula and } Q \text{ a quantifier.} \end{aligned}$$

The notion of realizability is degenerated in the sense that the formula  $x \Vdash A$  is essentially  $A$ : it is simply obtained from  $A$  by replacing each atomic formula of the form  $P(t_1, \dots, t_n)$  by  $P^*(t_1, \dots, t_n, x)$ , without doing any transformation at the level of the logical operators..

**Theorem 1.** (Baaz and Fermueller, 2003). *Let  $T$  be a theory and  $A(x)$  a formula in first order logic. Then*

$$T \vdash_c A(t), \text{ for some term } t \quad \text{iff} \quad \forall x(x \Vdash T) \vdash_c \exists x(x \Vdash A(x))$$

Moreover, there is an algorithm allowing to extract from a proof of  $\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$ , a term  $t$  and a proof of  $T \vdash A(t)$ .

M. Baaz and C. Fermueller [2] gave a proof of this result based on properties of resolution. In the following we give a very simple proof based on cut-elimination, which allows in particular to generalize the result to other logics enjoying cut-elimination.

**Remark.** As pointed in [1], if one wants to deal with equality, one has to treat it as a predicate symbol among others, i.e. to translate it in  $=^*$ , and not as a logical symbol.

## 2 Proof of the Theorem

It is obvious that the condition is necessary. From a sequent calculus proof of the sequent  $T \vdash A(t)$ , one obtains a proof of the sequent  $t \Vdash T \vdash t \Vdash A(t)$  by replacing in each sequent of the proof, each formula  $B$  by  $t \Vdash B$  (and possibly renaming some eigenvariables), and thus a proof of the sequent  $\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$ .

In order to prove that the condition is sufficient, one uses the cut-elimination theorem of sequent calculus and considers a cut-free proof  $\pi$  of

$$\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$$

From this proof  $\pi$ , one constructs a term  $t_0$  and a cut-free proof  $\pi_0$  of

$$t_0 \Vdash T \vdash t_0 \Vdash A(t_0)$$

as follows.

First, one can push the contractions on  $\forall x(x \Vdash T)$  and  $\exists x(x \Vdash A(x))$  at the bottom of  $\pi$  and erase weakenings on these formulas. Thus one can assume that  $\pi$  is a proof of a sequent of the form

$$\forall x(x \Vdash T), \dots, \forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x)), \dots, \exists x(x \Vdash A(x))$$

without contractions and weakenings on  $\forall x(x \Vdash T)$  and  $\exists x(x \Vdash A(x))$ .

Note that each subformula of a formula of the form  $t \Vdash B$  is of the form  $t \Vdash C$ , with  $C$  subformula of  $B$ . It follows that  $\forall x(x \Vdash T)$  and  $\exists x(x \Vdash A(x))$  do not occur in axioms and all formulas occurring in  $\pi$  are either formulas of the conclusion or formulas of the form  $t \Vdash C$ , for some term  $t$  and formula  $C$ .

For  $t$  a term occurring in  $\pi$ , we call  $t$ -formulas the formulas of the form  $t \Vdash C$  and the formulas of the conclusion of  $\pi$  "coming from" formulas of the form  $t \Vdash C$ . It follows from the previous remark that all formulas occurring in  $\pi$  are  $t$ -formulas for various  $t$ 's.

**Lemma** (splitting).

Let  $t_1$  a term,  $\delta$  a subproof of  $\pi$  and  $\Gamma_1, \Gamma \vdash \Delta_1, \Delta$  the conclusion of  $\delta$ . Suppose that

$\Gamma_1, \Delta_1$  contain only  $t_1$ -formulas and  
 $\Gamma, \Delta$  contain no  $t_1$ -formulas.

Then there is a proof  $\delta'$  (obtained by erasing parts of  $\delta$  and adding weakenings) such that

either  $\delta'$  is a proof of  $\Gamma_1 \vdash \Delta_1$   
 or  $\delta'$  is a proof of  $\Gamma \vdash \Delta$ .

By an inductive application of this lemma to the conclusion of  $\pi$ , one gets a cut-free proof of a sequent

$$\forall x(x \Vdash T), \dots, \forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x)), \dots, \exists x(x \Vdash A(x))$$

with only  $t_0$ -formulas for a certain term  $t_0$ . By pushing the introduction rules for  $\forall x(x \Vdash T)$  and  $\exists x(x \Vdash A(x))$  to the end of the proof, one gets a cut free proof of

$$t_0 \Vdash T, \dots, t_0 \Vdash T \vdash t_0 \Vdash A(t_0), \dots, t_0 \Vdash A(t_0)$$

and thus a cut-free proof of

$$t_0 \Vdash T \vdash t_0 \Vdash A(t_0).$$

By replacing in each sequent of the proof, each formula  $t_0 \Vdash B$  by  $B$ , one gets a proof of

$$T \vdash A(t_0).$$

**Comment.** The proof rests on the fact that the existential statement  $\exists xA(x)$  is translated into  $\exists x(x \Vdash A(x))$  with the same  $x$  occurring at the two places.

In a classical proof of an existential statement, several witnesses can appear, whose subproofs cannot be unravelled. But in cut-free classical proof of a statement of the form  $\exists x(x \Vdash A(x))$ , the fact that the witnesses occur in the special place created by the realizability translation ensures that the subproofs associated with distinct witnesses cannot mix and thus that the original proof contains a true subproof associated with one of these witnesses.

**Proof of the lemma.**

The result is proved by induction on the length of the proof  $\delta$  of  $\Gamma_1, \Gamma \vdash \Delta_1, \Delta$ . If the last rule of the proof is an axiom, the result is trivial. Otherwise the

result is obtained by an immediate application of the induction hypothesis to the premisses of the last rule. Consider for instance the case where the last rule is an introduction for the connective  $\vee$ .

$$\text{- The last rule is } \frac{\Gamma_1, \Gamma \vdash A, \Delta_1, \Delta}{\Gamma_1, \Gamma \vdash A \vee B, \Delta_1, \Delta}$$

Note that  $A \vee B$  is a  $t_1$ -formula iff  $A$  is a  $t_1$ -formula.

If  $A \vee B$  is a  $t_1$ -formula, then by induction hypothesis, either we have a proof  $\delta'$  of  $\Gamma \vdash \Delta$  and we take  $\delta'$ , or we have a proof  $\delta'$  of  $\Gamma_1 \vdash A, \Delta_1$  and we take the proof

$$\frac{\frac{\delta'}{\Gamma_1 \vdash A, \Delta_1}}{\Gamma_1 \vdash A \vee B, \Delta_1}$$

If  $A \vee B$  is not a  $t_1$ -formula, then by induction hypothesis, either we have a proof  $\delta'$  of  $\Gamma_1 \vdash \Delta_1$  and we take  $\delta'$ , or we have a proof  $\delta'$  of  $\Gamma \vdash A, \Delta$  and we take the proof

$$\frac{\frac{\delta'}{\Gamma \vdash A, \Delta}}{\Gamma \vdash A \vee B, \Delta}$$

$$\text{- The last rule is } \frac{\Gamma_1, \Gamma, A \vdash \Delta_1, \Delta \quad \Sigma_1, \Sigma, B \vdash \Pi_1, \Pi}{\Gamma_1, \Sigma_1, \Gamma, \Sigma, A \vee B \vdash \Delta_1, \Pi_1, \Delta, \Pi}$$

Note that  $A \vee B$  is a  $t_1$ -formula iff both  $A$  and  $B$  are  $t_1$ -formulas.

Assume  $A \vee B$  is a  $t_1$ -formula. We apply the induction hypothesis to the premisses.

If the induction hypothesis gives a proof  $\delta'$  of  $\Gamma \vdash \Delta$  or of  $\Sigma \vdash \Pi$ , we add weakenings to get a proof of  $\Gamma, \Sigma \vdash \Delta, \Pi$ . Otherwise we have proofs  $\delta'_1$  of  $\Gamma_1, A \vdash \Delta_1$  and  $\sigma'_1$  of  $\Sigma_1, B \vdash \Pi_1$  and we take the proof

$$\frac{\frac{\delta'_1}{\Gamma_1, A \vdash \Delta_1} \quad \frac{\sigma'_1}{\Sigma_1, B \vdash \Pi_1}}{\Gamma_1, \Sigma_1, A \vee B \vdash \Delta_1, \Pi_1}$$

Assume  $A \vee B$  is not a  $t_1$ -formula. We apply the induction hypothesis to the premisses. If the induction hypothesis gives a proof  $\delta'$  of  $\Gamma_1 \vdash \Delta_1$  or of  $\Sigma_1 \vdash \Pi_1$ , we add weakenings to get a proof of  $\Gamma_1, \Sigma_1 \vdash \Delta_1, \Pi_1$ . Otherwise we have proofs  $\delta'$  of  $\Gamma, A \vdash \Delta$  and  $\sigma'$  of  $\Sigma, B \vdash \Pi$  and we take the proof

$$\frac{\frac{\delta'}{\Gamma, A \vdash \Delta} \quad \frac{\sigma'}{\Sigma, B \vdash \Pi}}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi}$$

**Remark.** The notion of realizability used to characterize constructive existence cannot be customized by restricting the translation to the predicates occurring in the existential statement. The theory  $T = \{P0 \vee P1, P0 \rightarrow Q0, P1 \rightarrow Q1\}$  provides a counterexample. We have  $T \vdash \exists x Qx$ , but for every term  $t$ ,  $T \not\vdash Qt$ . Consider the modified notion of realizability  $\Vdash_1$ , where the predicate  $P$  is not translated:  $\forall x(x \Vdash_1 T) = \{P0 \vee P1, P0 \rightarrow \forall x(x \Vdash_1 Q0), P1 \rightarrow \forall x(x \Vdash_1 Q1)\}$ . We have  $P0 \vdash \exists x(x \Vdash_1 Qx)$ ,  $P1 \vdash \exists x(x \Vdash_1 Qx)$  and  $\forall x(x \Vdash_1 T) \vdash P0 \vee P1$ . Therefore we have  $\forall x(x \Vdash_1 T) \vdash \exists x(x \Vdash_1 Qx)$ , but as remarked before, there is no  $t$  such that  $T \vdash Qt$ .

### 3 Discussion

Two points deserves to be discussed: the significance of the characterization and the interest of the proof method.

*Interest of the proof method.* The proof we give in this paper for classical first order logic is a generic proof. It works for any logical system having a reasonable cut elimination theorem. It allows in particular to show that the same characterization of constructive existence holds for intuitionistic first order logic.

Note that this characterization of constructive existence in intuitionistic logic is not meaningless. Intuitionistic logic is constructive in the sense that if  $\vdash_i \exists x A(x)$ , then there is a term  $t$  such that  $\vdash_i A(t)$ , which in addition can be extracted from the intuitionistic proof of  $\exists x A(x)$ . But this is no more true over a theory: one can have  $T \vdash_i \exists x A(x)$  without having a term  $t$  such that  $T \vdash_i A(t)$ . The cases where there exists a term  $t$  such that  $T \vdash_i A(t)$  are precisely characterized by the fact that  $\forall x(x \Vdash T) \vdash_i \exists x(x \Vdash A(x))$ .

The proof method can also be extended to characterize constructive existence in high order logics.

*Significance of the characterization.* It is not yet clear whether the characterization of constructive existence given by Baaz and Fermueller is deep or not. But it is worth trying to answer the question. This characterization could possibly lead to a very interesting speed-up result.

The problem can be roughly stated as follows: find a theory  $T$  and a statement  $A$  in first order classical logic such that

- there is a short proof of  $\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$ ;
- there are only long proofs of  $T \vdash A(t)$ .

Our proof seems at first glance to suggest a negative answer, because we have extracted in a trivial way a term  $t$  and a proof of  $T \vdash A(t)$  from a proof of  $\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$ . However this is only due to the fact that we used a cut-free proof of  $\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$ . It is true that a cut-free proof of  $\forall x(x \Vdash T) \vdash \exists x(x \Vdash A(x))$  is essentially a cut-free proof of  $T \vdash A(t)$ , for a certain term  $t$ . But proofs with cuts in the extended language are much richer, not only because of the cuts as such, but also because these cuts allow to work



with formulas of the extended language which are not of the form  $t \Vdash C$ , with  $C$  formula of the original language: one can use arbitrary formulas formed over atomic formulas of that form.

## References

- [1] M. Baaz. *Note on a translation to characterize constructivity*, Proceedings of the Steklov Institute of Mathematics, 242:125,129, 2003.
- [2] M. Baaz, C. Fermueller, *A translation characterizing the constructive content of classical theories*, Logic for Programming and Automated Reasoning (LPAR'2003), LNAI 2850:107-121, 2003.

# Author Index

- Adams, Robin 1  
Altenkirch, Thorsten 252  
Asperti, Andrea 17
- Barthe, Gilles 33  
Benke, Marcin 154  
Berghofer, Stefan 50  
Bertot, Yves 66  
Bove, Ana 82
- Coquand, Thierry 82  
Coupet-Grimal, Solange 115
- Delobel, William 115
- Goguen, Healdene 186  
Grabowski, Adam 138  
Grégoire, Benjamin 66  
Guidi, Ferruccio 17
- Leroy, Xavier 66  
Lindblad, Fredrik 154
- Mamane, Lionel Elie 170  
McBride, Conor 186, 252  
McKinna, James 186  
Meyer, Thomas 201  
Michelbrink, Markus 215  
Miquel, Alexandre 232  
Morris, Peter 252
- Parigot, Michel 268
- Sacerdoti Coen, Claudio 17, 98
- Tarento, Sabrina 33  
Tassi, Enrico 17
- Wolff, Burkhart 201
- Zacchiroli, Stefano 17