Proof Exchange for Theorem Proving — Second International Workshop, PxTP 2012

Manchester, UK
June 30, 2012

Proceedings

Edited by David Pichardie and Tjark Weber

Copyright © 2012 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

Preface

The goal of the PxTP workshop series is to bring together researchers working on proof production from automated theorem provers with potential consumers of proofs. Machine-checkable proofs have been proposed for applications like proof-carrying code and certified compilation, as well as for exchanging knowledge between different automated reasoning systems. For example, interactive theorem provers can import results from otherwise untrusted high-performance solvers, by means of proofs the solvers produce. In such situations, one automated reasoning tool can make use of the results of another, without having to trust that the second tool is sound. It is only necessary to be able to reconstruct a proof that the first tool will accept, in order to import the result without increasing the size of the trusted computing base.

This simple idea of proof exchange for theorem proving becomes quite complicated under the real-world constraints of highly complex and heterogeneous proof producers and proof consumers. For example, even the issue of a standard proof format for a single class of solvers, like SMT solvers, is quite difficult to address, as different solvers use different inference systems. It may be quite challenging, from an engineering and possibly also theoretical point of view, to fit these into a single standard format. Emerging work from several groups proposes standard meta-languages or parametrised formats to achieve flexibility while retaining a universal proof language.

PxTP 2012, the Second International Workshop on Proof Exchange for Theorem Proving, was held as a satellite event of IJCAR on June 30, 2012, in Manchester, UK. The workshop featured invited talks by Robert L. Constable and Stephan Merz, a joint session with the Third Workshop on Practical Aspects of Automated Reasoning (PAAR 2012), and presentations of the five accepted papers that are collected in this volume. We would like to thank the IJCAR organisers, the PxTP program committee, the authors and speakers, and everyone else who contributed to the workshop's success.

David Pichardie and Tjark Weber
Co-chairs, PxTP 2012

Program Committee

Jasmin Blanchette, Technische Universität München, Germany
Pascal Fontaine, University of Nancy, France
John Harrison, Intel Corporation, USA
Leonardo de Moura, Microsoft Research, USA
David Pichardie, Inria Rennes, France
Pierre-Yves Strub, Inria/Microsoft, France
Aaron Stump, The University of Iowa, USA
Geoff Sutcliffe, University of Miami, USA
Laurent Théry, Inria Sophia-Antipolis, France
Allen Van Gelder, University of California at Santa Cruz, USA
Tjark Weber, Uppsala University, Sweden

Contents

Invited Talks

Proof Assistants and the Dynamic Nature of Formal Theories 1
Robert L. Constable

Proofs and Proof Certification in the TLA$^+$ Proof System 16
Stephan Merz

Contributed Papers

LFSC for SMT Proofs: Work in Progress 21
Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen, Harley Eades, Corey Oliver, and Ruoyu Zhang

The $\lambda\Pi$-calculus Modulo as a Universal Proof Language 28
Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant

CoqInE: Translating the Calculus of Inductive Constructions into the $\lambda\Pi$-calculus Modulo 44
Mathieu Boespflug and Guillaume Burel

System Feature Description: Importing Refutations into the GAPT Framework 51
Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, and Bruno Woltzenlogel-Paleo

Walking through the Forest: Fast EUF Proof-Checking Algorithms 58
Frédéric Besson, Pierre-Emmanuel Cornilleau, and Ronan Saillard

Proof Assistants and the Dynamic Nature of Formal Theories*

Robert L. Constable

Cornell University

Abstract

This article shows that *theory exploration* arises naturally from the need to progressively modify *applied formal theories*, especially those underpinning deployed systems that change over time or need to be attack-tolerant. Such formal theories require us to *explore a problem space* with a proof assistant and are naturally dynamic.

The examples in this article are from our on-going decade-long effort to formally synthesize critical components of modern distributed systems. Using the Nuprl proof assistant we created *event logic* and its protocol theories. I also mention the impact over this period of extensions to the constructive type theory implemented by Nuprl. One of them led to our solution of a long standing open problem in constructive logic.

Proof exchange among theorem provers is promising for improving the "super tactics" that provide *domain specific reasoners* for our protocol theories. Both theory exploration and proof exchange illustrate the dynamic nature of applied formal theories built using modern proof assistants. These activities dispel the false impression that formal theories are rigid and brittle artifacts that become less relevant over time in a fast moving field like computer science.

1 Introduction

I believe that one of the major scientific accomplishments of computer science is the creation of software systems that provide indispensable help in performing complex reasoning tasks and in automating abstract intellectual processes. Among such systems are *proof assistants*. They have helped people solve open mathematical problems, build provably correct hardware and software, create attack-tolerant distributed systems, and implement foundational theories of mathematics and computer science. These assistants are also finding a place in advanced technical education [53]. This development fits John McCarthy's early definition of computer science as the subject concerned with the *automation of intellectual processes* [44], and I think it would have fit Turing's, who I regard as the founder of the field. In recognition of this historical Turing celebration year, this article will include some bits of history related to Turing's role in logic and computer science and some bits related to constructive type theory.

These accomplishments with proof assistants are well documented in academic journals and confirmed by industrial uptake. Nevertheless, many otherwise well-informed and influential scientists believe that proof assistants are very difficult to use in these ways and that their solutions are expensive, rigid, and inflexible [21]. I claim that in many cases the formal solutions provided by proof assistants are flexible and easily modified. In some cases that I will discuss, as in creating attack-tolerant networked systems, it is imperative that we automatically modify the supporting formal theories.

*This paper appears also, by agreement, in J. Fleuriot, P. Höfner, A. McIver, and A. Smaill (eds.), Proceedings ATx/WInG 2012.

The fact that we can easily modify, reuse, and replay formal theories might be the main driving force spreading their widening use, just as it is a driving force in the universal spread of text editors, compilers, programming environments, and other commonly used "tools for precise thought". I predict that we will see in the near future dynamic formal theories that are developed and maintained by a community of researchers using proof assistants to support applied formal theories that in turn support deployed software systems "on the fly". We already see communal activity in joint efforts to build mathematical theories or settle open problems, and that is a first step.

1.1 Overview

I explain these claims by referring to proof assistants that implement *constructive type theories* – because these are the proof assistants I know best and because they also connect directly to programming practice. Indeed, they support what has come to be called *correct-by-construction* programming and *formal system evolution* [39, 36, 38]. This is because critical elements of the executing code in deployed systems built this way are synthesized from proofs. To reliably change that code, one must change the proofs and extract new code. Such *applied formal theories* commonly evolve with the application. Moreover, we see another interesting phenomenon, namely the underlying foundational type theories also continue to evolve, perhaps faster than foundational theories that mainly support established mathematics. Those deep theory extensions can be brought to bear on improving the applied theories embedded in them. I will illustrate this and claim that type theories will continue to evolve for quite some time.

I mention briefly just one example of the gradual extension of Constructive Type Theory (CTT), the theory implemented by Nuprl. This example led my colleague Mark Bickford and me to solve an open problem in logic by finding a completeness theorem for intuitionistic first-order logic [14]. We systematically, but gradually over a few years, expanded the use of the intersection type over a family of propositions – at times treating it as a polymorphic universal quantifier because it made program extraction simpler; this led us to the idea of *uniform validity* which became the key to solving the open problem in logic.

Additions to the underlying foundational type theory can be indexed by the years in which they were made, up to the one we used to solve the problem, say from CTT00 to CTT10. The base theory is CTT84, and we are now using CTT12.[1] By and large each theory is an extension of the previous one, although there was a major improvement from CTT02 onwards along with a corresponding change in the proof assistant from Nuprl4 to Nuprl5.

In the special case of proof assistants based on the *LCF tactic mechanism* [29], there is an interesting way to create *domain specific reasoners.* This is simple to describe and remarkably effective in the case of proof assistants with distributed computing capabilities as is the case with Nuprl in support of *Constructive Type Theory* circa 2012 (CTT12). We turn briefly to that example later.

These scientific contributions are enabled by a powerful and ubiquitious information technology that integrates programming languages, interactive provers, automatic provers, model checkers, and databases of formal knowledge. I believe that this integrated technology is rapidly moving toward a point at which it will provide tools for thought able to accelerate scientific research and enrich education in ways computer scientists have only imagined before. The practical integration of computing and logic seen for the first time in this research area has made us aware in detail that there is a *computational reality to logic* which Turning glimpsed in general outline in 1936.

[1] In contrast, I believe that the formalized Tarski-Grothendieck set theory used by the Mizar system [43] to support the *Journal of Formalized Mathematics* has been stable from 1989 until now.

1.2 Historical Background

According to Andrew Hodges [34], one of Alan Turing's deepest insights about computing was that his universal machines would compute with logical formulas as well as numbers. Hodges says: "It put logic, not arithmetic, in the driving seat." Turing believed that this extension would allow computers to participate in the whole range of rational human thought. Turing also understood that we would reason precisely about programs.

Turing also made the notion of a *computable function* a central concept in science. This in turn made the notion of *data types* fundamental and linked them to Russell's conception of a type. Now we see types in the curriculum from the school level through university education. Sets are taught in mathematics, types in programming. In modern type theory, we see sets (in their infinite variety) as a special data type that appeals to mathematicians. So we take a brief historical look at types to explain their use in proof assistants.[2]

1.3 Types in programming and logic

The notion of type is a central organizing concept in the design of programming languages, both to define the *data types* and also to determine the range of significance of procedures and functions.[3] Types feature critically in reasoning about programs as Hoare noted in his fundamental paper on data types [33]. The role of types in programming languages is evident in Algol 60 [59] and its successors such as Pascal and Algol 68 (where types were called *modes*). One of the most notable modern examples is the language ML, standing for MetaLanguage, designed by Milner as an integral part of the *Edinburgh LCF* mechanized *Logic for Computable Functions* [29]. This ML programming language with its remarkably effective *type inference algorithm* and its recursive data types is widely taught in computer science and is central to a large class of proof assistants.

Type theory serves as the logical language of several interactive theorem provers including Agda [12], Coq [19, 4], HOL [28], Isabelle [48], MetaPRL [32], Minlog [3], Nuprl [3, 1], PVS [51], and Twelf [52]. Among these provers, the constructive ones build *correct by construction* software. All of them formalize mathematical theories whose logical correctness is assured to the highest standards of certainty ever achieved. Interactive provers have also been used to solve open mathematical problems, e.g. definitively proving the Four Color Theorem [27]. The accumulation of large libraries of formalized mathematical knowledge using proof assistants has led to the field of *mathematical knowledge management.* Constructive type theories for *constructive and intuitionistic mathematics* serve as practical programming languages, a role imagined forty years ago [47, 30, 15] yet only recently made practical in several settings. The programming language ML also provides the metalanguage for several of the interactive proof assistants in service today, including Agda, Coq, HOL, Isabelle, MetaPRL, Nuprl among others.

2 Automated Reasoning in Constructive Type Theories

The gap between data types in programming languages and those in *foundational type theory* and from there to *constructive type theory* are large. That first gap can be traced to the influence of *Principia Mathematica* and the second to the influence of the mathematician L.E.J. Brouwer

[2] I wrote a longer unpublished paper in 2010 for the 100 year anniversary of *Principia Mathematica* entitled "The Triumph of Types: *Principia Mathematica's* Impact on Computer Science". It describes the influence of type theory on computer science and is available at the celebration web page. These notes rephrase a few examples used there.

[3] This use matches Russell's definition of a type as the *range of significance* of a propositional function.

and his intuitionistic program for mathematics. We look very briefly at these historical gaps to establish the context for the work of modern proof assistants supporting applied formal theories and correct-by-construction programming.

2.1 Comprehensive Mathematical Theories

Near the turn of the last century, Frege [24, 22], Russell [55] and Whitehead [58] strove to *design of a logical foundation for mathematics* free from the known paradoxes and able to support an extremely precise comprehensive treatment of mathematics in an axiomatic logical language. *Principia Mathematica (PM)* was created with those goals in mind, intended to be *safe enough* to avoid paradox and *rich enough* to express all of the concepts of modern pure mathematics of its time in a language its authors regarded as *pure logic*.

For Russell and Whitehead, type theory was not introduced because it was interesting on its own, but because it served as a tool to make logic and mathematics safe. According to *PM* page 37: Type theory "only recommended itself to us in the first instance by its ability to solve certain contradictions. ... it has also a certain consonance with common sense which makes it inherently credible". This common sense idea was captured in Russell's definition of a type in his *Principles of Mathematics, Appendix B The Doctrine of Types* [56] where he says "Every propositional function $\phi(x)$ – so it is contended – has, in addition to its range of truth, a range of significance, i.e. a range within which x must lie if $\phi(x)$ is to be a proposition at all,...." It is interesting that later in computer science, types are used precisely in this sense: to define the *range of significance of functions* in programming languages. It is also interesting that Frege devised a judgment form: $-A$ to indicate that the syntactic term A made sense or was a proposition. The notation $\vdash A$ is the judgment that proposition A is provable. Both of these judgment forms were used in ITT82 and then adopted in CTT84. These forms have proved to be extremely important in practice. In CTT84 they are used to exploit the Turing completeness of the computation system and thus produce efficient extracted code, suitable for practical use.

According to *PM*, statements of *pure mathematics* are inferences of pure logic. All commitments to "reality" (Platonic or physical) such as claims about infinite totalities, the interpretation of implication as a relation, the existence of Euclidean space, etc. were taken as hypotheses. At the time of *PM* it appeared that there would emerge settled agreement about the nature of pure inferences and their axiomatization. That was not to be. Even the notion of pure logic would change.

2.2 Computation in logic and mathematics

While Russell was working on his *theory of types* [55], circa 1907-1908, another conception of logic arose in the writings of L.E.J. Brouwer [31, 57] *circa* 1907, a conception that would depart radically from the vision of Frege and Russell, as they departed from Aristotle. By the early 1930's a mature expression of a new semantics of the logical operators emerged from the work of Brouwer, Heyting, and Kolmogorov; it is now called the *BHK semantics* for intuitionistic versions of formalisms originally developed based on truth-functional semantics. BKH semantics is also called the *propositions-as-types principle*. By 1945 Kleene captured this semantics for first-order logic and Peano arithmetic in his notion of recursive *realizability* based on general recursive functions [35]. By 1968, a formal version of a comprehensive theory of types based on the *propositions as types principle* was implemented in the *Automath* theories of de Bruijn and his colleagues [20]. Unlike Kleene's work, these theories did not take advantage of the

computational interpretation of the logical primitives made possible by BHK, instead treating them formally as rules in the style of *PM*.

This line of research eventually led to two extensional type theories, Intuitionistic Type Theory (ITT82, ITT84) [41, 42] and *Constructive Type Theory (CTT84)* [17] related to ITT82. Subsequently the type theory of the *Calculus of Constructions (CoC)* [18] was designed based on Girard's logic [25, 26]. Over the next few years, CoC moved closer to the ITT82 conception and also built on the CTT82 recursive types [45, 46, 17] to produce the widely used (intensional) type theory *Calculus of Inductive Constructions (CIC)* [19] implemented in the Coq prover [4]. Then an intensional version of ITT82 was defined [49], say ITT90, and implemented much later in the Alf and Agda provers [12]. All three of these efforts, but especially CTT84 and ITT82, were strongly influenced by *Principia* and the work of Bishop [11] presented in his book *Foundations of Constructive Analysis* [11] who set about doing real analysis in the constructive tradition.

Computational content is present in the provable assertions of the constructive type theories, and one of the features of CIC, CTT, and ITT is that implementing the proof machinery makes it possible to *extract* this content and execute it. This discovery has led to a new proof technology for *extracting computational content* from assertions. That technology has proven to be very effective in practice, where it is called correct-by-construction programming. There is a growing literature on this subject with references in survey articles such as [1].

2.3 Effectively Computable, Turing Computable, and Subrecursive Computation Systems

Brouwer's notion of computability was not formal and not axiomatic. It was intuitive and corresponds to what is called *effective computability.* The *Church/Turing Thesis* claims that all effectively computable functions are computable by Turing machines (or any equivalent formalism, e.g. the untyped λ-calculus). There is no corresponding formalism for *Brouwer Computable.* However, I believe that this notion can be captured in intuitionistic logics by leaving a Turing complete computation system for the logic *open-ended* in the sense that new primitive terms and rules of reduction are possible. This method of capturing effective computability may be unique to CTT in the sense that the computation system of CTT is open to being "Brouwer incomplete" as a logic. We have recently added a primitive notion of *general process* to formalize distributed systems whose potentially nonterminating computations are not entirely effective because they depend on asynchronous message passing over a network which can only be modeled faithfully by allowing unpredictable choices by the *environment*, e.g. the internet.

3 A Case Study in Theory Exploration: the Logic of Events

The PRL group's most practical results in formal methods might be the optimization of protocol stacks [39, 36]. The optimization system was adopted by Nortel as a product because it could significantly improve the performance of a distributed system by using automatic compression techniques that we proved were safe. That is, the optimized protocols produced identical results to the original but over five times faster.

Based on our work in this area, we came to see that writing correct distributed protocols was an extremely difficult business. Basically no one can write these protocols completely correctly

"by hand", so they are all gradually debugged in the field. This became an excellent target on which to deploy formal methods, in particular correct-by-construction programming. To start this we needed a specification language for distributed system behavior. We noticed that our collaborators used *message sequence diagrams* as their main informal method of description and reasoning. They did not use temporal logic in any form, they did not use any process algebra, they did not use a logic of actions. So we decided to create a logic that matched their way of thinking as closely as possible. As we spoke it with them, it gradually evolved. The tools of the Nuprl proof assistant and its Logical Programming Environment greatly aided that evolution in ways that I will explain *as an example of theory exploration in action.*

3.1 A sequence of theories

The interested reader could track the evolution of the logic of events from the first publications in 2003 to the latest publications in 2012, [8, 13, 10, 9, 5, 6, 7][4] All stages of this evolution were formalized and applied. They illustrate *theory exploration* in detail. I will summarize the nature of the changes briefly here. In the lecture associated with this article, I will tell some of the stories about why the changes were made and the formal tools used to manage them.

3.2 Summarizing an example of theory exploration and evolution

In *A Logic of Events* [8] circa 2003, we defined together a logic and a computing model. The computing model was based closely on the IO Automata from the book of Nancy Lynch, *Distributed Algorithms*[40]. We called our machines *Message Automata* (MA) because they differed from IOA in the way they composed; we added *frame conditions* to the automata to facilitate reasoning about composition. We also provide addresses which identified the machines and explicit communication channels. We intermixed axioms about events with axioms about the machines, and proved that these axioms were validated by the standard model of computing for the automata. This model was close to that presented by Lynch and also the model used in other textbooks [2]. We used axioms about sending and receiving messages, about state change, and ordering of events, about kinds of events, about how processes represented as Message Automata were composed, and how their computations validated the axioms. Later in *A Causal Logic of Events* [10] circa 2005, we classified the event logic axioms more finely in a progression, starting with axioms for event ordering, then events with values, then events relative to states, and so on.

We learned that the key axiom for many proofs was that *causal ordering* [37] of events is well founded. We could justify this property based on the computation model, and we made that an explicit axiom. We used it to formally prove many important theorems and derive several simple protocols. As we began to specify and prove properties about complex consensus protocols, we discovered that small changes in the protocol would cause disproportionate expansion of our proofs which had to be redone by hand. So we modified the theory to raise the level of abstraction and increase the amount of automation connecting layers of abstraction. The discovery of the notion of an *event class* by 2008 [5] allowed us to postpone reasoning about specific automata until we had the abstract argument finished at this much higher level. The event classes eventually led us to a more abstract programming notation called *event combinators* that we have now implemented directly [7].

By 2010 we could automatically generate proofs from the higher level event classes to the lowest level process code, now implemented in our new programming interface to Nuprl built

[4] This style has the unintended consequence of citing a very large number of our own papers.

by Vincent Rahli called EventML [54]. We were able to create domain specific tactics that would automate in a matter of a day or two the work that previously took us a month or two of hand proofs. By exploiting the distributed implementation of Nuprl, we can now run these domain specific proof tactics in a matter of a few hours by using a cluster of many multi-core processors.

During the period from 2005 to 2009 we learned to modify and extend the entire theory step by step and replay all the proofs. Mark Bickford and Richard Eaton developed mechanisms to replay the proofs, find the broken steps, and modify the tactics and then replay the entire event theory library.

I can only cover one point of this long and rich technical story in this short article. We turn to that one point next. In due course we will publish more "chapters" of the story now that we have deployed our synthesized consensus protocols in a working system called ShadowDB and have started collecting data about its performance [50]. We will also be studying how this system adapts to attack by exploiting *synthetic code diversity*, discussed briefly later in this article.

3.3 Evolving understanding – causal order

We now look briefly at how our understanding of causal order changed as the theories evolved. The same kind of story could be told about minimizing the role of state, about the need for event combinators, and about the need for sub-processes that live only for short periods. All of this would take a monograph. So I will focus on only one such aspect.

Lamport discovered in 1978 that causal order was the right notion of time in distributed systems [37]. Intuitively, an event e comes before e' if e can influence e'. We give the definition below.

To show that causal order is well-founded, we notice that in any execution of a distributed system given by reduction rules, we can conduct a "tour of the events" and number them so that events caused by other events have a higher number. Thus causal order is well founded and if equality of events is decidable, then so is causal order. We show this below.

3.3.1 Signature of EOrder

The signature of these events requires two types, and two partial functions. The types are *discrete*, which means that their defining equalities are decidable. We assume the types are disjoint. We define $\mathbb{D}$ as $\{T : Type \mid \forall x, y : T.\ x = y \text{ in } T \vee \neg\ (x = y \text{ in } T)\}$, the large type of discrete types.

Events with order (EOrder)

E: $\mathbb{D}$
Loc:$\mathbb{D}$
pred?: $E \rightarrow E + Loc$
sender?: $E \rightarrow E + Unit$

The function *pred*? finds the predecessor event of e if e is not the first event at a locus or it returns the *location* if e is the first event. The *sender?(e)* value is the event that sent e if e is a *receive*, otherwise it is a unit. We can define the location of an event by tracing back the predecessors until the value of *pred* belongs to *Loc*. This is a kind of partial function on E.

From *pred? and sender?* we can define these Boolean valued functions:

$$first(e) = \textit{if is_left } (pred?(e)) \textit{ then true else false}$$
$$rcv?(e) = \textit{if is_left } (sender?(e)) \textit{ then true else false}$$

The relation *is_left* applies to any disjoint union type $A+B$ and decides whether an element is in the left or right disjunct. We can "squeeze" considerable information out of the two functions *pred?* and *sender?*. In addition to *first* and *rcv?*, we can define the order relation

$$pred!(e, e') == (\neg\ first(e') \Rightarrow e = pred?(e')) \vee e = sender(e').$$

The transitive closure of *pred*! is Lamport's *causal order* relation denoted $e < e'$. We can prove that it is well-founded and decidable; first we define it.

The nth power of relation R on type T, is defined as

$xR^o y$ iff $x = y \ in \ T$
$xR^n y$ iff $\exists z : T.\ xRz \ \&\ zR^{n-1}y$

The *transitive closure* of R is defined as xR^*y iff $\exists n : \mathbb{N}^{+\cdot}(xR^n y)$.

Causal order is $x \ pred!^* y,\ abbreviated\ x < \ y.$

3.3.2 Axioms for event structures with order (EOrder)

There are only three axioms that constrain event systems with order beyond the typing constraints.

Axiom 1. *If event e emits a signal, then there is an event e′ such that for any event e″ which receives this signal, e″ = e′ or e″ < e′.*

$$\forall e : E.\exists e' : E.\ \forall e'' : E.\ (rcv?(e'') \ \&\ sender?(e'') = e) \Rightarrow (e'' = e' \vee e'' < e)$$

Axiom 2. *The pred? function is injective.*

$$\forall e, e' : E.\ loc(e) = loc(e') \Rightarrow pred?(e) = pred?(e') \Rightarrow e = e'$$

Axiom 3. *The pred! relation is strongly well founded.*

$$\exists f : E \rightarrow \mathbb{N}.\ \forall e, e' : E.pred!(e, e') \Rightarrow f(e) < f(e')$$

To justify f in Axiom 3 we arrange a linear "tour" of the event space in any computation. We imagine that space as a subset of $\mathbb{N} \times \mathbb{N}$ where $\mathbb{N}$ numbers the locations and discrete time. Events happen as we examine them on this tour, so a receive can't happen until we activate the send. Local actions are linearly ordered at each location. Note, we need not make any further assumptions.

We can define the finite list of events before a given event at a location, namely

$before(e) ==$ if first(e) then[]
else $pred?(e)$ append *before* $(pred?\ (e))$

Similarly, we can define the finite tree of all events *causally before* e, namely

$prior(e) ==$ if first(e) then[]
else if $rcv?(e)$
then $< e, prior(sender?(e)), prior(pred?(e)) >$
else $< e, prior(pred?(e)) >$

3.3.3 Properties of events with order

We can prove many interesting facts about events with order. The basis for many of the proofs is induction over causal order. We prove this by first demonstrating that causal order is strongly well founded.

Theorem 3.1. $\exists f : E \to \mathbb{N}. \forall e, e' : E.\ \ e < e' \Rightarrow f(e) < f(e')$

The argument is simple. Let $x \lhd y$ denote $pred!(x, y)$ and let $x \lhd^n y$ denote $pred!^n(x, y)$. Recall that $x \lhd^{n+1} y$ iff $\exists z : E.\ x \lhd z \ \&\ z \lhd^n y$. From Axiom 3 there is function $f_o : E \to \mathbb{N}$ such that $x \lhd y$ implies $f_o(x) < f_o(z)$. By induction on $\mathbb{N}$ we know that $f_o(z) < f_o(y)$. From this we have $f_o(x) < f_o(y)$. So the function f_o satisfies the theorem. The simple picture of the argument is

$$x \lhd z_1 \lhd z_2 \lhd \ldots \lhd z_n \lhd y$$

so

$$f_o(x) < f_o(z_1) < \ldots < f_o(z_n) < f_o(y).$$

We leave the proof of the following induction principle to the reader.

Theorem 3.2. $\forall P : E \to Prop.\ \forall e' : E.\ ((\forall e : E.\ e < e'.\ P(e)) \Rightarrow P(e')) \Rightarrow \forall e : E.\ P(e)$

Using induction we can prove that causal order is decidable.

Theorem 3.3. $\forall e, e' : E.\ e < e' \vee \neg\ (e < e')$

We need the lemma.

Theorem 3.4. $\forall e, e' : E.\ (e \lhd e' \vee \neg\ (e \lhd e'))$

This is trivial from the fact that $pred!(x, y)$ is defined using a decidable disjunction of decidable relations, recall

$$x \lhd y \text{ is } pred!(x, z)$$

and

$$pred!(x, y) = \neg\ first(y) \Rightarrow x = pred?(y) \vee x = sender?(y).$$

The local order given by $pred?$ is a total order. Define $x <_{loc} y$ is $x = pred?(y)$.

Theorem 3.5. $\forall x, y : E.\ \ (x <_{loc} y \vee x = y \vee y <_{loc} x)$

The environment as well as the processes are players in the execution, determining whether protocols converge or not and whether they converge sufficiently fast. However, unlike the processes, the environment is not a Turing computable player. It's behavior was not lawlike. This means that the reduction rule semantics should not be taken as the whole story, and the justification of causal order based on the computations is not the fundamental explanation, only a suggestive approximation. The best explanation are the axioms that define what the environment can contribute, and the induction principle on causal order.

3.4 The formal distributed computing model - circa 2012

Here is a brief overview of our General Process Model circa 2010 [6]. From it we can generate structures we call *event orderings*. We mention key concepts for reasoning about event orderings from these computations. A *system* consists of a set of *components*. Each component has a *location*, an *internal* part, and an *external* part. Locations are just abstract identifiers. There may be more than one component with the same location.

The internal part of a component is a *process*—its program and internal (possibly hidden) state. The external part of a component is its interface with the rest of the system. In this account, the interface will be a list of *messages*, containing either *data* or processes, labeled with the location of the recipient. The "higher order" ability to send a message containing a process allows a system to grow by "forking" or "bootstrapping" new components.

A system executes as follows. At each step, the *environment* may choose and remove a message from the external component. If the chosen message is addressed to a location that is not yet in the system, then a new component is created at that location, using a given *boot-process*, and an empty external part. Each component at the recipient location receives the message as input and computes a pair that contains a process and an external part. The process becomes the next internal part of the component, and the external part is appended to the current external part of the component.

A potentially infinite sequence of steps, starting from a given system and using a given boot-process, is a *run* of that system. From a run of a system we derive an abstraction of its behavior by focusing on the *events* in the run. The events are the pairs, $\langle x, n\rangle$, of a location and a step at which location x gets an input message at step n, i.e. information is transferred. Every event has a location, and there is a natural *causal-ordering* on the set of events, the ordering first considered by Lamport [37]. This allows us to define an *event-ordering*, a structure, $\langle E,\ loc,\ <,\ info\rangle$, in which the causal ordering $<$ is transitive relation on E that is well-founded, and locally-finite (each event has only finitely many predecessors). Also, the events at a given location are totally ordered by $<$. The information, $info(e)$, associated with event e is the message input to $loc(e)$ when the event occurred.

We have found that requirements for distributed systems can be expressed as (higher-order) logical propositions about event-orderings. To illustrate this and motivate the results in the rest of the paper we present a simple example of *consensus* in a group of processes.

Example 1. *A simple consensus protocol: TwoThirds*

Each participating component will be a member of some groups and each group has a name, G. A message $\langle G, i\rangle$ from the environment to component i informs it that it is in group G. The groups have $n = 3f + 1$ members, and they are designed to tolerate f failures. When any component in a group G receives a message $\langle [start], G\rangle$ it starts the consensus protocol whose goal is to decide on values received by the members from *clients*. We assume that once the protocol starts, each process has received a value v_i or has a default non-value.

The simple TwoThirds consensus protocol is this: A process P_i that has a value v_i of type T starts an *election* to choose a value of type T (with a decidable equality) from among those received by members of the group from clients. The elections are identified by natural numbers, el_i initially 0, and incremented by 1, and a Boolean variable $decide_i$ is initially $false$. The function from lists of values, Msg_i to a value is a parameter of the protocol. If the type T of values is Boolean, we can take f to be the majority function.

Begin
Until $decide_i$ **do**:

1. **Increment** el_i; 2. **Broadcast** vote $\langle el_i, v_i \rangle$ to G;
3. **Collect** in list Msg_i $2f + 1$ votes of election el_i;
4. **Choose** $v_i := f(Msg_i)$;
5. **If** Msg_i is unanimous **then** $decide_i := true$

End

We describe protocols like this by classifying the events occurring during execution. In this algorithm there are *Input*, *Vote*, *Collect*, and *Decide* events. The components can recognize events in each of these *event classes* (in this example they could all have distinctive headers), and they can associate information with each event (e.g. $\langle e_i, v_i \rangle$ with Vote, Msg_i with Collect, and $f(Msg_i)$ with Decide). Events in some classes *cause* events with related information content in other classes, e.g. Collect causes a Vote event with value $f(Msg_i)$.

In general, an *event class* X is function on events in an event ordering that *effectively partitions* events into two sets, $E(X)$ and $E - E(X)$, and assigns a value $X(e)$ to events $e \in E(X)$.

Example 2. *Consensus specification*

Let P and D be the classes of events with headers *propose* and *decide*, respectively. Then the *safety specification* of a consensus protocol is the conjunction of two propositions on (extended) event-orderings, called *agreement* (all decision events have the same value) and *responsiveness* (the value decided on must be one of the values proposed):

$$\forall e_1, e_2 : E(D).\ D(e_1) = D(e_2)$$
$$\forall e : E(D).\ \exists e' : E(P).\ e' < e\ \wedge\ D(e) = P(e')$$

It is easy to prove about TwoThirds the safety property that if there are two decide events, say $Decide(e)$ and $Decide(e')$, then $Decide(e) = Decide(e')$. We can also show that if $Decide(e_1) = v$, then there is a prior input event, e_0 such that $Input(e_0) = Decide(e_1)$.

We can prove safety and the following *liveness property* about TwoThirds. We say that activity in the protocol *contracts to a subset* S of exactly $2f + 1$ processes if these processes all vote in election n say at $vt(n)_1, ..., vt(n)_k$ for $k = 2f{+}1$ and collect these votes at $c(n)_1, ..., c(n)_k$, and all vote again in election $n{+}1$ at $vt(n{+}1)_1, ..., vt(n{+}1)_k$, and collect at $c(n{+}1)_1, ..., c(n{+}1)_k$. In this case, these processes in S all decide in round $n + 1$ for the value given by f applied to the collected votes. This is a *liveness* property.

If exactly f processes fail, then the activity of the group G contracts to some S and decides. Likewise if the message traffic is such that f processes are delayed for an election, then the protocol contracts to S and decides. This fact shows that the TwoThirds protocol is *non-blocking*, i.e. from any state of the protocol, there is a path to a decision. We can construct the path to a decision given a set of f processes that we delay.

We also proved safety and liveness of a variant of this protocol that may converge to consensus faster. In this variant, if P_i receives a vote $\langle e, v \rangle$ from a higher election, $e > el_i$, then P_i joins that election by setting $el_i := e; v_i := v$; and then going to step 2.

3.5 Attack Tolerant Distributed Systems

Networked systems will be attacked. One way to protect them is to enable them to adapt to attacks. One way to adapt is to change the protocol, but this is dangerous, especially for the most critical protocols. The responsible system administrators don't ever want to modify the most critical protocols such as consensus. One alternative is to switch to another provably

equivalent protocol, a task familiar to us from the stack optimization work. So we proposed to formally generate a large number of *logically equivalent variants*, store them in an *attack response library*, and switch to one of these variants when an attack is detected or even pro-actively. Each variant uses distinctly different code which a system under attack can install *on-the-fly* to replace compromised components. Each variant is known to be equivalent and correct.

We express threatening features of the environment formally and discriminate among the different types. We can do this in our new GPM model because *the environment is an explicit component about which we can reason.* This capability allows us to study in depth how diversity works to defend systems. We can implement attack scenarios as part of our experimental platform. It is interesting that we can implement a *constructive version* [16] of the famous FLP result [23] that shows how an attacker can keep any deterministic fault-tolerant consensus protocol from achieving consensus.

Synthetic Code Diversity We introduce diversity at all levels of the formal code development (synthesis) process starting at a very high level of abstraction. For example, in the TwoThirds protocol, we can use different functions f, alter the means of collecting Msg_i, synthesize variants of the protocol, alter the data types, etc. We are able to create multiple provably correct versions of protocols at each level of development, e.g. compiling TwoThirds into Java, Erlang, and $F^{\#}$. The higher the starting point for introducing diversity, the more options we can create.

Acknowledgement The ideas reported here are based on the work of the PRL research group, in particular our research on the logic of events by Mark Bickford, Richard Eaton, and Vincent Rahli, and by our collaboration with David Guaspari of ATC-NY corporation. Our formal work was strongly influenced by a long term collaboration with Robbert van Renesse and Ken Birman of the Cornell systems group and the recent work of Nicolas Schiper.

References

[1] S. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley Interscience, New York, 2nd edition, 2004.

[3] 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. G. Schmitt, editors, *Automated Deduction*, volume II. Kluwer, 1998.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[5] M. Bickford. Component specification using event classes. In *Lecture Notes in Computer Science 5582*, pages 140–155. Springer, 2009.

[6] M. Bickford, R. L. Constable, and D. Guaspari. Constructing event structures over a general process model. Department of Computer Science TR1813-23562, Cornell University, Ithaca, NY, 2011.

[7] M. Bickford, R. L. Constable, and V. Rahli. The logic of events, a framework to reason about distributed systems. Technical report.

[8] M. Bickford and R. L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[9] M. Bickford and R. L. Constable. Formal foundations of computer security. In *Formal Logical Methods for System Security and Correctness*, volume 14, pages 29–52, 2008.

[10] M. Bickford and R. L. Constable. A causal logic of events in formalized computational type theory. In *Logical Aspects of Secure Computer Systems, Proceedings of International Summer School Marktoberdorf 2005*, to Appear 2006. Earlier version available as Cornell University Technical Report TR2005-2010, 2005.

[11] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[12] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda – a functional language with dependent types. In C. Urban M. Wenzel S. Berghofer, T. Nipkow, (eds.), *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[13] R. L. Constable. Information-intensive proof technology. In H. Schwichtenberg and K. Spies, editors, *Proof Technology and Computation*. Kluwer, Amsterdam, 2005. 42 pages to appear.

[14] R. L. Constable and M. Bickford. Intuitionistic Completeness of First-Order Logic. Technical Report arXiv:1110.1614v3, Computing and Information Science Technical Reports, Cornell University, 2011.

[15] R. L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[16] R. L. Constable. Effectively nonblocking consensus procedures can execute forever: a constructive version of flp. Technical Report Tech Report 11512, Cornell University, 2008.

[17] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice-Hall, NJ, 1986.

[18] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[19] T. Coquand and C. Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[20] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[21] R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the Association of Computing Machinery*, 22:271–280, 1979.

[22] M. Dummett. *Frege Philosophy of Mathematics*. Harvard University Press, Cambridge, MA, 1991.

[23] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faculty process. *JACM*, 32:374–382, 1985.

[24] G. Frege. Begriffsschrift, a formula language, modeled upon that for arithmetic for pure thought. In J. van Heijenoort, (ed), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 1–82. Harvard University Press, Cambridge, MA, 1967.

[25] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. Springer-Verlag, NY, 1971.

[26] J-Y. Girard. The system F of variable types: Fifteen years later. *Journal of Theoretical Computer Science*, 45:159–192, 1986.

[27] G. Gonthier. Formal proof – the four color theorem. *Notices of the American Math Society*, 55:1382–1392, 2008.

[28] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[29] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[30] C. C. Green. An application of theorem proving to problem solving. In *IJCAI-69—Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, DC, May 1969.

[31] A. Heyting (ed) *L. E. J. Brouwer. Collected Works*, volume 1. North-Holland, Amsterdam, 1975. (see On the foundations of mathematics 11-98.).

[32] J. J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[33] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[34] A. Hodges. Beyond Turing's Machines. *Science*, 336, April 2012.

[35] S. C. Kleene. On the interpretation of intuitionistic number theory. *J. of Symbolic Logic*, 10:109–124, 1945.

[36] C. Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *JFP*, 14(1):21–68, 2004.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, 1978.

[38] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33d ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54. ACM Press, 2006.

[39] X. Liu, C. Kreitz, R. van Renesse, J. J. Hickey, M. Hayden, K. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In D. Kotz and J. Wilkes (eds), *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[40] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[41] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[42] P. Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[43] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and its Applications*, 4:8–14, 2005.

[44] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[45] P. F. Mendler. Recursive types and type constraints in second-order lambda calculus. In D. Gries, editor, *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 30–36. IEEE Computer Society Press, June 1987.

[46] P. F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[47] D. Michie. *Machine Intelligence 3*. American Elsivier, New York, 1968.

[48] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[49] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[50] N. Schiper, V. Rahli, R. van Renesse, and R. L. Constable M. Bickford. Shadowdb: A replicated database on a synthesized consensus core. Technical report, Computer Science Department, Cornell University, Ithaca, NY, 2012.

[51] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[52] F. Pfenning and C. Schürmann. Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632, pages 202–206. Trento, Italy, July 7–10 1999.

[53] B. C. Pierce, C. Casinghino, M. Greenberg, V. Sjberg, and B. Yorgey. *Software Foundations.* Electronic, 2011.

[54] V. Rahli. New tools for domain specific verified programming. Technical Report 1000, Cornell Computing and Information Science, 2012.

[55] B. Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[56] B. Russell. *The Principles of Mathematics.* Cambridge University Press, Cambridge, 1908.

[57] Wa. P. van Stigt. *Brouwer's Intuitionism.* North-Holland, Amsterdam, 1990.

[58] A. N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.

[59] N. Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9:413–432, 1966.

Proofs and Proof Certification in the TLA$^+$ Proof System

Stephan Merz

Inria Nancy Grand-Est & LORIA, Villers-lès-Nancy, France

Abstract

TLA$^+$ is a specification language originally designed for specifying concurrent and distributed systems and their properties. It is based on Zermelo-Fraenkel set theory for modeling data structures and on the linear-time temporal logic TLA for specifying system executions and their properties. The TLA$^+$ proof system (TLAPS) has been designed as an interactive proof assistant for deductively verifying TLA$^+$ specifications. Designed to be independent of any particular theorem prover, it is based on a hierarchical proof language. A proof manager interprets this language and generates proof obligations corresponding to leaf proofs, which can be discharged by different back-end provers. The current release, restricted to non-temporal reasoning, includes Isabelle/TLA$^+$, an encoding of the set theory underlying TLA$^+$ as an object logic in the proof assistant Isabelle, the tableau prover Zenon, and a back-end for SMT solvers.

This article will first give an overview of the overall design of TLAPS and its proof language and then focus on proof certification in TLAPS. Since the use of different back-end provers raises legitimate concerns about the soundness of the integration, we expect back-end provers to produce proofs that can be checked by Isabelle/TLA$^+$, our most trusted back-end, and this is currently implemented for the Zenon back-end. I will review our experiences with proof certification, and to what extent it has contributed to avoiding soundness bugs, and will indicate future work that we intend to carry out in order to improve our confidence in the soundness of TLAPS.

1 The TLA$^+$ proof language

TLA$^+$ [4] is a specification language originally designed for specifying concurrent and distributed systems and their properties. It is based on Zermelo-Fraenkel set theory for modeling data structures and on the linear-time temporal logic TLA for specifying system executions and their properties. The TLA$^+$ proof system TLAPS [2, 3] has been designed as an interactive proof assistant for deductively verifying TLA$^+$ specifications. Designed to be independent of any particular theorem prover, it is based on a hierarchical proof language.

As a simple example of the TLA$^+$ proof language, let us consider the proof of Cantor's theorem. Given the definitions[1]

$$\begin{aligned} \mathit{Range}(f) &\triangleq \{f[x] : x \in \text{DOMAIN } f\} \\ \mathit{Surj}(f, S) &\triangleq S \subseteq \mathit{Range}(f) \end{aligned}$$

of the range of a function and the notion of a function f being surjective for set S, Cantor's theorem can be stated as

$$\text{THEOREM } \mathit{Cantor} \triangleq \forall S : \neg\exists f \in [S \rightarrow \text{SUBSET } S] : \mathit{Surj}(f, \text{SUBSET } S)$$

[1] The application of function f to argument x is written as $f[x]$ in TLA$^+$. The TLA$^+$ expression $[S \rightarrow T]$ denotes the set of functions whose domain is S and such that $f[x] \in T$ for all $x \in S$.

 David Pichardie, Tjark Weber (eds.); PxTP 2012, pp. 16–20

THEOREM $Cantor \triangleq \forall S : \neg\exists f \in [S \rightarrow \text{SUBSET } S] : Surj(f, \text{SUBSET } S)$
PROOF
$\langle 1 \rangle 1.$ ASSUME NEW S,
    $\exists f \in [S \rightarrow \text{SUBSET } S] : Surj(f, \text{SUBSET } S)$
  PROVE FALSE
  $\langle 2 \rangle.$ PICK $f \in [S \rightarrow \text{SUBSET } S] : Surj(f, \text{SUBSET } S)$
    OBVIOUS
  $\langle 2 \rangle 2.$ $\neg Surj(f, \text{SUBSET } S)$
    $\langle 3 \rangle 1.$ DEFINE $D \triangleq \{x \in S : x \notin f[x]\}$
    $\langle 3 \rangle 2.$ $D \in \text{SUBSET } S$
      OBVIOUS
    $\langle 3 \rangle 3.$ $D \notin Range(f)$
      BY DEF $Range$
    $\langle 3 \rangle 4.$ QED
      BY $\langle 3 \rangle 2, \langle 3 \rangle 3$ DEF $Surj$
  $\langle 2 \rangle 3.$ QED
    BY $\langle 2 \rangle 2$
$\langle 1 \rangle 2.$ QED
  BY $\langle 1 \rangle 1$

Figure 1: A hierarchical proof of Cantor's theorem in TLA$^+$.

where SUBSET S denotes the powerset (the set of all subsets) of S. A hierarchical proof of that theorem appears in Figure 1. Proof steps carry labels of the form $\langle d \rangle n$, where d is the depth of the step (which is also indicated by indentation in the presentation of the proof), and n is a freely chosen name—in the example, steps at any given level are just numbered. There are many different ways to write any given proof; in fact, Cantor's theorem could simply be proved by one of the automatic proof backends available in TLAPS. The user chooses how to decompose a proof, starting a new level of proof that ends with a QED step establishing the assertion of the enclosing level. Assertions appearing at the leaves of the proof tree are considered trivial enough to be passed to the automatic provers of TLAPS.

Step $\langle 1 \rangle 1$ reformulates Cantor's theorem as a sequent (TLA$^+$ sequents are written as ASSUME ... PROVE ...) that assumes given some set S for which there exists a surjective function from S to SUBSET S, in order to prove a contradiction. Step $\langle 1 \rangle 2$ deduces the assertion of the theorem from that sequent. (In general, proofs of QED steps are written before the proofs of the steps that precede it.) The level-2 proof starts by picking some such surjective function f and then shows that it cannot in fact be surjective—hence $\langle 2 \rangle 3$ deduces a contradiction. The main argument of the proof appears in the level-3 step, which defines the diagonal set $D \subseteq S$, which can by definition not be in the range of f.

The proof of Cantor's theorem shows some of the main features of the TLA$^+$ proof language. Hierarchical structure is the key to managing complexity. All proof obligations that appear in a proof are independent of one another and can be elaborated and verified in any order. In general, facts and definitions to be used to establish leaf proof steps must be cited explicitly in order to keep the search space for the automatic proof backends manageable.

In our experience, the proof language scales well to large proofs. TLAPS is integrated into

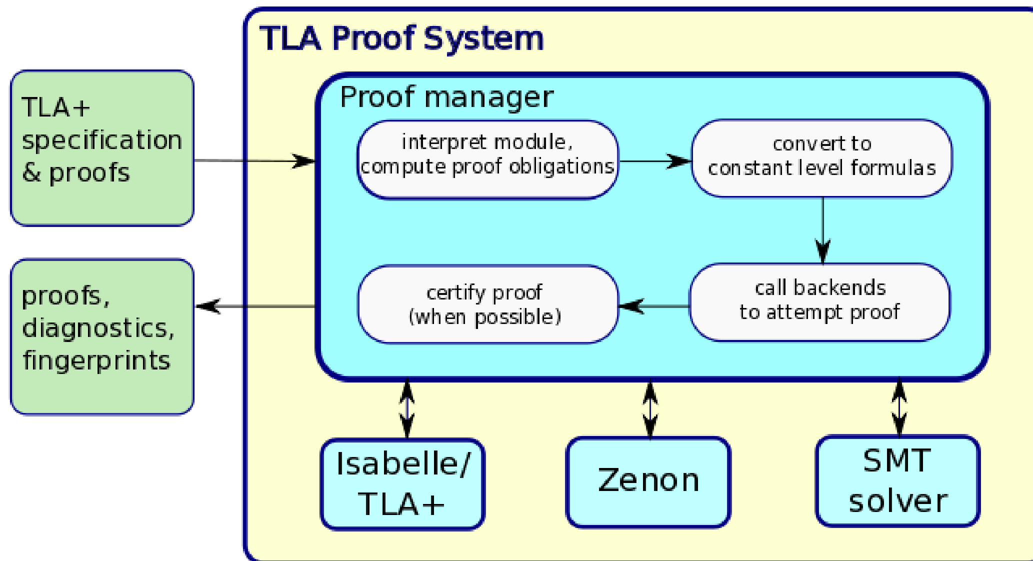


Figure 2: Architecture of TLAPS.

the TLA+ Toolbox, an Eclipse-based IDE that provides features to help reading and writing hierarchical proofs. In particular, proof levels can be hidden or expanded selectively, and the results of proof checking are indicated by color coding. The prover also maintains a database of fingerprints of proof obligations in order to remember what it has already proved previously—even if the proof has been reorganized, or if a step has been deleted and reinserted.

2 Proof checking in TLAPS

Figure 2 shows the basic architecture of the system. The central part of TLAPS is the proof manager, which interprets the proof language and generates proof obligations corresponding to leaf proofs. These can be dispatched to different backend provers. The current release [5], restricted to non-temporal reasoning, includes Isabelle/TLA+, an encoding of the set theory underlying TLA+ as an object logic in the proof assistant Isabelle, the tableau prover Zenon [1], and a backend for SMT solvers.

The proof language is designed to be independent of any particular backend prover. For example, users indicate which definitions to expand and which facts to use, but they cannot tell the backends how to use any given fact—such as for forward or backward chaining, or for rewriting. The set of backends integrated into TLAPS is therefore open-ended: it is enough to implement a translation from (a fragment of) TLA+ to the input language of the backend, and an interface to call the backend with the input corresponding to a proof obligation and retrieve the result. However, the use of different backends, based on different logical theories, raises legitimate concerns about the soundness of their joint use. TLAPS therefore provides a facility for proof certification. Whenever possible, we expect backend provers to produce a proof trace that can be checked by Isabelle/TLA+, our most trusted backend. This is currently implemented for Zenon, which can produce a proof script in the Isar language that is checked by Isabelle.

In our experience, the overhead of proof checking is insignificant—all the more so because

users will check the proof only at the very end, after proof development has finished. The proofs that Zenon produces are detailed and can be checked without any significant proof search. We suspect that checking SMT proofs could become more problematic because SMT solvers can handle much larger formulas that may lead to correspondingly larger proof traces. Techniques for compressing SMT proofs by transforming them in order to avoid redundant steps are then likely to be of interest.

We have yet to find our first Zenon bug. However, certain transformations carried out by the proof manager are critical for soundness, and these should also be checked. The operations of the proof manager fall into two categories. First, it maintains the context of the current proof step, including the symbols that are currently declared and the facts and hypotheses that can be used. The proof language has a block structure with clear scoping rules, and therefore errors in this part can in general be avoided by enforcing a clear program structure.

Secondly, the proof manager is responsible for eliminating references to state variables in (non-temporal) TLA+ formulas. State variables appear in formulas describing state and transition predicates, such as

$$x > c \qquad \text{or} \qquad x > c \wedge x' = x - 1$$

where x is a state variable and c is a constant.[2] Such formulas are evaluated over pairs of states, with the convention that unprimed variable occurrences denote the value of the variable in the first state, and primed occurrences in the second state. If P is a state predicate (i.e., a formula without free primed state variables), the notation P' refers to a copy of P where all state variables have been replaced by their primed versions, so $(x > c)'$ is $x' > c$. Similarly, for any transition predicate A, the notation ENABLED A denotes A with existential quantification over all primed state variables, so[3]

$$\text{ENABLED } (x > c \wedge x' = x - 1) \;\equiv\; \exists x' : x > c \wedge x' = x - 1$$

Since backend provers implement standard first-order logic and do not know about state variables, they handle x and x' as two unrelated first-order variables, and the proof manager must expand notation such as the above. Although this is not difficult as long as definitions are completely expanded, it is easy to get the translation wrong in the presence of unexpanded definitions, in the case of more complicated expressions such as (ENABLED $A)'$, or when expressions appear in argument positions of operators. The soundness of these transformations should therefore be checked by the trusted backend Isabelle/TLA+ of TLAPS, and indeed we have already seen errors in our implementations of these operations. However, this requires a proper definition in Isabelle not only of the set theory underlying TLA+, but also of the notions of state, state variables, and state and transition expressions.

3 Conclusion

TLAPS is based on a hierarchical proof language that results in a clear proof structure and enables independent checking of the different proof obligations that arise in a proof. From a logical point of view, TLA+ proof trees correspond to natural-deduction proofs. The language is interpreted by the proof manager that tracks the context, computes the proof obligations, and translates them to the input formats of backend provers. TLAPS provides an option for retrieving proof traces produced by the backends and having them certified by Isabelle/TLA+,

[2]The category of each symbol appearing in a TLA+ module is declared in the header part of the module.

[3]Semantically, the state predicate ENABLED A holds in a state s if there is some state t such that A holds over the pair (s, t).

a faithful encoding of the TLA$^+$ set theory as an object logic in the proof assistant Isabelle. This is currently implemented for the tableau prover Zenon, and we plan to extend proof certification to the SMT backend. However, in our experience, it is much more likely that a proof is meaningless because of some error in the underlying specification than because of an error in one of the backends. Moreover, the proof manager implements some transformations that are critical for the overall soundness of the system, and these are currently not checked. In the near future, we plan to extend TLAPS for supporting temporal reasoning, and in particular the verification of liveness properties, and this will pose new challenges for proof certification.

Acknowledgement The development of TLAPS is a joint effort carried out at the Joint Microsoft Research-INRIA Centre in Saclay. Kaustuv Chaudhuri, Denis Cousineau, Damien Doligez, Leslie Lamport, and Hernán Vanzetto contributed to different aspects of the work reported here.

References

[1] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *LPAR*, volume 4790 of *LNCS*, pages 151–165, Yerevan, Armenia, 2007. Springer.

[2] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA$^+$ proof system. In J. Giesl and R. Hähnle, editors, *IJCAR*, volume 6173 of *LNCS*, pages 142–148, Edinburgh, UK, 2010. Springer.

[3] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA$^+$ proofs. In D. Giannakopoulou and D. Méry, editors, *Formal Methods*, LNCS, Paris, France, 2012. Springer. To appear.

[4] L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[5] The TLA Proof System. Web page. `http://msr-inria.inria.fr/~doligez/tlaps/`.

LFSC for SMT Proofs: Work in Progress

Aaron Stump, Andrew Reynolds, Cesare Tinelli, Austin Laugesen,
Harley Eades, Corey Oliver and Ruoyu Zhang

The University of Iowa

Abstract

This paper presents work in progress on a new version, for public release, of the Logical Framework with Side Conditions (LFSC), previously proposed as a proof meta-format for SMT solvers and other proof-producing systems. The paper reviews the type-theoretic approach of LFSC, presents a new input syntax which hides the type-theoretic details for better accessibility, and discusses work in progress on formalizing and implementing a revised core language.

1 LFSC and the Challenge of SMT Proofs

The recent widespread adoption of SMT solvers for applications in program analysis, hardware and software verification, and others, has been enabled largely by the SMT community's successful adoption of first the SMT-LIB 1.x and now the SMT-LIB 2.x format [8, 1]. This is truly a laudable achievement of the whole SMT community, which has provided many benefits to applications. The community is also quite interested in doing the same for proofs produced by SMT solvers. This would enable advanced applications which need proofs from SMT solvers (e.g., [4]), and would benefit interactive theorem proving [3]. Having a common proof format and providing suitable incentives (e.g., through competition) could help increase the number of proof-producing solvers. But as the authors and others have argued, devising a common proof system suitable for all SMT solvers is a daunting task, as proof systems tend to mirror solving algorithms, which are quite diverse.

So the community has been considering different proposals for a common meta-format. Besson *et al.* have proposed a meta-format parametrized by solver-specific rules [2]. Their format provides notions of named clauses (via clause ids) and local proofs. This is quite useful, as one can expect to need these features across all particular sets of solver-specific rules. Nevertheless, their format does not provide means for formally defining the semantics of such rules. The ability to give a formal, declarative specification of solver-specific inferences serves as formal documentation, and also enables generic implementation of proof tools like proof checkers or proof translators (to interactive theorem provers, say). We believe Besson *et al.*'s format should be compatible with what we are proposing, though despite some preliminary conversations (particularly with Pascal Fontaine) the details of this remain to be worked out.

In previous papers, we proposed a version of the Edinburgh Logical Framework (LF) [5], extended with support for computational side conditions on inference rules, as a meta-format for SMT [6, 12]. This Logical Framework with Side Conditions (LFSC) combines powerful features of LF (reviewed below) with support for defining side conditions as recursive programs written in a simple first-order functional language. This allows proof systems to be expressed partly declaratively (with inference rules), and partly computationally (with side conditions). The advantage of using computational side conditions is that proofs written in the proof system do not contain any subproof corresponding to the side conditions. Instead, the proof checker executes the side conditions whenever it checks inferences in the proof. This can save both space and checking time. Previous work explored different balances of declarative and computational

$$
\begin{array}{lll}
\textit{formulas}\ \phi & ::= & p \mid \phi_1 \rightarrow \phi_2 \\
\textit{contexts}\ \Gamma & ::= & \cdot \mid \Gamma, \phi
\end{array}
$$

$$
\frac{\phi \in \Gamma}{\Gamma \vdash \phi}\ \textit{Assump} \qquad \frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \rightarrow \phi_2}\ \textit{ImpIntro} \qquad \frac{\Gamma \vdash \phi_1 \rightarrow \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}\ \textit{ImpElim}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot, (p \rightarrow (q \rightarrow r)), q, p \vdash p \rightarrow (q \rightarrow r)} \quad \dfrac{}{\cdot, (p \rightarrow (q \rightarrow r)), q, p \vdash p}}{\cdot, (p \rightarrow (q \rightarrow r)), q, p \vdash q \rightarrow r} \quad \dfrac{}{\cdot, (p \rightarrow (q \rightarrow r)), q, p \vdash q}}{\cdot, (p \rightarrow (q \rightarrow r)), q, p \vdash r}}{\cdot, (p \rightarrow (q \rightarrow r)), q \vdash (p \rightarrow r)}}{\cdot, (p \rightarrow (q \rightarrow r)) \vdash (q \rightarrow (p \rightarrow r))}}{\cdot \vdash (p \rightarrow (q \rightarrow r)) \rightarrow (q \rightarrow (p \rightarrow r))}
$$

Figure 1: Standard mathematical definition of minimal propositional logic, and example proof

rules for the SMT-LIB logics QF_IDL and QF_LRA, and showed how to use type computation for LFSC to compute interpolants [10, 9]. Optimized LFSC proof checking is quite efficient relative to solving time: for QF_UF benchmarks we found a total average overhead of 30% for proof generation and checking (together) over and above solving alone.

We are now focusing on providing a high-quality implementation of LFSC for public release. It is our hope that the availability of such an implementation will be the final step which will enable solver implementors to adopt LFSC as a single, semantics-based meta-format for SMT proofs. In the rest of this short paper, we describe our work in progress towards this vision, which consists of devising a more accessible input syntax (Section 2) and formalizing a new core language which is able to accommodate features like local definitions and implicit arguments (Section 3), which we found to be important in practice but which are often not considered in standard treatments of LF.

We would like to acknowledge also the contributions of Jed McClurg and Cuong Thai to earlier work on this new version of LFSC.

2 A Surface Syntax Based on Grammars and Rules

LF and LFSC are meta-languages based on a dependent type theory. The methodology for using LF/LFSC is to encode object-language constructs using dependently typed data constructors in LF/LFSC. If used directly, this requires unfamiliar type-theoretic notation and conceptualization, a burden we do not wish to impose on SMT solver implementors.

To illustrate these points, let us consider an example proof system in standard mathematical notation. Figure 1 shows the syntax and inference rules of minimal propositional logic. As our proposed syntax for side-condition code has not changed fundamentally, we are deliberately restricting our attention here to a signature without side conditions. For more on side-condition code in LFSC, including examples for a resolution calculus, see [6].

In the LF methodology, a proof system like the one in Figure 1 is encoded as a sequence of type declarations called a signature. The syntactic categories, here *Formula* and *Context*, are encoded as types, whose syntactic constructs are encoded as constructors of those types.

$$
\begin{array}{lcl}
\mathit{formula} & : & \textbf{Type} \\
\mathit{imp} & : & \mathit{formula} \rightarrow \mathit{formula} \rightarrow \mathit{formula} \\
\mathit{holds} & : & \mathit{formula} \rightarrow \textbf{Type} \\
\mathit{ImpIntro} & : & \Pi f_1 : \mathit{formula}.\Pi f_2 : \mathit{formula}.((\mathit{holds}\ f_1) \rightarrow (\mathit{holds}\ f_2)) \rightarrow (\mathit{holds}\ (\mathit{imp}\ f_1\ f_2)) \\
\mathit{ImpElim} & : & \Pi f_1 : \mathit{formula}.\Pi f_2 : \mathit{formula}.(\mathit{holds}\ (\mathit{imp}\ f_1\ f_2)) \rightarrow (\mathit{holds}\ f_1) \rightarrow (\mathit{holds}\ f_2)
\end{array}
$$

$$
\begin{array}{l}
\mathit{ImpIntro}\ (\mathit{imp}\ p\ (\mathit{imp}\ q\ r))\ (\mathit{imp}\ q\ (\mathit{imp}\ p\ r)) \\
\quad \lambda u.\mathit{ImpIntro}\ q\ (\mathit{imp}\ p\ r) \\
\qquad \lambda v.\mathit{ImpIntro}\ p\ r \\
\qquad\quad \lambda w.\mathit{ImpElim}\ q\ r\ (\mathit{ImpElim}\ p\ (\mathit{imp}\ q\ r)\ u\ w)\ v
\end{array}
$$

Figure 2: Minimal propositional logic in LF (core syntax), with encoded example proof (no inferred arguments)

The judgments of the proof system are also encoded as types, with the inference rules as constructors. Object-language variables are represented with meta-language variables, object-language binders with LF's λ-binder, and logical contexts (like Γ in this example) with the LF typing context.

Figure 2 shows the corresponding signature in standard type-theoretic notation for LF. The *Assump* rule does not have a constructor, because assumptions in the object language have been mapped to variables in LF. For the example proof, the λ-bound variables u, v, and w correspond to the assumptions of $p \rightarrow (q \rightarrow r)$, q, and p, respectively. Since inference rules are encoded using term constructors with dependent types (Π types), there is rather a lot of redundant information in that encoded example proof. Implementations of LF like Twelf allow one to omit some of these arguments in some cases, if they can be inferred from other parts of the proof term [7]. In our first implementation of LFSC, we allowed proofs to contain underscores for omitted arguments, which we then sought to reconstruct in a similar way to what is done in Twelf.

We are designing a surface language for LFSC, intended to rely on the more familiar concepts of context-free grammars for describing syntax, and more familiar notation for inference rules, as is done in tools like Ott (and others) [11]. Using this surface language, our sample signature is encoded as in Figure 3. The `ImpIntro` rule uses square brackets and a turnstile for hypothetical reasoning; in this case, to represent the fact that `holds f2` is to be proved under the assumption of `holds f1`. We use root names like `f` for `formula` to avoid specifying types for meta-variables in the rules. Root names are specified when a syntactic category is defined. The example proof term is similar to the one in Figure 2, except that arguments which an analysis determines can be reconstructed can be omitted. Defining this analysis is still work in progress, but in this case, it will determine that of the meta-variables used in the `ImpIntro` and `ImpElim` rules, only `f1` needs a value in order to guarantee that the types computed for terms are fully instantiated (that is, without free meta-variables). The benefits of omitting arguments in dependently typed languages are well known: notice how much more compact the example proof term is without values for the meta-variables `f2` for `ImpIntro` and `f1` and `f2` for `ImpElim`.

For encoding SMT logics, we made use, in our previous proposals, of LF type-checking to implement SMT type-checking of SMT terms contained in proofs. This is done in a standard way by using an indexed type `term T` to represent SMT terms which have (encoded) SMT type `T`. Our new surface syntax also supports indexed syntactic categories. For example, Figure 4

```
  SYNTAX
  formula f ::= imp f1 f2 .

  JUDGMENTS
  (holds f)

  RULES

  [ holds f1 ] |- holds f2
  ----------------------------  ImpIntro
  holds (imp f1 f2) .

  holds (imp f1 f2) ,  holds f1
  ----------------------------  ImpElim
  holds f2 .


ImpIntro (imp p (imp q r))
   u. ImpIntro q
      v. ImpIntro p
        w. ImpElim (ImpElim u w) v
```

Figure 3: LFSC encoding of minimal propositional logic in proposed surface syntax, with example proof

```
  SYNTAX
  sort s ::= arrow s1 s2 | bool .
  term<sort> t ::=
                true<bool>
              | (not t1<bool>)<bool>
              | (impl t1<bool> t2<bool>)<bool>
              | (forall t<s> ^ t<bool>)<bool>
              | (ite t1<bool> t2<s> t3<s>)<s>
              | (eq t1<s> t2<s>)<bool>.
  formula f ::= t<bool>.
```

Figure 4: LFSC encoding (surface syntax) of part of the syntax of SMT terms, indexed by their sorts

gives part of a signature for a minimal SMT theory. This syntax definition indexes the syntactic category `term` by the syntactic category `sort`. Everywhere a meta-variable for terms is used, it must be listed with its sort. Similarly, the definitions of constructors show the resulting indices. For example, since equality is sort-polymorphic, we see that its meta-variables are indexed by (the same) sort `s`, while its resulting sort is indicated as `bool`. This example also demonstrates our surface syntax for binding, namely `^`, in the declaration of `forall`. This notation means that `forall` binds a variable ranging over terms of sort `s`, and then has a body which must be a term of sort `bool`.

As a final example of the new syntax, Figure 5 shows the LFSC encodings of SMT rules for universal quantifiers. The rules are complicated somewhat by the need to express the sort information for terms. The first rule uses the `^` binding notation to indicate a parametric

```
t<s> ^ holds f { t<s> }
-------------------------------- all_intro
holds (forall t<s> ^ f{t<s>} ).

holds (forall t<s> ^ f { t<s> } )
---------------------------------- all_elim
holds f{t<s>} .
```

Figure 5: LFSC encoding (surface syntax) of SMT quantifier rules

judgment in the premise: to apply `all_intro`, we must supply a proof of `holds f{t<s>}` for an arbitrary term `t` of sort `s`. The fact that the term is arbitrary is indicated by the use of `^` in the premise. We also see a notation for expression contexts, using curly braces. Any meta-variable can be used as a context variable, although if that meta-variable is from an indexed syntactic category, the result index must also be shown. Here, we make use of a defined syntactic category of formulas, so that no result sort need be listed for meta-variable `f`. All these different binding concepts (expression contexts, parametric proofs, and hypothetical proofs) are ultimately implemented with just the single λ-binder of LF. But for the surface language, our common informal meta-language tends to distinguish them, and so we follow that approach in our surface language.

3 Improving the Core Language

While the surface language distinguishes syntactic constructs and proof rules, LF elegantly unites them (under encoding) as just term constructors for possibly indexed datatypes. So both for conciseness and to give a semantics for the different features sketched above, we are working to compile signatures from our surface language to a core language for LFSC, which we are specifying formally with the help of the Ott tool [11]. We are addressing a number of issues important for practical use of LF/LFSC for large proofs:

- We have explicit syntax to indicate which arguments to term constructors are implicit and which are explicit.

- We are devising an algorithm to ensure that all higher-order implicit arguments (which our surface language limits to second order) can be automatically reconstructed during proof checking. Other approaches simply delay constraints falling outside the decidable higher-order pattern fragment in the hopes of not needing to solve them.

- In general, we only intend to execute side-condition code on concrete terms. So we are designing the language to help ensure that as much as statically possible, the arguments to side condition programs do not contain meta-variables for omitted implicit arguments. Such meta-variables can be filled in by type checking, but not during execution of side-condition code.

- We have support for global and local definitions, both of which are incorporated into definitional equality. Some other treatments of LF do not handle definitions at all, but these are essential for keeping proof sizes down.

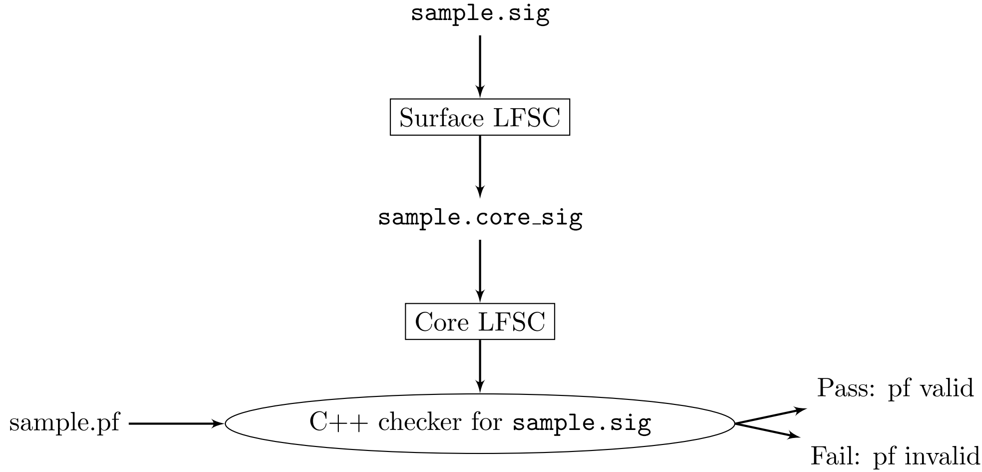


Figure 6: System architecture for new implementation of LFSC

- Since we use definitions, we cannot so easily make use of the so-called "canonical forms" version of LF, where all terms are assumed to be in β-short η-long normal form: applications of defined symbols can be redexes. Instead, we are formulating a system with explicit definitional equality, which can also fill in implicit arguments.
- We retain support for compiling proof rules to efficient low-level (C/C++) code by avoiding explicit substitutions in our formalization. Meta-level substitutions will be carried out directly in code emitted by the compiler, so using explicit substitutions merely obscures what will happen in compiled code.
- We are designing the language for side-condition programs to respect dependent typing of terms. This does not require implementing a full-blown dependently typed programming language, because indices to types can only contain terms which normalize to constructor terms. (In contrast, in dependently typed programming, indices can include terms with recursion.)

While none of these features is terribly complex, designing detailed typing rules that accommodate them all simultaneously has taken time. For example, we are implementing a requirement that all omitted arguments to a term constructor C must be reconstructed by the time we complete the type checking of an application of C. For this, we constrain the syntax of types for term constructors so that implicit arguments are listed first, then explicit arguments and possible invocations of side-condition code, and finally a possibly indexed result type.

4 Status and Conclusion

Figure 6 shows the architecture for the LFSC system we are currently implementing. A sample signature `sample.sig` in the surface syntax discussed above is translated to a core-language signature `sample.core_sig`, which is then compiled to C++ code optimized for checking proofs in that signature. A sample proof `sample.pf` can then be fed to that checker, which reports whether the proof passes (all inferences correct and all side conditions succeed) or fails.

Co-author Corey Oliver has completed the translator for surface-language signature ("Surface LFSC" in Figure 6). Co-author Austin Laugesen has completed implementation of the compiler for side-condition code (part of LFSC core in Figure 6). This compiler translates the LFSC side-condition language to C++, and supports both reference counting or regions for managing memory allocated during execution of side-condition code. A preliminary evaluation confirms the performance benefits for this application of region-based memory management, where all new memory is allocated from a single monolithic region of memory which can then be reclaimed at once in constant time. Co-authors Harley Eades and Ruoyu Zhang have worked with Aaron Stump on the design of the core language, which is almost complete. The first three co-authors have worked to devise the surface syntax, which is complete now. Implementation of the core-language type checker and compiler ("Core LFSC" in Figure 6) in OCaml are just beginning, and should be done Summer 2012. We expect to have an initial public release of the tool in early Fall 2012. We hope that this release will be a decisive step in the quest for a standard meta-format for SMT proofs, and will help the community continue its remarkable success in providing high-performance, usable logic solvers for advanced applications in many fields of Computer Science.

References

[1] C. Barrett, A. Stump, and C. Tinelli. *The SMT-LIB Standard: Version 2.0*, 2010. available from `www.smtlib.org`.

[2] F. Besson, P. Fontaine, and L. Thry. A Flexible Proof Format for SMT: a Proposal. In P. Fontaine and A. Stump, editors, *Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.

[3] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[4] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *Programming Language Design and Implementation (PLDI)*, pages 412–423. ACM, 2010.

[5] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[6] D. Oe, A. Reynolds, and A. Stump. Fast and Flexible Proof Checking for SMT. In B. Dutertre and O. Strichman, editors, *International Workshop on Satisfiability Modulo Theories*, 2009.

[7] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[8] S. Ranise and C. Tinelli. The SMT-LIB Standard, Version 1.2, 2006. Available from the "Documents" section of http://www.smtlib.org.

[9] Andrew Reynolds, Liana Hadarean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic with LFSC. In A. Gupta and D. Kroening, editors, *International Workshop on Satisfiability Modulo Theories*, 2010.

[10] Andrew Reynolds, Cesare Tinelli, and Liana Hadarean. Certified interpolant generation for EUF. In S. Lahiri and S. Seshia, editors, *International Workshop on Satisfiability Modulo Theories*, 2011.

[11] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[12] A. Stump. Proof checking technology for satisfiability modulo theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

The $\lambda\Pi$-calculus Modulo as a Universal Proof Language

Mathieu Boespflug[1], Quentin Carbonneaux[2] and Olivier Hermant[3]

[1] McGill University (mboes@cs.mcgill.ca)
[2] INRIA and ENPC (carbonnq@eleves.enpc.fr)
[3] ISEP (ohermant@isep.fr)

Abstract

The $\lambda\Pi$-calculus forms one of the vertices in Barendregt's λ-cube and has been used as the core language for a number of logical frameworks. Following earlier extensions of natural deduction [14], Cousineau and Dowek [11] generalize the definitional equality of this well studied calculus to an arbitrary congruence generated by rewrite rules, which allows for more faithful encodings of foreign logics. This paper motivates the resulting language, the $\lambda\Pi$-calculus modulo, as a universal proof language, capable of expressing proofs from many other systems without losing their computational properties. We further show how to very simply and efficiently check proofs from this language. We have implemented this scheme in a proof checker called Dedukti.

1 Introduction

The success of formal methods since the pioneering efforts of the teams around LCF and Automath, both as tools of practical importance and as objects of intellectual curiosity, has spawned a bewildering variety of software systems to support them. While the field has developed to maturity in academia and has registered some important successes in the industry, such as in aerospace, mass transit and smart cards to name a few, the full benefit of formal methods in an industrial setting remains largely untapped. We submit that a lack of standards and easy interoperability in the field is one explanation to this state of affairs.

Indeed, the field of formal methods has largely passed on the typical trend of other fields in computer science and engineering as they mature: the shift from systems to data. Standards have emerged for numbers, texts, images, sounds, videos, structured data, etc. These standards are not linked to a particular application, but all the applications comply to these standards, allowing the interoperability of various software, even on heterogeneous networks. When a community reaches a critical mass, standardization is unavoidable to allow exchange and collaborative improvements. As long as the use of proof systems was restricted to a small community, each system could develop its own independent language. But, this is not possible anymore: the success of proof systems and the widening of their audience put the interoperability problem to the fore.

Yet, importing and exporting lumps of proofs from system to system is a thorny issue, because different systems implement different formalisms: some are constructive, some are not, some are predicative, some are not, some are first-order, some are not, etc. Thus, a standard must not only be able to define the proofs themselves, but also to specify the formalism in which they are expressed. To define such a standard, we propose the $\lambda\Pi$-calculus modulo, a simple but very powerful extension of a well-known and well understood proof language for minimal first-order logic that embeds a congruence on types generated by a set of rewrite rules.

1.1 From deep to shallow

Avoiding the quadratic blowup in the number of adapters necessary between n different systems means introducing one common target for all these adapters. All foreign systems read and write a single proof format. One might hope to construct a formalism so powerful that other formalisms form a fragment of this super formalism, but this is infeasible in practice. For one, different formalisms require different but incompatible features, such as predicativity or impredicativity. Moreover, justifying the consistency of such a tower of Babel might not be quite so easy. Finally, the vast majority of proofs simply would not need the full power offered by the formalism, only ever using a small fragment of it.

The alternative is to chose a simple formalism, say first-order logic, and make other formalisms the objects of discourse of this simple formalism. This is known as a *deep embedding*. One will assert appropriate axioms to introduce these objects of discourse, such as axioms characterizing the set of formulas, proof objects, validity judgments on proof objects, etc. This is the approach used by Edinburgh Logical Framework [18], TWELF [24], ISABELLE [23] and others.

Proofs, however, are not interesting merely for the validity of the judgment they establish. It can be desirable to reason about the proofs themselves. Realistic proofs of interesting facts typically contain detours in the reasoning. These detours often make the proof shorter, though they will usually be taken to be equivalent in some precise sense to a canonical derivation that does not contain any detours, or *cuts*. Eliminating cuts in a proof is a procedure, *i.e.* a computation.

When proofs are objects of discourse, it is convenient to be able to reason about them modulo elimination of any cuts. Many formalisms, such as the Calculus of Constructions or the original LF, include a powerful notion of definitional equality that includes equality modulo cuts. Hence, identity of two proof objects can be established not by writing tedious reasoning steps, but rather by an appeal to this rather large definitional equality, written $(\equiv)$, which can be decided by applying the cut elimination procedure. The more powerful the definitional equality, the more often derivations may appeal to it, hence the shorter they become.

Pure Type Systems (PTS), of which the Calculus of Constructions and LF are an instance, internalize definitional equality using the following *conversion* rule:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B} \; A \equiv B$$

where s is some PTS dependent sort. Arguing that two objects are equal can be done in only one derivation step if they are definitionally equal. The trouble here is that the notion of definitional equality is fixed once and for all, no matter the objects of discourse du jour. Hence, a deep embedding of foreign logics treats formulas, judgments and proofs of these logics as second class citizens — one can always define a judgment $(\equiv_L)$ on proofs of a foreign logic L, but given two proofs P, Q in L, one can only justify $P \equiv_L Q$ using a derivation, not the native definitional equality decision procedure.

In short, deep embeddings destroy the computational behaviour of proofs, because proof identity can no longer be justified computationally. Moreover, in general desirable properties such as canonicity of values and the existence of empty types are lost due to the presence of axioms. Many of these axioms uniquely characterize functions on the objects of discourse, for example as a set of equations. If only we could turn these equations into rewrite rules instead, we would be able to directly compute the results of applying these functions, rather than having to systematically prove using derivations that a function application is related to its result.

The λΠ-calculus modulo extends the λΠ-calculus, the core calculus of the original LF,

with the ability to define one's own rewrite rules. The conversion rule above is modified to take definitional equality to be an arbitrary congruence, generated by some set of user defined rewrite rules $\mathcal{R}$. In the λΠ-calculus modulo, we eschew equational axioms and instead turn them into rewrite rules. Intuitively, because the native definitional equality is now extensible, when embedding a foreign logic within the λΠ-calculus modulo, we can tune the native definitional equality to internalize the foreign definitional equality. The computational behaviour of proofs is thus restored. The resulting embeddings are still deep embeddings, but we can introduce rewrite rules in a systematic way that essentially interpret a deep embedding into another *shallow embeddings*.

1.2 Specifying logics

In the λΠ-calculus modulo, a theory is specified by a finite set of typed constants and a finite set of typed rewrite rules. This set of rules defines the computational behavior of the set of constants. Encoding proofs of a logical system is achieved by writing a translation function from proofs of the original logical system to well typed terms of the λΠ-calculus modulo extended by the appropriate constants and rewrite rules. The same translations as the ones defined for LF could be reused since LF is a strict subset of the λΠ-calculus modulo. However, in this case no benefit is drawn from the new computational facilities offered by our framework and the computational properties of proof objects are lost.

We propose to use shallow embeddings as described above. Not only do proofs enjoy smaller representations but consistency of theories can be proved with more generic techniques using, for instance, super-consistency based approaches as described in [15]. Specifying full theories with rewrite rules only, as opposed to traditional axiomatic encodings, allows to show semantic properties of the logical system by checking properties on the set of rewrite rules. For example, if a theory can be expressed in λΠ-calculus modulo with a set of rewrite rules $\mathcal{R}$ and if the rewriting modulo $\mathcal{R} + \beta$ is well-behaved, then the theory is consistent and constructive proofs have the disjunction property and the witness property. Moreover, these shallow embeddings present theories in ways which share a generic notion of cut, rather than a theory specific notion of cut.

1.3 A new proof checker

To support this language, we propose a new proof-checker called Dedukti. A number of proof checkers for similar logical frameworks have been proposed, such as Twelf [24] and LFSC [27], alternately aiming for relative simplicity and correctness, or for focusing more on performance but hopefully also maintaining correctness. LFSC in particular focuses on checking proofs of very large size or many proofs with respect to a fixed signature. Within the design space, we focus on checking small but computationally intensive proofs, through translations of proofs into higher order data and into functional programs. We obtain a very simple implementation to boot, that directly corresponds to the bidirectional typing rules that form its specification.

1.4 Overview

This paper will first present the λΠ-calculus modulo and motivate its use as a target language for many other systems (Section 2). We will then describe informally the rationale behind a purpose built proof-checker we have developped. A companion paper [8] describes in details the encoding of the calculus of constructions inside the λΠ-calculus modulo, as well as practical results in checking Coq's standard library.

$\boxed{\Gamma\ \mathrm{WF}}$ Context Γ is well-formed

$$(\mathit{empty})\ \frac{}{\cdot\ \mathrm{WF}} \qquad (\mathit{decl})\ \frac{\Gamma\ \mathrm{WF} \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x{:}A\ \mathrm{WF}}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$\boxed{\Gamma \vdash M : A}$ Term M is of type A in context Γ

$$(\mathit{sort})\ \frac{\Gamma\ \mathrm{WF}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}} \qquad (\mathit{var})\ \frac{\Gamma\ \mathrm{WF} \qquad x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

$$(\mathit{prod})\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash \Pi x{:}A.\ B : s}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(\mathit{abs})\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(\mathit{app})\ \frac{\Gamma \vdash M : \Pi x{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

$$(\mathit{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A \equiv B$$

Figure 1: Typing rules for the λΠ-calculus (modulo)

2 The λΠ-calculus modulo

2.1 The λΠ-calculus

The λΠ-calculus modulo is a familiar formalism — it is a variation on the λΠ-calculus, which under the guise of many names (LF, λP, etc), has been used with great success to specify other formalisms, from logics to programming languages. The λΠ-calculus is a simple proof language for minimal first-order logic, whose syntax for pre-terms can be given succinctly as follows, provided we have an infinite set X of variable names:

$$M, N, A, B ::= x \mid \lambda x{:}A.\ M \mid \Pi x{:}A.\ B \mid M\ N \mid \mathsf{Type} \mid \mathsf{Kind}$$

The notation $A \to B$ stands for $\Pi x{:}A.\ B$ when x does not appear in B. In the tradition of Pure Type Systems (PTS) [3], we conflate the language of formulas, proofs and sorts (or types, terms and kinds if we are looking from the other side of Curry-Howard lens) into a single language. Within the set of *pre-terms* (or *raw terms*) we distinguish the set of *terms*, which are well-typed.

The type system for the λΠ-calculus, given in figure 1, does enforce a stratification between sublanguages, that of terms, of types and of kinds: terms are typed by a type, types are typed by a kind and kinds are of type Kind.

Notice that in general types can be indexed by terms, giving a *family* of types, and that one can λ-abstract not only over terms but also over types. For these two reasons, β-redexes can appear at the level of types and it is convenient to quotient them by β-equivalence (written (≡)) through the conversion rule. In figure 1, all the contexts are lists of pairs of a variable name and a type:

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x{:}A$$

Figure 1 serves just as well as a presentation of the rules of the λΠ-calculus modulo — the only difference between the two formalisms is in the conversion rule. In the λΠ-calculus, the

congruence is generated by the β-reduction rule alone, while in the λΠ-calculus modulo it is generated by β-reduction *and* the rewrite rules, as presented below.

2.2 Adding rewrite rules: the λΠ-calculus modulo

Well-typed terms live within a context. One can segregate this context into a global shared context and a local context. We will write Σ for this global context, which we call a *signature* and we assume to be well-formed. The assumptions of a signature are *constants*.

A system of rewrite rules is declared over constants of some signature Σ. Each rewrite rule pertains to one constant of the signature. A typed rewrite rule is a rewrite rule paired with a typing context Γ that assigns types to all the free variables appearing in the left hand side of the rule. Both sides of a rewrite rule must be typable with the same type A.

Definition 1 (Rewrite rule)**.** Let Σ be a context. A *rewrite rule* in Σ is a quadruple $l \longrightarrow^{\Gamma,A} r$ composed of a context Γ, and three β-normal terms l, r, A, such that it is *well-typed*:

- the context Σ, Γ is well-formed,
- and $\Sigma, \Gamma \vdash l : A$ and $\Sigma, \Gamma \vdash r : A$ are derivable judgments.

In order to use the rewrite rule, we must rewrite the instances of the left member by instances of the right member. Instances are done through *typed substitutions*.

Definition 2. Let Σ, Γ and Δ be contexts. A substitution binding the variables declared in Γ is said to be of type $\Gamma \rightsquigarrow \Delta$ if for any $x{:}T \in \Gamma$, we can derive $\Sigma, \Delta \vdash \theta x : \theta T$.

Lemma 3. *Let Σ and Δ be contexts, $l \longrightarrow^{\Gamma,T} r$ be a rewrite rule in Σ and θ be a substitution of type $\Gamma \rightsquigarrow \Delta$. Then $\Sigma, \Delta \vdash \theta l : \theta T$ and $\Sigma, \Delta \vdash \theta r : \theta T$ and we say that θl rewrites to θr.*

For Σ a context and $\mathcal{R}$ a set of rewrite rules in Σ, we let $\equiv_{\mathcal{R}}$ be the smallest congruence generated by $\mathcal{R}$. We write $(\longrightarrow)$ and $(\equiv)$ (respectively) for the contextual closure and the smallest congruence generated by β-reduction and $\mathcal{R}$.

2.3 How to encode formalisms in the λΠ-calculus modulo

The claim of this paper is that the λΠ-calculus modulo can serve as a melting pot, capable of expressing all manner of other type theories. This is in particular the case for all functional PTS [11]. By way of example, let us see how to encode polymorphism (*la* System F).

The typing rules of System F are identical to the rules of the λΠ-calculus that are presented in figure 1, except for the rules (*prod*) and (*abs*) that become:

$$(\textit{F-prod})\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi x : A.B : \mathsf{Type}}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(\textit{F-abs})\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : \mathsf{Type} \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

System F is also stratified into kinds (Type is the only kind), types and terms. For brevity and to lift any ambiguities, in the following we write $\star$ (resp. $\Box$) for the constant Type (resp. Kind) of System F. A typical example of term that is typable in System F is the polymorphic identity:

$$\vdash \lambda A{:}\star.\ \lambda x{:}A.\ x : \Pi A{:}\star.\ A \to A$$

It is not possible to write this term in the pure λΠ-calculus.

To encode System F in the $\lambda\Pi$-calculus modulo, we introduce four constants. $U_\star, U_\square$, also called *universe* constants reflect the types $\star$ and $\square$ of System F in the $\lambda\Pi$-calculus modulo. To each universe is associated a *decoding* function ε from System F types and function spaces to the native $\lambda\Pi$-calculus modulo types and function spaces.

$$\begin{array}{ll} U_\square : \mathsf{Type} & \varepsilon_\square : U_\square \to \mathsf{Type} \\ U_\star : \mathsf{Type} & \varepsilon_\star : U_\star \to \mathsf{Type} \end{array}$$

We also introduce a constant $\star : U_\square$ reflecting the fact that $\star : \square$ and the following rewrite rule:

$$\varepsilon_\square\ \star \quad \longrightarrow \quad U_\star$$

Now we introduce a constant that represents a deep embedding of the Π-types of System F as higher order abstract syntax into the $\lambda\Pi$-calculus modulo:

$$\begin{array}{lll} \dot{\Pi}_{\langle\square,\star,\star\rangle} & : & \Pi X{:}U_\square.\ (\varepsilon_\square\ X \to U_\star) \to U_\star \\ \dot{\Pi}_{\langle\star,\star,\star\rangle} & : & \Pi X{:}U_\star.\ (\varepsilon_\star\ X \to U_\star) \to U_\star \end{array}$$

Together with rewrite rules extending the decoding function to these System F types:

$$\begin{array}{lll} \varepsilon_\star\ (\dot{\Pi}_{\langle\square,\star,\star\rangle}\ X\ Y) & \longrightarrow & \Pi x{:}(\varepsilon_\square\ X).\ \varepsilon_\star\ (Y\ x) \\ \varepsilon_\star\ (\dot{\Pi}_{\langle\star,\star,\star\rangle}\ X\ Y) & \longrightarrow & \Pi x{:}(\varepsilon_\star\ X).\ \varepsilon_\star\ (Y\ x) \end{array}$$

Both rewrite rules have type Type (see Definition 1). They contain the free variables X and Y, so they are defined (respectively) in the contexts:

$$\begin{array}{l} X : U_\square,\ Y : (\varepsilon_\square\ X) \longrightarrow U_\star \\ X : U_\star\,,\ Y : (\varepsilon_\star\ X) \longrightarrow U_\star \end{array}$$

Cousineau and Dowek [11] go on to define a translation from terms and types of System F to terms and types using the above constants, and proving conservativity results for this translation. Since our goal here is didactic, we will only give the example of the translation of the previous polymorphic identity, that can now be written in our settings as (keeping the global signature made up of the above constants implicit):

$$\vdash (\lambda A{:}(\varepsilon_\square\ \star).\ \lambda x{:}(\varepsilon_\star\ A).\ x)\ \ :\ \ \varepsilon_\star\ \left(\dot{\Pi}_{\langle\square,\star,\star\rangle}\ \star\ \ \left[\lambda A : (\varepsilon_\square\ \star).\ (\dot{\Pi}_{\langle\star,\star,\star\rangle}\ A\ (\lambda x : (\varepsilon_\star\ A).\ A))\right]\right)$$

which, after simplification of $\varepsilon_\square\ \star$ in $U_\star$ and rewriting of the $\dot{\Pi}_{\langle\square,\star,\star\rangle}$ constant gives the term:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ \ :\ \ \Pi A{:}(\varepsilon_\square\ \star).\ \varepsilon_\star \left(\left[\lambda A{:}U_\star.\ (\dot{\Pi}_{\langle\star,\star,\star\rangle}\ A\ (\lambda x{:}(\varepsilon_\star\ A).\ A))\right]\ A\right)$$

after β-reduction and rewriting of $(\varepsilon_\square\ \star)$, we get:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ \ :\ \ \Pi A{:}U_\star.\ \varepsilon_\star\ \left(\dot{\Pi}_{\langle\star,\star,\star\rangle}\ A\ (\lambda x{:}(\varepsilon_\star\ A).\ A)\right)$$

and after another step of rewriting of the $\dot{\Pi}_{\langle\star,\star,\star\rangle}$:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ \ :\ \ \Pi A{:}U_\star.\ \Pi y{:}(\varepsilon_\star\ A).\ \ (\varepsilon_\star\ ((\lambda x{:}(\varepsilon_\star\ A).\ A)\ y))$$

After β-reduction and α-conversion, we have:

$$\vdash \lambda A{:}U_\star.\ \lambda x{:}(\varepsilon_\star\ A).\ x\ :\ \Pi A{:}U_\star.\ \Pi x{:}(\varepsilon_\star\ A).\ (\varepsilon_\star\ A)$$

a judgment for which one can easily construct a derivation in the λΠ-calculus modulo.

Remark in particular how the use of higher order abstract syntax in the encoding of System F types affords us substitution on those types for free. Also, polymorphism has been introduced in this term, through the constant $\varepsilon_\star$ that behave as a lifting from terms to types and from *deep to shallow* embeddings: this is clearly allowed by its nonempty computational content. Finally, conceivably one could along the same lines encode the λΠ-calculus into the λΠ-calculus modulo through the same mechanisms described above. This encoding would *not* be the identity (we would still have ε and "dotted" constants appearing in types).

3 An abstract type checking algorithm

3.1 Bidirectional type checking

Dedukti's internal representation of terms is domain-free, as introduced in [5], meaning that abstractions are not annotated by the type of the variable that they bind. This information is largely redundant if one adopts a bidirectional typing discipline, a technique going back to [10]. Hence, input terms are smaller at no cost to the complexity or size of the type checker. Bidirectional type systems split the usual type judgments into two forms: checking judgments and synthesis judgments. Dropping the domains on abstractions means that some terms cannot have their type readily inferred — these terms can only be *checked* against a given type, their type cannot be *synthesized*.

Contrary to the usual presentation of λΠ-calculus as a PTS, given in Section 2, the bidirectional type system we present in Figure 2 is syntax directed. As such, the choice of rule to apply at each step of the derivation is entirely determined by the shape of the term that is being typed. In particular, conversion only happens during a phase change between checking mode and synthesis mode. The moment at which this phase change is allowed to occur is guided by the "inputs" to the judgments. As such, the type system presented here is deterministic. One can therefore readily extract an algorithm from these typing rules: checking rules should be read as functions taking a context, a term and type as input and answering true or false; synthesis rules should be read as functions taking a context and term as input and producing a type as output.

3.2 Context-free type checking

One particularly thorny issue when checking dependently typed terms is the treatment of renaming of variables to ensure that contexts remain well-formed and the implementation of substitution. In keeping with the spirit of the de Bruijn criterion [13], we strive to obtain an implementation of a type checker that is as simple and small as possible. In Dedukti, we have done so using a higher-order abstract representation (HOAS) of terms, which allows us to piggy-back substitution over terms on the substitution already provided by the implementation language. In such a setting, abstractions are encoded as functions of the implementation language. They are therefore opaque structures that cannot be directly analyzed. Functions are black boxes whose only interface is that they can be applied. In HOAS, it is therefore necessary to draw the parameter/variable distinction common in presentations of first-order logic — *variables* are always bound and *parameters* (or *eigenvariables*) are variables that are free. As

we move under a binding, such as an abstraction, the bound variable becomes a parameter. In HOAS, this means that we must substitute a piece of data representing a new parameter for the previously bound variable. In this section we do not address the implementation of a concrete algorithm (postponed to Section 4) but elaborate a type system that capture the essence of the way the implementation of Section 4 works.

If in order to proceed to the body of abstractions it is necessary to unfold implementation-level functions by feeding them parameters, we might as well feed more than just a parameter to these functions. We can indeed pass the type of the parameter along with it. In other words, we substitute a *box* — a pair of a term and its type — for bound variable as we move between bindings. We arrive in this way at a *context-free* type system, so-called because we no longer need to carry around a typing context in judgments, provided all free variables (parameters) in terms are always boxed.

3.3 Putting it all together

The resulting type system, bidirectional and context-free, is given in Figure 2. A straightforward mapping of these rules into a purely functional program is discussed in Section 4. The syntax over which these rules are defined is given below.

In its essence, a proof script consists of a sequence of non-recursive declarations of the form $x : A$. Later declarations have earlier declarations in scope. Proof scripts are type checked following the order of the sequence. Given a term M, checking that this term has some given type A will first assume that A has already been type checked, just as the types of constants appearing in A may be assumed to already be checked when checking A. Therefore adding premises ensuring that contexts are well-formed to the leaf rules and mixing premises about types in the deduction rules for term typing as is done in the standard presentation of Section 2 does not accurately reflect how type checking works in practice. In the rules of Figure 2, the type A in a judgment $\vdash M \Leftarrow A$ is assumed to be a proper type or kind.

For the needs of type checking we can therefore use two distinct representations: one for the *subject* of a judgment (to the left of $\Rightarrow$ or $\Leftarrow$), underlined,

$$\underline{M}, \underline{N}, \underline{A}, \underline{B} ::= x \mid [y : \overline{A}] \mid \lambda x.\ \underline{M} \mid \Pi x{:}\underline{A}.\ \underline{B} \mid \underline{M}\ \underline{N} \mid \mathsf{Type}$$

and one for the *classifier* of a judgment (to the right of $\Rightarrow$ or $\Leftarrow$), overlined,

$$\overline{M}, \overline{N}, \overline{A}, \overline{B} ::= x \mid y \mid \lambda x.\ \overline{M} \mid \Pi x{:}\overline{A}.\ \overline{B} \mid \overline{M}\ \overline{N} \mid \mathsf{Type} \mid \mathsf{Kind}$$

We reuse the same syntactic elements for both representations. The essential difference between the two representations is that in the first, parameters y are tagged with their type, whereas in the second parameters are bare. This is because it is not the classifiers that we are type checking, but only the subjects of typing judgments. The type of a parameter in a classifier is therefore not needed. We split the infinite set of variables into two disjoint infinite sets of (bound) variables x on the one hand and parameters y on the other. There is no way of binding parameters, because they stand in place of free variables.

In the rules of Figure 2, the judgment $\vdash M \Rightarrow A$ reads "the expression N synthesizes type A" and the judgment $\vdash M \Leftarrow A$ reads "the expression M checks against type A". Within judgments, we omit making the representation explicit through overlining and underlining, because which representation we mean is evident from the position in a judgment. In some rules we observe a crossover of subparts of the subject into the classifier and *vice versa*. In the (app^b) rule, the argument part N crosses over to the right of the synthesis arrow. In the (abs^b) rule, the type A appears both in the subject part of a premise, but in the classifier part

$\boxed{\vdash M \Rightarrow A}$ Term M synthesizes type A

$$(sort^b)\ \dfrac{}{\vdash \mathsf{Type} \Rightarrow \mathsf{Kind}} \qquad\qquad (var^b)\ \dfrac{}{\vdash [x : A] \Rightarrow A}$$

$$(app^b)\ \dfrac{\vdash M \Rightarrow C \qquad C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash N \Leftarrow A}{\vdash M\ N \Rightarrow \{N/x\}B}$$

$\boxed{\vdash M \Leftarrow A}$ Term M checks against type A

$$(abs^b)\ \dfrac{C \longrightarrow_w^* \Pi x{:}A.\ B \qquad \vdash \{[y : A]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.\ M \Leftarrow C}$$

$$(prod^b)\ \dfrac{\vdash A \Leftarrow \mathsf{Type} \qquad \vdash \{[y : A]/x\}B \Leftarrow s}{\vdash \Pi x{:}A.\ B \Leftarrow s}\ s \in \{\mathsf{Type}, \mathsf{Kind}\}$$

$$(conv^b)\ \dfrac{\vdash N \Rightarrow B}{\vdash N \Leftarrow A}\ A \equiv B$$

Figure 2: Bidirectional context-free type checking for the $\lambda\Pi^b$-calculus modulo

of the conclusion. We are therefore implicitly applying a coercion between representations, determined by the following relation between subject and classifier representations:

$$\dfrac{}{x \sim x} \qquad \dfrac{}{[y : \overline{A}] \sim y} \qquad \dfrac{}{\mathsf{Type} \sim \mathsf{Type}} \qquad \dfrac{}{\mathsf{Kind} \sim \mathsf{Kind}}$$

$$\dfrac{\underline{M} \sim \overline{M}}{\lambda x.\ \underline{M} \sim \lambda x.\ \overline{M}} \qquad \dfrac{\underline{A} \sim \overline{A} \qquad \underline{B} \sim \overline{B}}{\Pi x{:}\underline{A}.\ \underline{B} \sim \Pi x{:}\overline{A}.\ \overline{B}} \qquad \dfrac{\underline{M} \sim \overline{M} \qquad \underline{N} \sim \overline{N}}{\underline{M}\ \underline{N} \sim \overline{M}\ \overline{N}}$$

This relation is left-total and right-unique, so that it determines a function from subject representations to classifier representations. Synthesis rules should be read top-down and checking rules bottom-up, so we don't need a coercion in the opposite direction.

In the (abs^b) and $(prod^b)$ rules, the new parameter y introduced in the premises must be chosen fresh. This condition can always be satisfied since terms are of finite size and the set of parameters is infinite.

Example 4. Consider the polymorphic identity as described in section 2. A type derivation for this term in the $\lambda\Pi^b$-calculus modulo can be given as follows:

$$\dfrac{\Pi A.\ \ldots \longrightarrow_w^* \Pi A.\ \ldots \qquad \dfrac{\Pi x{:}(\varepsilon_\star\ y_1).\ (\varepsilon_\star\ y_1) \longrightarrow_w^* \Pi x{:}(\varepsilon_\star\ y_1).\ (\varepsilon_\star\ y_1) \qquad \dfrac{\dfrac{}{\vdash [y_2 : \varepsilon_\star\ y_1] \Rightarrow \varepsilon_\star\ y_1}}{\vdash [y_2 : \varepsilon_\star\ y_1] \Leftarrow \varepsilon_\star\ y_1}}{\vdash \lambda x.\ x \Leftarrow \Pi x{:}(\varepsilon_\star\ y_1).\ (\varepsilon_\star\ y_1)}}{\vdash \lambda A.\ \lambda x.\ x \Leftarrow \Pi A{:}U_\star.\ \Pi x{:}(\varepsilon_\star\ A).\ (\varepsilon_\star\ A)}$$

3.4 Reduction on types

The algorithm presented here relies on finding the weak head normal form of types in order to proceed. This imposes that the set of rewrite rules used to generate the congruence ($\equiv$) verifies a certain number of properties for the algorithm to be deterministic and complete. Notice that reduction happens only on the classifier parts of a judgment. Reduction hence does not need to concern itself with dealing with boxes.

Definition 5 (Weak reduction)**.** Given a set $\mathcal{R}$ of rewrite rules, we define weak reduction ($\longrightarrow_w$) as the contextual closure of the union of the usual β-rule with $\mathcal{R}$ under the reduction context

$$C ::= [] \mid C\ M \mid M\ C \mid \Pi x{:}C.\ B$$

Its reflexive and transitive closure is written ($\longrightarrow_w^*$) and its reflexive, transitive and symmetric closure is written ($\equiv^w$).

Definition 6 (Standardization)**.** A relation on terms, provided it is confluent, is said to be standardizing if:

1. $M \equiv x\ N_1\ \ldots\ N_n$ implies $M \longrightarrow_w^* x\ N_1'\ \ldots\ N_n'$ and $N_1 \equiv N_1', \ldots, N_n \equiv N_n'$;
2. $M \equiv \lambda x.\ N$ implies $M \longrightarrow_w^* \lambda x.\ N'$ and $N' \equiv N$;
3. $M \equiv \Pi x{:}A.\ B$ implies $M \longrightarrow_w^* \Pi x{:}A'.\ B'$ and $A' \equiv A$ and $B' \equiv B$;
4. $M \equiv \mathsf{Type}$ implies $M \longrightarrow_w^* \mathsf{Type}$;
5. $M \equiv \mathsf{Kind}$ implies $M \longrightarrow_w^* \mathsf{Kind}$.

If ($\longrightarrow$) is confluent, standardizing and strongly normalizing, then it is sufficient to compute weak head normal forms [2], which are the normal forms of the weak reduction relation defined above. The reduction relation ($\longrightarrow_w$) may in general be non-deterministic — in that case one can fix a specific reduction strategy. This can also be a way to ensure confluence and strong normalization.

Remark 7. The typing rules of this section are closer to an actual implementation of a type checker, but a legitimate question to ask is how do we know that they are sound and complete with respect to the typing rules of Figure 1. Coquand [10] presents essentially the same algorithm as above (though not context-free) for a simple dependent type theory where $\mathsf{Type} : \mathsf{Type}$, and proves it sound through semantic means. Abel and Altenkirch [2] prove soundness and partial completeness of a very similar algorithm to that of Coquand through syntactic means. Their proof relies only on confluence and standardization properties of β-reduction and can readily be adapted to a type system such as the λΠ-calculus modulo provided the rewrite rules respect the same properties. The first author has formalized soundness and completeness results about various context-free type systems [7].

3.5 Rewriting and dependent types

Rewrite rules can be classified according to the head constant to which they pertain. One can read the set of rewrite rules for one same head constant as clauses of a functional program, the left hand side of which contains a number of patterns.

However, contrary to an ordinary functional program, in the presence of dependent types matching one term might force the shape of another term. This will typically happen when a constructor forces one of the indexes of the type family in its result type. One simple example of this situation is the head function on vectors, defined in the following signature:

$$\begin{array}{rclrclrcl} \mathsf{Nat} & : & \mathsf{Type} & \mathsf{Z} & : & \mathsf{Nat} & \mathsf{S} & : & \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{Vector} & : & \Pi n : \mathsf{Nat}.\ \mathsf{Type} & \mathsf{Nil} & : & \mathsf{Vector}\ \mathsf{Z} & \mathsf{Cons} & : & \Pi n{:}\mathsf{Nat}.\ \mathsf{Nat} \to \mathsf{Vector}\ n \to \mathsf{Vector}\ (\mathsf{S}\ n) \end{array}$$

The type and the rewrite rule associated to the head function are:

$$\mathsf{head} \;:\; \Pi n{:}\mathsf{Nat}.\ \mathsf{Vector}\ n \to \mathsf{Nat} \qquad\qquad \mathsf{head}\ \{\mathsf{S}\ n\}\ (\mathsf{Cons}\ n\ h\ tl) \;\longrightarrow\; h$$

A non-linear pattern seems to be needed because n appears twice in its left hand side. We cannot just generalize the first pattern in this rewrite rule, because then the left hand side would be ill-typed. A non-linear pattern would mean that we must check that the instances for the two occurrences of n are convertible. Moreover, in general inductive type families might be indexed by higher-order terms, meaning that we would need to implement higher-order matching. However, what we actually want to express is that any instance to the first pattern of the rewrite rule above is uniquely determined by a matching instance of the second pattern. Indeed, matching on the Cons constructor forces the dependent first argument to head to be $\mathsf{S}\ n$. As such, the first argument to head plays no operational role; it is merely there for the sake of typing. Following Agda [22], in DEDUKTI we support marking parts of the left hand side of a rule as operationally irrelevant using curly braces — actual arguments are not pattern matched against these operationally irrelevant parts, hence we can avoid having to implement full higher-order matching, which carries with it an inherent complexity that we would rather keep out of a proof checking kernel.

Note that the braces could equally have been placed around the n that is fed as argument to Cons. The only important property to verify is that one and only one occurrence of n must not have braces. This occurrence will be the one that binds n.

4 An implementation of the type checking algorithm

We have by now developed an abstract characterization of a proof checking algorithm. However, we have not so far been explicit about how to implement substitution on terms, nor have we discussed how to decide definitional equality algorithmically. Ideally, whatever implementation we choose, it should be both simple and efficient. Indeed, encodings in the style of Section 2.3 introduce many more redexes in types than were present in the original input. Moreover, computing the weak head normal form of some types may require an arbitrarily large number of reductions, even before any encoding overhead, such as can be the case in proofs by reflection.

Normalization by evaluation (NbE) is one particularly easy to implement normalization scheme that alternates phases of weak reduction (*i.e.* evaluation) and *reification* phases (or *readback* phases [17]). Weak reduction is significantly easier to implement than strong reduction, which in general requires reduction under binders. Evaluation conveniently side steps the traditionally vexing issues of substitution such as renaming of variables to avoid capture, because evaluation only ever involves substituting closed terms for variables[1]. What's more, if evaluation is all that is required, we can pick efficient off-the-shelf evaluators for stock functional programming languages to do the job. These evaluators are invariably much more efficient than we could feasibly implement ourselves — their implementations have been fine-tuned and well studied for the benefit of faster functional programs over many years. What's more, we achieve a better separation of concerns by using off-the-shelf components: we are in the business of writing type checkers, not evaluators. The size of the trusted base arguably does not decrease if we are to rely on an existing evaluator, but trust is all the better for it nonetheless, because a mature off-the-shelf component has likely been stress tested by many more users in many more settings than any bespoke implementation we can come up with.

[1] A term may contain parameters but those cannot be captured: there are no binding forms for parameters in the syntax of terms.

$$\begin{aligned}
||x|| &= x \\
||y|| &= \mathsf{Var}_C\ y \\
||\lambda x.\ \overline{M}|| &= \mathsf{Lam}_C\ (\lambda x \rightarrow ||\overline{M}||) \\
||\Pi x{:}\overline{A}.\ \overline{B}|| &= \mathsf{Pi}_C\ ||\overline{A}||\ (\lambda x \rightarrow ||\overline{B}||) \\
||\overline{M}\ \overline{N}|| &= \mathsf{app}\ ||\overline{M}||\ ||\overline{N}|| \\
||\mathsf{Type}|| &= \mathsf{Type}_C \\
||\mathsf{Kind}|| &= \mathsf{Kind}_C
\end{aligned}$$

(a)

$$\begin{aligned}
|x| &= x \\
|[y : \overline{A}]| &= \mathsf{Box}_T\ y \\
|\lambda x.\ \underline{M}| &= \mathsf{Lam}_T\ (\lambda x \rightarrow |\underline{M}|) \\
|\Pi x{:}\underline{A}.\ \underline{B}| &= \mathsf{Pi}_T\ |\underline{A}|\ (\lambda x \rightarrow |\underline{B}|) \\
|\underline{M}\ \underline{N}| &= \mathsf{App}_T\ |\underline{M}|\ |\underline{N}| \\
|\mathsf{Type}| &= \mathsf{Type}_T
\end{aligned}$$

(b)

Figure 3: Deep embedding of (a) classifiers and (b) subjects into HASKELL.

For simplicity, we use an untyped variant of NbE, in the style of [6] (see also [16, 20, 9]). We proceed by translating terms of the λΠ-calculus modulo into HASKELL programs. Assume the following datatype of evaluated terms:

$$\begin{aligned}
\textbf{data}\ \mathsf{Code} = {} & \mathsf{Var}_C\ \mathsf{Int} \mid \mathsf{App}_C\ \mathsf{Code}\ \mathsf{Code} \\
& \mid\ \mathsf{Lam}_C\ (\mathsf{Code} \rightarrow \mathsf{Code}) \mid \mathsf{Pi}_C\ \mathsf{Code}\ (\mathsf{Code} \rightarrow \mathsf{Code}) \\
& \mid\ \mathsf{Type}_C \mid \mathsf{Kind}_C
\end{aligned}$$

One can view the values of this datatype as code because these values are interesting for their computational behavior. Translation of terms into values of this type is given in Figure 3. Because we only ever need to compute with classifiers during type checking, this translation is defined over the classifier representation of terms, not the subject representation.

The actual identity of a bound variable is usually quite irrelevant[2]. What is relevant is to be able to conveniently substitute another term for all free occurrences of a variable under its binder. Paramaters are dual, in that we never need to substitute for a parameter, nor do they have scope since they cannot be bound, but their identity is crucial. During type-checking, we must be able to ask whether this parameter is the same as this other parameter.

We therefore map (bound) variables to variables of the target language, allowing us to piggy back substitution and scope management onto that of the target language. We get substitution essentially for free. Parameters, on the other hand, are a ground piece of data: a name. Here, for simplicity, we use integers for names.

In a similar spirit, we wish to reuse the target language's computational facilities to reduce terms to normal forms. Therefore applications are mapped to target language applications. However, our embedding of terms in the target language is not quite shallow, in that functions and so on are not represented directly as functions of the target language but rather as values of type Code. This embedding into a monomorphic datatype is essentially an untyped embedding, thus avoiding the impedence mismatch between the type system of the λΠ-calculus modulo and that of the target language, in our case HASKELL. Therefore we must introduce app, a wrapper function around the native application operation, which essentially decapsulates functions embedded in values of type Code:

$$\mathsf{app} :: \mathsf{Code} \rightarrow \mathsf{Code} \rightarrow \mathsf{Code}$$

[2] Indeed this is why formal developments in the presence of binding internalize Barendregt's *variable convention* in some form.

$$\begin{array}{ll} \mathsf{app}\ (\mathsf{Lam}_C\ f)\ t & = f\ t \\ \mathsf{app}\ t_1\ \quad t_2 & = \mathsf{App}_C\ t_1\ t_2 \end{array}$$

If the first argument of app is a function, then we can proceed with the application. If not, we are stuck, and the result is a neutral term.

Deciding the equality between two terms can now be done near structurally:

$$\begin{array}{l} \mathsf{conv} :: \mathsf{Int} \to \mathsf{Code} \to \mathsf{Code} \to \mathsf{Bool} \\ \mathsf{conv}\ n\ (\mathsf{Var}_C\ x)\ (\mathsf{Var}_C\ x') = x \equiv x' \\ \mathsf{conv}\ n\ (\mathsf{Lam}_C\ t)\ (\mathsf{Lam}_C\ t') = \mathsf{conv}\ (n+1)\ (t\ (\mathsf{Var}_C\ n))\ (t'\ (\mathsf{Var}_C\ n)) \\ \mathsf{conv}\ n\ (\mathsf{Pi}_C\ ty_1\ ty_2)\ (\mathsf{Pi}_C\ ty_3\ ty_4) = \mathsf{conv}\ n\ ty_1\ ty_3 \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{conv}\ (n+1)\ (ty_2\ (\mathsf{Var}_C\ n))\ (ty_4\ (\mathsf{Var}_C\ n)) \\ \mathsf{conv}\ n\ (\mathsf{App}_C\ t_1\ t_2)\ (\mathsf{App}_C\ t_3\ t_4) = \mathsf{conv}\ n\ t_1\ t_3 \wedge \mathsf{conv}\ n\ t_2\ t_4 \\ \mathsf{conv}\ n\ \mathsf{Type}_C\ \mathsf{Type}_C = \mathsf{True} \\ \mathsf{conv}\ n\ \mathsf{Kind}_C\ \mathsf{Kind}_C = \mathsf{True} \\ \mathsf{conv}\ n\ _\ _ = \mathsf{False} \end{array}$$

Descending under binding structures gives the weak head normal form of their body, which is computed automatically according to the semantics of the target language. The first argument serves as a supply of fresh parameter names; these need only be locally unique so a local supply is sufficient.

We now have a succinct and yet efficient way of deciding equality of two terms that we know to be well-typed. The translation of Figure 3 for classifiers isn't appropriate, however, for subjects of type checking, because contrary to classifiers, subjects are not *a priori* well-typed. Therefore, we cannot assume that any kind of normal form exists for subjects, lest we compromise the decidability of type checking. We therefore introduce an alternative translation for subjects, where terms are not identified modulo any reduction. This translation is given in part (b) of Figure 3. Note that it is a completely standard mapping of terms to higher order abstract syntax, into values of following datatype:

$$\begin{array}{ll} \mathbf{data}\ \mathsf{Term} = & \mathsf{Box}_T\ \mathsf{Code}\ \mathsf{Code} \mid \mathsf{App}_T\ \mathsf{Term}\ \mathsf{Term} \\ & \mid\ \mathsf{Lam}_T\ (\mathsf{Term} \to \mathsf{Term}) \mid \mathsf{Pi}_T\ \mathsf{Term}\ (\mathsf{Term} \to \mathsf{Term}) \\ & \mid\ \mathsf{Type}_T \end{array}$$

One particularly nice feature of this translation is that it differs from that for classifiers in one single place: the interpretation of applications.

In Dedukti, to control the amount of code that we generate, and to avoid writing wasteful coercions between values of type Term and values of type Code, which we would have to invoke at every application node in the term that we are checking, we generate a single Haskell source for a single proof script, containing both the subject translation and classifier translation for every subterm of the proof script. That is, Term and Code representations are tied together for every subterm, through a system of pairing (or *gluing*). This way no coercion is necessary, but if we performed this gluing naively, it would obviously provoke and explosion in the size of the generated code. The size would grow exponentially with the tree depth of terms, because we would be duplicating subtrees at every level.

For this reason, in the actual implementation, we first transform terms into a linear form, namely A-normal form. In this form, all subterms are named, so that it becomes possible to share substructures of translations at every level, hence avoiding any kind of exponential blowup. The rationale for this translation is similar to that of let-insertion to avoid blowups during partial evaluation [12]. We also perform closure conversion to further improve sharing.

$$||x||^{pat} = x$$
$$||c\ N_1\ \ldots\ N_n||^{pat} = \mathsf{App}\ (\ldots(\mathsf{App}\ (\mathsf{Var}\ c)\ ||N_1||^{pat})\ldots)\ ||N_n||^{pat}$$

$$\left\|\begin{array}{ccc} c\ N_{11}\ \ldots\ N_{1n} & \longrightarrow & M_1 \\ \vdots & & \vdots \\ c\ N_{m1}\ \ldots\ N_{mn} & \longrightarrow & M_m \end{array}\right\| = \begin{array}{cccccc} c & ||N_{11}||^{pat} & \ldots & ||N_{1n}||^{pat} & = & ||M_1|| \\ & \vdots & & \vdots & & \vdots \\ c & ||N_{m1}||^{pat} & \ldots & ||N_{mn}||^{pat} & = & ||M_m|| \\ c & _ & \ldots & _ & = & ||c\ x_1\ \ldots\ x_n||^{pat} \end{array}$$

Figure 4: Translation of rewrite rules as functions over classifiers.

Both of these standard code transformations can be performed selectively, only for subterms where both the subject and classifier representations are needed (both forms are need only when checking applications $M\ N$, and then only when the type synthesized for M is of the form $\Pi x{:}A.\ B$ where x *does* occur in B).

4.1 Extending the algorithmic equality

If we commit to a particular rewrite strategy, and for convenience we commit to a particular rewrite strategy that matches the evaluation order of the target language, then we can extend the scheme given previously to gracefully handle custom rewrite rules. The idea is to read the rewrite rules for one same head constant as the clauses of a functional program. This is the idea behind the translation of rewrite rules as functional programs over values of type Code given in Figure 4. The resulting functional program implements the rewrite rules in exactly the same way that the app function implements β-reduction; if for a given subterm none of the rewrite rules apply, then the result is a neutral term.

5 Conclusion

We have presented in this paper a simple and extensible calculus that works well as a common format for a large number of other proof systems. The essential difference with previous proposals is that through an extensible definitional equality, the λΠ-calculus modulo can act as a logical framework that respects the computational properties of proofs. We showed how this calculus can be implemented very succinctly by leveraging existing evaluation technology.

Our approach is firmly grounded in type theory; computation only happens as a result of checking for convertibility between two types, just as in other type theories. Other recent efforts for a common proof format, such as the proposal of Miller [21] grounded in proof theory, use *focusing* to structure proofs and to formulate customized macro proof rules in terms of some primitive rules. This framework captures a more general notion of computation than the one presented here: Miller's is formulated as proof search rather the functional programming centric approach we present here (our implementation views rewrite rules as inducing a recursive, pattern matching function). Computations in Miller's framework need not be deterministic and can indeed backtrack. The cost of this generality is that the proof checker needs to know how to perform unification and backtracking, whereas in our approach the trusted base is arguably smaller but effects such as backtracking need to be encoded as a pure functional program, for example by using a monad [26].

We have an implementation of a proof checker, Dedukti, which expects the system of rewrite rules provided by the user to be reasonably well behaved. This proof checker is therefore only one small piece of the larger puzzle: confidence in the results of this tool are contingent upon trusting that the provided rewrite rules do not compromise logical consistency, or that these rewrite rules indeed form a confluent and terminating system. Dedukti only checks types, it does not check the rewrite rules. Future work will focus providing these other pieces to support our methodology: tools to (semi-)automatically verify properties about rewrite rules, for example.

For given translations, a system of rewrite rules or a schema of rewrite rules need only be proven to behave properly once, rather than every time for every results of these translations. However, we could also envisage making the λΠ-calculus modulo the core of a full fledged proof environment for end users, in which users can define their own domain specific rewrite rules, in which case automated certification of properties about the resulting system of rewrite rules becomes essential.

We are currently working on adapting size-based termination techniques [1, 25, 4]. This would give a general criterion that would work for the rewrite rules that encodes inductive or coinductive structures. This criterion would be more powerful than purely syntactic criteria such as Coq's guard condition [4].

Finally, we currently type check by first translating proofs and formulas into a functional program in Haskell, which we compile to native code using the GHC compiler. However, the compilation process is slow. Its cost is only amortized given proofs with significant computational content and even then, we would rather not have the compiler waste time optimizing parts of the proof that are computationally completely irrelevant. Other systems such as Coq and Isabelle ask the user to decide exactly when to use an optimizing compiler and when to compute using a much more lightweight interpreter. The advantage of our approach is its simplicity, both for the user and for the implementation. We would like to keep this model, but to make it scale, we are investigating translating terms to a language that supports "just in time" compilation, whereby hotspots that are worth compiling to native code are identified at runtime and automatically, still without any intervention from the user.

A stable release of Dedukti is available at `https://www.rocq.inria.fr/deducteam/Dedukti/index.html`, however some of the improvements discussed here (in particular, the dot patterns of section 3.5) have been integrated only in the development version of Dedukti, that can be found on Github: `https://github.com/mpu/dedukti`.

References

[1] Andreas Abel. Miniagda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *PAR*, volume 43 of *EPTCS*, pages 14–28, 2010.

[2] Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type: Type. *Electr. Notes Theor. Comput. Sci.*, 229(5):3–17, 2011.

[3] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[4] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comp. Sci.*, 14(1):97–141, 2004.

[5] Gilles Barthe and Morten Heine Srensen. Domain-free pure type systems. In *J. Funct. Program.*, pages 9–20. Springer, 1993.

[6] Mathieu Boespflug. Conversion by evaluation. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *LNCS*, pages 58–72. Springer, 2010.

[7] Mathieu Boespflug. *Conception d'un noyau de vrification de preuves pour le λΠ-calcul modulo.* PhD thesis, Ecole polytechnique, January 2011.

[8] Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

[9] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jouannaud and Shao [19], pages 362–377.

[10] Thierry Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

[11] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[12] Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 73–94. Springer, 1996.

[13] NG De Bruijn. A plea for weaker frameworks. In *Logical frameworks*, pages 40–67. Cambridge University Press, 1991.

[14] Gilles Dowek. Proof normalization for a first-order formulation of higher-order logic. In Elsa L. Gunter and Amy P. Felty, editors, *TPHOLs*, volume 1275 of *LNCS*, pages 105–119. Springer, 1997.

[15] Gilles Dowek and Olivier Hermant. A simple proof that super-consistency implies cut-elimination. *Notre-Dame Journal of Formal Logic*, 2012. to appear.

[16] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *LNCS*, pages 167–181. Springer, 2004.

[17] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *ICFP*, pages 235–246. ACM, 2002.

[18] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[19] Jean-Pierre Jouannaud and Zhong Shao, editors. *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *LNCS*. Springer, 2011.

[20] S. Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages.* PhD thesis, 2005.

[21] Dale Miller. A proposal for broad spectrum proof certificates. In Jouannaud and Shao [19], pages 54–69.

[22] Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2008.

[23] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Ewing L. Lusk and Ross A. Overbeek, editors, *CADE*, volume 310 of *LNCS*, pages 772–773. Springer, 1988.

[24] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999.

[25] Cody Roux. *Terminaison à base de tailles: Sémantique et généralisations.* Thèse de doctorat, Université Henri Poincaré — Nancy 1, April 2011.

[26] Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, pages 113–128, 1985.

[27] Michael Zeller, Aaron Stump, and Morgan Deters. Signature compilation for the edinburgh logical framework. *Electr. Notes Theor. Comput. Sci.*, 196:129–135, 2008.

CoqInE: Translating the Calculus of Inductive Constructions into the $\lambda\Pi$-calculus Modulo

Mathieu Boespflug[1] and Guillaume Burel[2]

[1] McGill University
Montréal, Québec, Canada
mboes@cs.mcgill.ca
[2] ÉNSIIE/Cédric/Inria AE Deducteam
1 square de la résistance, 91025 Évry cedex, France
guillaume.burel@ensiie.fr

Abstract

We show how to translate the Calculus of Inductive Constructions (CIC) as implemented by Coq into the $\lambda\Pi$-calculus modulo, a proposed common backend proof format for heterogeneous proof assistants.

1 Introduction

Verification of large developments will rely increasingly on a combination of heterogeneous provers, each being adapted to a particular aspect of the specification to realize. The question of the validity of the whole proof quickly arises. A way to recover confidence over all parts of the proofs is to translate them into a unique, simple but strong enough, formalism, in which they are all checked. A good candidate for such a formalism is the $\lambda\Pi$-calculus modulo, a simple yet extensible logical framework that conserves good properties of proofs, as presented in a companion paper [3]. A proof checker based on it, called Dedukti, has been developed by the first author. It was shown that many proof systems, if not all, can be encoded into the $\lambda\Pi$-calculus modulo, including HOL [7], pure type systems [4], resolution proofs for first-order logic, etc. In this paper, we present how to translate proofs of the Coq proof assistant into this formalism. This translation has been implemented in a tool called CoqInE[1], making it possible to use Dedukti to check Coq's developments.

The core formalism implemented by Coq, the Calculus of inductive Constructions (CIC), is an immensely expressive type theory. As implemented by Coq, it embeds all other systems of the λ-cube [1], includes local and global definitions, full dependent inductive datatype definitions, guarded fixpoint definitions and pattern matching, large eliminations, as well as a cumulative hierarchy of universes, to name but a few features. None of these features are native to the $\lambda\Pi$-calculus modulo in its bare form, which we call the $\lambda\Pi$-calculus. Encoding specifications and proofs from such a powerful formalism is therefore a good benchmark as to the adaptability and versatility of the $\lambda\Pi$-calculus modulo, and its convenience as a common proof format for all manner of systems.

2 Encoding the Calculus of Inductive Constructions

2.1 The Calculus of Constructions

The calculus of inductive constructions (CIC) is an extension of the calculus of constructions (CC), which is a type system for the λ-calculus with dependent types. [4] provides a sound

[1] Available at https://github.com/gburel/coqine.

and conservative encoding of all pure type systems, including CC, into the λΠ-calculus modulo (see also in [3] the instance of this encoding for System F). The CIC as implemented in Coq is in fact a pure type system with an infinity of sorts, called universes. To keep things simple, we will as a first approximation keep to only three sorts, Prop, Set and Type. Section 2.5 will discuss how to deal with universes.

In this paper, we use Dedukti's syntax for the λΠ-calculus modulo, where `x : A -> B` denotes the dependent product, `x : A => B` denotes the abstraction (*la* Church) and `[ Γ ] l --> r.` is the rule rewriting `l` to `r`, Γ being the typing context of the free variables of `l`. Following [4], we have the following constants for each sort $\mathsf{s} \in \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}\}$:

```
Us : Type.
es : Us -> Type.
```

The first is called a *universe constant*, for which we also have an associated *decoding function* (ε_{s}). For each axiom $\mathsf{s}_1 : \mathsf{s}_2$ in $\{\mathsf{Prop} : \mathsf{Type}, \mathsf{Set} : \mathsf{Type}\}$, [4] introduce a constant ($\dot{\mathsf{s}}$) and a rewrite rule:

```
dots1 : Us2.
[] es2 dots1 --> Us1.
```

For each rule $\langle \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_2\rangle$ in $\{\langle\mathsf{Prop},\mathsf{Prop},\mathsf{Prop}\rangle, \langle\mathsf{Prop},\mathsf{Set},\mathsf{Set}\rangle, \langle\mathsf{Prop},\mathsf{Type},\mathsf{Type}\rangle, \langle\mathsf{Set},\mathsf{Prop},\mathsf{Prop}\rangle, \langle\mathsf{Set},\mathsf{Set},\mathsf{Set}\rangle, \langle\mathsf{Set},\mathsf{Type},\mathsf{Type}\rangle, \langle\mathsf{Type},\mathsf{Prop},\mathsf{Prop}\rangle, \langle\mathsf{Type},\mathsf{Set},\mathsf{Set}\rangle, \langle\mathsf{Type},\mathsf{Type},\mathsf{Type}\rangle\}$ we have a constant encoding the dependent product ($\dot{\Pi}_{\langle \mathsf{s}_1,\mathsf{s}_2,\mathsf{s}_2\rangle}$) together with a rewrite rule:

```
dotpis1s2 : x : Us1 -> y : (es1 x -> Us2) -> Us2).
[x : Us1, y : es1 x -> Us2]  es2 (dotpis1s2 x y) --> w : es1 x -> es2 (y w).
```

[4] also defines two translations: $|t|$ translates t viewed as a term, and $||t||$ translates t viewed as a type, with $||A|| = \mathsf{es}\ |A|$ for terms A typed by a sort s.

2.2 Inductive Types

CIC extends CC such that it becomes possible to define inductive types. In our translation, each time an inductive type is defined, we add new declarations and rewrite rules corresponding to the type. To give an intuition, let us first present how a concrete example, namely the vector indexed type, is translated.

Indexed vectors are defined in Coq by:

```
Inductive vector (A : Type) : nat -> Type :=
  Vnil : vector A O
| Vcons : forall (a : A) (n : nat), vector A n -> vector A (S n).
```

where `nat` is itself an inductive type with constructors `O` and `S`. The translation consists first in declaring two corresponding types in λΠ-calculus modulo:

```
vector : A : Utype -> (eset nat) -> Utype.
pre__vector : A : Utype -> (eset nat) -> Utype.
```

The `pre__vector` will be used to tag constructors. This will be useful for fixpoints (see next section). First, we give constants for each constructor, building a `pre__vector`:

```
Vnil : A : Utype -> etype (pre__vector A (nat__constr O)).
Vcons : A : Utype -> a : etype A -> n : eset nat ->
  etype (vector A n) -> etype (pre__vector A (nat__constr (S n))).
```

As we can see above in the case of the constructors of `nat`, constructors are guarded by a constant `vector__constr` allowing to pass from `pre__vectors` to `vectors`:

```
vector__constr : A : Utype -> n : eset nat ->
   etype (pre__vector A n) -> etype (vector A n).
```

Thus, the translation as a term of a constructor applied to its arguments will use this guard:

$$\begin{aligned}|\texttt{Vnil}\ A| &= \texttt{vector__constr}\ |A|\ \texttt{(nat__constr O) (Vnil}\ |A|\texttt{)}\\ |\texttt{Vcons}\ A\ a\ n\ v| &= \texttt{vector__constr}\ |A|\ \texttt{(nat__constr S (}|n|\texttt{)) (Vcons}\ |A|\ |a|\ |n|\ |v|\texttt{)}.\end{aligned}$$

If a constructor is only partially applied, we use η-expand it during translation. For the sake of readability, we now hide `__constr` guards using italics : *O* represents `nat__constr O`, *Vnil* A denotes `vector__constr` A *O* `(Vnil` A`)`, etc.

To translate pattern matching, we also add a case constant:

```
vector__case : A : Utype ->
  P : (n : eset nat -> vector A n -> Utype) ->
  fVnil : etype (P O (Vnil A)) ->
  fVcons : ( a : A -> n : eset nat -> v : etype (vector A n) ->
             P (S n) (Vcons A a n v) ) ->
  n : eset nat ->
  m : etype (vector A n) ->
    etype (P n m).
```

together with two rewrite rules, one for each constructor:

```
[A : Utype, P : ...] vector__case A P fVnil fVcons O (Vnil A) --> fVnil.
[...] vector__case A P fVnil fVcons (S n) (Vcons A a n v) --> fVcons a n v.
```

We now turn to the general case. Following the notations of Coq's manual [8], an inductive definition is given by $\mathsf{Ind}()[p](\Gamma_I := \Gamma_c)$ where p is the number of inductive parameters (as opposed to real arguments), Γ_I is the context of the definitions of (mutual) inductive types and Γ_C the context of constructors. For each inductive type definition $I : t$ we first add two declarations

$\texttt{I :}\ ||t||.$
$\texttt{pre__I :}\ ||t||.$

When the type t of the inductive type is of the form $\Pi x_1 : t_1.\ \ldots \Pi x_n : t_n.\ \mathsf{s}$, we also define a guarding constant:

$\texttt{I__constr : x1:}||t_1||\ \texttt{->} \cdots \texttt{->}\ \texttt{xn:}||t_n||\ \texttt{-> es (pre__I x1}\ \cdots\ \texttt{xn) -> es (I x1}\ \cdots\ \texttt{xn).}$

For each constructor $c : \Pi x_1 : t_1.\ \ldots \Pi x_p : t_p.\ \Pi y_1 : s_1.\ \ldots \Pi y_l : s_l.\ I x_1 \ldots x_p\ u_1 \ldots u_{n-p}$ we declare a constant:

$\texttt{c : x1 :}\ ||t_1||\ \texttt{->} \cdots \texttt{-> xp :}\ ||t_p||\ \texttt{-> y1 :}\ ||s_1||\ \texttt{->} \cdots \texttt{-> yl :}\ ||s_l||\ \texttt{->}$
$\quad \texttt{es (pre__I x1}\ \cdots\ \texttt{xp}\ |u_1|\ \cdots\ |u_{n-p}|\texttt{).}$

The translation as a term of a constructor applied to its arguments will therefore be:

$$|c\ p_1 \ldots p_p\ r_1 \ldots r_l| = \texttt{I__constr}\ |p_1| \ldots |p_p|\ |v_1| \ldots |v_{n-p}|\ (\texttt{c}\ |p_1| \ldots |p_p|\ |r_1| \ldots |r_l|)$$

where $v_i = u_i\{p_1/x_1, \ldots, p_p/x_p, r_1/y_1, \ldots, r_l/y_l\}$.

To translate pattern matching, we also add a case constant:

```
I__case : x1 : ||t_1|| -> ··· -> xp : ||t_1|| ->
  P : (xp+1 : ||t_{p+1}|| -> ··· -> xn : ||t_n|| -> I x1 ··· xn -> UType) ->
    ... ->
  fc : (y1:||s_1|| -> ··· -> yl:||s_l|| -> P |u_1| ··· |u_{n-p}| |c x1 ··· xp y1 ··· yl|) ->
    ... ->
  xp+1 : ||t_{p+1}|| ->  ··· -> xn : ||t_n|| ->
  m : I x1 ··· xn ->
  etype (P xp+1 ··· xn m).
```

where `P` is the return type, and we have `fc` for each constructor, representing the corresponding branch of the matching. We then add a rewrite rule for each constructor:

```
[...] I__case x1 ... xp P ... fc ... |u_1| ... |u_{n-p}|
        (I__constr x1 ... xp |u_1| ... |u_{n-p}| (c x1 ... xp y1 ... yl)
  --> fc y1 ... yl.
```

2.3 Fixed points

For each fixed point in a term, we add a new constant with an associated rewrite rule. Since rewrite rules are defined globally, this constant must take as parameters the context of the fixed point, that is, all bounded varables above the position of the fixed point.

Given a fixed point $\mathsf{Fix} f_i\{f_1/k_1 : A_1 := t_1 \quad \ldots \quad f_n/k_n : A_n := t_n\}$ in a context $x_1 : B_1$, $\ldots$, $x_m : B_m$, assuming A_i is $\Pi y_i^1 : A_i^1. \ldots \Pi y_i^{k_i} : A_i^{k_i}. A_i'$ where $A_i^{k_i}$ is an inductive type $I_i w_i^1 \ldots w_i^{l_i}$ of sort `s`, we obtain the following declarations:

```
f1 : x1 : ||B_1|| -> ... -> xm : ||B_m|| -> ||A_1||.
[x1 : ||B_1||, ..., xm : ||B_m||, y11 : ||A_1^1||, ..., yk1-1 : ||A_1^{k_1-1}||,
 z1 : es (pre__I1 |w_1^1| ... |w_1^{l_1}|)]
   f1 x1 ... xm y11 ... y1k1-1 (I1__constr |w_1^1| ... |w_1^{l_1}| z1)
     --> |t_1| y11 ... y1k1-1 (I1__constr |w_1^1| ... |w_1^{l_1}| z1).
  ⋮
fn : x1 : ||B_1|| -> ... -> xm : ||B_m|| -> ||A_n||.
[x1 : ||B_1||, ..., xm : ||B_m||, yn1 : ||A_n^1||, ..., ynkn-1 : ||A_n^{k_n-1}||,
 zn : es (pre__In |w_n^1| ... |w_n^{l_n}|)]
   fn x1 ... xm yn1 ... ynkn-1 (In__constr |w_n^1| ... |w_n^{l_n}| zn)
     --> |t_n| yn1 ... ynkn-1 (In__constr |w_n^1| ... |w_n^{l_n}| zn).
```

The `Ii__constr` prevents the reduction of the fixed point unless a constructor is present as its k_ith parameter, which is exactly its semantics. Then, $\mathsf{Fix} f_i\{f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n\}$ is translated as `fi x1` ... `xm` where the `xi` are the actual variables of the context of the fixed point. Because of dependent types, it is not possible to have a rewrite rule for the unfolding of the fixed point for each constructors of the inductive type. Indeed, the arguments w_i^j of the inductive type I_i can depend on y_i^k, and this can be incompatible with the type of the constructors. For instance, if we had a fixed point $\mathsf{Fix} f\{f : \Pi A : \mathsf{Set}.\ \Pi x\ y : A.\ \Pi p : x =_A y.\ P := t\}$, the following rewrite rule would be ill-typed:

```
f : A : Uset -> x : eset A -> y : eset A -> p : eprop (eq A x y) -> ||P||.
[A:Uset, x:eset A, y:eset A] f A x y |eq_refl A x|
--> |t| A x y |eq_refl A x|.
```

because $|eq_refl\ A\ x|$ has type `eprop (eq A x x)` and not `eprop (eq A x y)`. This is the reason why we choose to translate constructors with the guarding constant `I__constr`.

Another solution would have been to duplicate the argument on which structural induction is based, one copy serving to match a constructor, the other being used to typecheck the body of the fixed point. Concretely, on the example above, this would give:

```
f : A : Uset -> x : eset A -> y : eset A -> p : eprop (eq A x y) -> ||P||.
[...] f A x y p -> f' A x y p A x y p.
f' : A : Uset -> x : eset A -> y : eset A -> p : eprop (eq A x y) ->
 A' : Uset -> x' : eset A' -> y' : eset A' -> p' : eprop (eq A' x' y') -> ||P||.
[...] f' A x y p A' x' x' (eq_refl A' x') --> |t|.
```

We plan to implement this second solution in a future version of CoqInE, since it avoids doubling the size of ground terms (*i.e.* terms build using only constructors).

Note that Dedukti does not check that the rewrite system terminates, in particular the rules corresponding to the fixed points. Indeed, the philosophy behind Dedukti is to have different tools for each purpose. The termination of the system corresponding to a fixed point should, however, be checkable using a tool based on size-based termination [2].

2.4 Modules

Coq features a powerful system of modules, with functors and refined structures. However, Dedukti understands only a notion of namespaces, lacking anything close to a full fledged module system.

To be able to translate functors, our choice is to parameterize each definition of a structure by all the parameters of the argument module. Then, for functor application, we need to apply all arguments of the applied module in each definitions. Concretely, if we have the following definitions in Coq:

```
Module Type T.
Parameter a : Type.
Parameter b : a.
End T.
```

```
Module M : T.
Definition a := Prop.
Definition b : Prop := forall p : Prop, p.
End M.
```

```
Module N (O : T).
Definition x := O.b.
End N.
```

```
Module P := N(M).
```

we obtain the following modules in Dedukti:

In module `M.dk`*:*

```
a : Utype.
[] a --> dotprop.
b : Uprop.
[] b --> dotpitp dotprop (p : etype dotprop => p).
```

In module `N.dk`*:*

```
x : fp_a : Utype -> fp_b : etype fp_a -> etype fp_a.
[fp_a : Utype, fp_b : etype fp_a] x fp_a fp_b --> fp_b.
```

In module `P.dk`*:*

```
x : etype M.a.
[] x --> N.x M.a M.b.
```

2.5 Universes

Since in CIC there are in fact an infinity of universes, we cannot declare constants `Us`, `es`, etc. for each of them. A first solution would be to restrict oneself to the universes actually used in the module we are translating. This number is finite and typically very small [2]. However, this solution is not modular, since there are definitions that are polymorphic with respect to the universes. Typically, the `list` type takes as first parameter an argument of type `Type` for all universes, so that it can be seen as a family of inductive types. If a module uses `list` on a higher universe than those that were used when `list` was defined, we would have to duplicate its definition.

A better solution would be to have sorts parameterized by universe levels. Although technically possible, it should be proved that this provides a sound and conservative encoding of CIC. Currently, we bypass this issue by translating all universes by `Type`, and having `dottype : Utype`. Of course, this is not sound, since Girard's paradox can be reproduced.

Another issue concerns subtyping of sorts. Indeed, in CIC, if $t : Type(i)$ then $t : Type(j)$ for all $j \geq i$. Rewrite rules could be used to encode subtyping, for instance `Utype i --> Utype j`, but only if they are applied at positive positions. Since subtyping is contravariant, at negative positions, the dual rule `Utype j --> Utype i` should be used. Nevertheless, such a mechanism of *polarized* rewrite rules [5, 6] is currently not present in Dedukti.

3 Implementation details

CoqInE takes a Coq compiled file (`.vo`) and translates it into a Dedukti input file (`.dk`). Concretely speaking, a Coq compiled file contains a copy of the Coq kernel's in-memory representation of a module marshalled to disk. Since the marshalling facility is provided by the OCaml compiler, the format of the content in a `.vo` file is dependent both on the version of the compiler that compiled Coq, and also the version of Coq itself. Reading a Coq compiled file is a simple matter of unmarshalling the data, as implemented in the OCaml standard library.

The advantage of this scheme is that it avoids having to write a parser to read Coq proofs. Moreover, this ensures that the translation is performed on exactly the same terms that are used in Coq: on the contrary, an export of Coq's terms into another format could be unsound. And last but not least, it makes it easy to reuse part of Coq's code, because the data structures used to represent a Coq module are identical. In fact, CoqInE started as a fork of Chicken, a stripped down version of the Coq kernel. However, this has to following drawbacks:

- the same version of OCaml as the one used to compile Coq has to be used to compile CoqInE;
- The close coupling between the data structures of CoqInE and those of the Coq implementation means that CoqInE is tributary to Coq's implementation: core data structures change frequently between Coq releases (as happened between versions 8.2 and 8.3 for instance) and these changes need to be ported to CoqInE's codebase every time.

A better scheme would be to harness an export facility to a generic and stable format. Coq can already export proof snippets in XML. Unfortunately, however, this facility does not currently scale up to all the content of a full module.

[2] For instance, it is possible to type all of the Coq 8.2 standard library using only 4 levels of universes.

4 Results and Future Work

CoqInE is able to translate 256 modules of the standard library of Coq out of 304 (84%). The main features that are missing to translate the whole library are the following: coinductive types are not supported for the moment. A mechanism for handling lazy terms would be needed to get them. We cannot rely on abstracting terms to avoid their reduction, because Dedukti reduces terms under abstractions when it checks their convertibility. Also, translation of modules is not complete, in particular we do not translate refined structures (`S with p := p'`).

We could not check all of these translations using Dedukti. Indeed, although the translation is linear, some of them are too big. For instance, the `Coq.Lists.List` module, whose `.vo` file has length 260KB, is translated into a Dedukti file of length 1.2MB, which is in turn translated into a Haskell file of size 12MB. Of course, GHC is not able to run or compile this file. A simple solution to overcome this issue would be to cut big modules into smaller files. A more stable solution consists in making Dedukti no longer produce Haskell code, but code that can be compiled just-in-time. Dedukti is currently rewritten to allow this.

Another limitation of Dedukti is the absence of higher-order matching. Higher-order patterns can appear for instance in the rules for `I__case` when the real arguments of the inductive type I are higher order. For instance, if we define the inductive type

```
Inductive I : (Prop -> Prop) -> Prop := c : I (fun t => t).
```

we obtain the following rewrite rule:

```
[...] I__case P fc (t : Uprop => t) (I__constr (t : Uprop => t) c) --> fc.
```

which contains higher-order patterns. However, these higher-order patterns are not useful for the matching, since the important pattern is the constructor `c`. Therefore, the possibility to declare that a pattern should be used for the typing of the rule but not for the matching has been added recently to Dedukti. Nevertheless, these so-called dot patterns (see [3]) are currently not used in CoqInE yet.

As noted in Section 2.5, one last remaining point is the encoding of universes.

References

[1] Henk Barendregt. *Handbook of logic in computer science*, volume 2, chapter Lambda calculi with types, pages 117–309. Oxford University Press, 1992.

[2] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comp. Sci.*, 14(1):97–141, 2004.

[3] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

[4] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[5] Gilles Dowek. What is a theory? In Helmut Alt and Afonso Ferreira, editors, *STACS*, volume 2285 of *LNCS*, pages 50–64. Springer, 2002.

[6] Gilles Dowek. Polarized resolution modulo. In Cristian S. Calude and Vladimiro Sassone, editors, *IFIP TCS*, volume 323 of *IFIP AICT*, pages 182–196. Springer, 2010.

[7] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):1–25, 2001.

[8] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA, 2010. Version 8.3, available at `http://coq.inria.fr/refman/index.html`.

System Feature Description: Importing Refutations into the GAPT Framework

Cvetan Dunchev[1], Alexander Leitsch[1], Tomer Libal[1], Martin Riener[1], Mikheil Rukhaia[1], Daniel Weller[2] and Bruno Woltzenlogel-Paleo[1]

{cdunchev,leitsch,shaolin,riener,mrukhaia,weller,bruno}@logic.at

[1] Institute of Computer Languages (E185)

[2] Institute of Discrete Mathematics and Geometry (E104)

Vienna University of Technology

Abstract

This paper describes a new feature of the GAPT framework, namely the ability to import refutations obtained from external automated theorem provers. To cope with coarse-grained, under-specified and non-standard inference rules used by various theorem provers, the technique of proof replaying is employed. The refutations provided by external theorem provers are replayed using GAPT's built-in resolution prover (TAP), which generates refutations that use only three basic fine-grained inference rules (resolution, factoring and paramodulation) and are therefore more suitable for manipulation by the proof-theoretic algorithms implemented in GAPT.

1 Introduction

GAPT[1] (General Architecture for Proof Theory) is a framework that aims at providing data structures, algorithms and user interfaces for analyzing and transforming formal proofs. GAPT was conceived to replace and expand the scope of the CERES system[2] beyond the original focus on cut-elimination by resolution for first-order logic [BL00]. Through a more flexible implementation based on basic data structures for simply-typed lambda calculus and for sequent and resolution proofs, in the hybrid functional object-oriented language Scala [OSV10], GAPT has already allowed the generalization of the cut-elimination by resolution method to proofs in higher-order logic [HLW11] and to schematic proofs [DLRW12]. Furthermore, methods for structuring and compressing proofs, such as cut-introduction [HLW12] and Herbrand Sequent Extraction [HLWWP08] have recently been implemented.

However, due to GAPT's focus on flexibility and generality, efficiency is only a secondary concern. Therefore, it is advantageous for GAPT to delegate proof search to specialized external automated theorem provers (ATPs), such as Prover9 [McC10a], Vampire [RV02] or Otter[3]. This poses the technical problem of importing proofs from ATPs into GAPT, which is less trivial than it might seem, because different ATPs use different inference rules and some inference rules are too coarse-grained, under-specified, not precisely documented [BW11] and possibly not so standard from a proof-theoretical point of view. As an example, take the explanation of the `rewrite` rule from Prover9's manual [McC10b]:

> `rewrite([38(5,R),47(5),59(6,R)])` – rewriting (demodulation) with equations 38, 47, then 59; the arguments (5), (5), and (6) identify the positions of the rewritten subterms (in an obscure way), and the argument R indicates that the demodulator is used backward (right-to-left).

[1] GAPT: http://code.google.com/p/gapt/

[2] CERES: http://www.logic.at/ceres/

[3] Other well-known and very efficient provers such as E and SPASS have not received much of our attention yet, because their output seemed not as easy to understand and parse.

The use of coarse-grained, under-specified and non-standard inference rules is especially problematic for GAPT, because its proof transformation algorithms require that the proofs adhere to strict and minimalistic calculi, as is usually the case in proof theory. To solve this problem in a robust manner, the technique of proof replaying was implemented in GAPT.

In GAPT's historical predecessor CERES, a more direct translation of each of Prover9's inference rules into pre-defined corresponding sequences of resolution and paramodulation steps had been implemented. Recomputing missing unifiers and figuring out the obscure undocumented ways in which Prover9 assigns numbers to positions made this direct translation particularly hard. Thanks to the technique of proof replaying, these problems were avoided in GAPT.

The main purpose of this paper is to document how GAPT's internal resolution prover (TAP) was extended to support proof replaying and to report our overall experience with this technique. TAP outputs resolution proofs containing only fine-grained resolution, factoring and paramodulation steps, as desired.

Proof replaying is a widely used technique, and the literature on the topic is vast (see e.g. [Fuc97, Amj08, PB10, ZMSZ04, Mei00]). One thing that distinguishes the work presented here is that proofs are replayed into a logical system whose main purpose differs substantially from the typical purposes (e.g. proving theorems, checking proofs) of most logical systems. GAPT's ongoing goal of automating the analysis and transformation of proofs can be seen as complementary and posterior to the goals of most other systems.

The rest of the paper is organized as follows: Section 2 describes the general algorithm for proof replay in our system and explains in more details how we implemented this algorithm for Prover9 output. In Section 3 we give a concrete example. The final section concludes our work and discusses some future improvements.

2 Proof Replaying in GAPT

The aim of this section is to describe how a proof from an external theorem prover is replayed in GAPT. At our disposal is the interactive prover TAP, which implements Robinson's resolution calculus [Rob65] and paramodulation. It is not efficient enough to prove complex theorems, but provided with an external proof it is often able to derive, for each inference step, the conclusion clause from its premises. In principle, if the conclusion clause C is not a tautology, a clause will be derived subsuming C by the forward computation of resolution (see [Lee67]). It works for any calculus with tautology-deletion. For our purposes, the specialized coarse-grained inference steps used in proofs output by optimized theorem provers have to be translated into a series of simple resolution and paramodulation steps. If this series is not too long, TAP will find it and use it instead of the specialized inference rule used by the external prover.

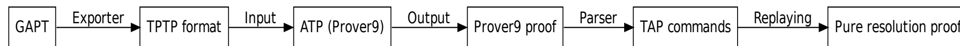


Figure 1: Flowgraph of the Transformation

The complete transformation process is visualized in Figure 1. The theorem to be proved is exported into TPTP format [Sut11, Sut09] and passed to the external theorem prover. For the sake of this paper, we chose to use Prover9 as the external theorem prover. In principle, any

prover whose proof output can be parsed by GAPT can be used in place of Prover9, and we plan to add support for more external provers in the future. In the case of a successful result, the proof output is usually a text file, containing for each inference step its ID, a clause which was derived and a list of the rules which were applied[4]. The output file is then parsed into commands which steer TAP to replay the proof.

The API of TAP has two main modules: the search algorithm and the proof replay. The search space consists of elements called configurations. Each configuration consists of a state, a stream of scheduled commands, additional arbitrary data and a result (possibly empty) which is a resolution proof. The state consists of the persistent data of the configuration and might be shared between different configurations. A command transforms a configuration to a list of successor configurations.

The so called "engine function" takes a configuration, executes the first of its scheduled commands and inserts the newly generated configurations into the search space. By default, the prover explores the search space using breadth-first search, but this is configurable.

We now describe the commands used for replay in more detail. In principle, we could replay an input proof purely using the Replay command. However, the actual implementation treats Prover9's inference rules *assumption*, *copy* and $factor$ specially because they do not create new proofs and are therefore translated directly into TAP commands. In the first case, a proof of the assumption without premises is inserted into the list of derived clauses. In the second case, the guidance map containing the association of proofs to Prover9's inference identifiers is updated. Factoring is treated as a copy rule, because it is integrated into the resolution rule like in Robinson's original resolution calculus[Rob65]. Therefore we postpone the factoring of a clause until its next use in a resolution step. All other Prover9 inferences are replayed. The commands are grouped into different tasks: initialization, data manipulation and configuration manipulation. Guided commands are a subset of data commands. Table 1 provides an overview over the commands necessary for replay (commands in italics were specifically added for replaying).

Initialization Commands	**Replay Commands**	**Data Commands**
Prover9Init	*Replay*	SetTargetClause
		SetClauseWithProof
Configuration Commands	**Guided Commands**	Variants
SetStream	*AddGuidedInitialClause*	Factor
PrependOnCond	*AddGuidedClauses*	DeterministicAnd
RefutationReached	*GetGuidedClauses*	Resolve
	IsGuidedNotFound	Paramodulation
		InsertResolvent

Table 1: Selection of TAP Commands

The initialization commands interface TAP with an external prover. At the moment, there is only one command handling Prover9. It exports the given clause set to TPTP format, hands it over to Prover9, processes its output with the Prooftrans utility to annotate the inference

[4]Some provers have scripts translating their proof format to XML or TSTP. Since TSTP does not fix the set of inference rules, some adjustments to the replaying have to be made for the different provers. In the actual system, a postprocessing to XML format is used which already separates the inference identifiers from the rule name and the list of clauses, but which does not apply any proof tranformations.

identifiers, clauses and rule names with XML tags and uses Scala's XML library to parse the resulting proof into the system. Each assumption is registered together with its inference ID and put into the set of derived clauses (using AddGuidedInitialClause and InsertResolvent). The copy and the factor rules are treated by adding the proof with the new ID to the guidance map (using AddGuidedClauses). For all other rules, the replay command is issued.

The configuration commands allow control over the proof search process. It is possible to schedule insertion of additional commands into certain configurations and to stop the prover when a (desired) resolution deduction is found.

All the data commands transform a configuration to a (finite) list of successor configurations. A simple example is SetTargetClause which configures with which derived clause to stop the prover. Also the commands for the usual operations of variant generation, factoring, paramodulation and resolution are in this group. It also contains commands to insert a found proof into the set of already found derivations and a command for executing two commands after each other on the same state.

The purpose of the guided commands is the bookkeeping of derived proofs. It allows storage of the proof of a clause in a guidance map which is part of the state. When a guided inference is retrieved, the proof is put into the list of derived clauses within that state. There is also a special command looking for the result of a guided inference and inserting it into the set of derived clauses.

Replaying a rule first needs to retrieve the proofs of the parent clauses from the guidance map. Then it creates a new TAP instance, which it initializes with these proofs as already derived clauses and the reflexivity axiom for equality. The conclusion clause of the inference step to be replayed is set as target clause and the prover is configured to use a strategy which tries alternating applications of the resolution and paramodulation rule on variants of the input clauses. Also forward and backward subsumption are applied after each inference step. In this local instance neither applications of the replay rule nor access to the guidance map is necessary. If the local TAP instance terminates with a resolution derivation, it is put into the global guidance map and returned as part of the configuration. In case TAP can not prove the inference, the list of successor states is empty. Since the scheduled replay commands are consecutive transformations on the same configuration, this also means the global TAP instance will stop without result.

3 An Example

In this section we explain with a simple example how our algorithm works for a concrete proof. Consider the clause set from Figure 2, which was obtained from an analysis of a mathematical proof [BHL+06].

```
cnf( sequent0,axiom,'f'('+'(X1, X0)) = '0' | 'f'('+'(X0, X1)) = '1').
cnf( sequent1,axiom,~'f'('+'(X2, X1)) = '0' | ~'f'('+'('+'('+'(X2, X1), '1'), X0)) = '0').
cnf( sequent2,axiom,~'f'('+'(X2, X1)) = '1' | ~'f'('+'('+'('+'(X2, X1), '1'), X0)) = '1').
```

Figure 2: Example of a clause set in TPTP format

We give this clause set to Prover9, which outputs the refutation given on Figure 3. We see that the information contained in a rule description is incomplete – the unifier is normally left out and the variable names in the resulting clause are normalized. In many cases (such as in the last step) more than one step is applied at once. Clause 22 is rewritten twice

into clause 3 (back_rewrite(3),rewrite([22(2),22(8)])), yielding two equational tautologies (xx(a),xx(b)) which are deleted, resulting in the empty clause.

```
1 f(plus(A,B)) = zero | f(plus(B,A)) = one # label(sequent0) # label(axiom).
[assumption].
2 f(plus(A,B)) != zero | f(plus(plus(plus(A,B),one),C)) != zero # label(sequent1) #
label(axiom).  [assumption].
3 f(plus(A,B)) != one | f(plus(plus(plus(A,B),one),C)) != one # label(sequent2) #
label(axiom).  [assumption].
5 f(plus(A,B)) != zero | f(plus(C,plus(plus(A,B),one))) = one.  [resolve(2,b,1,a)].
11 f(plus(A,plus(plus(B,C),one))) = one | f(plus(C,B)) = one.  [resolve(5,a,1,a)].
16 f(plus(A,B)) = one | f(plus(C,D)) != one.  [resolve(11,a,3,b)].
20 f(plus(A,B)) = one | f(plus(C,D)) = one.  [resolve(16,b,11,a)].
22 f(plus(A,B)) = one.  [factor(20,a,b)].
24 $F. [back_rewrite(3),rewrite([22(2),22(8)]),xx(a),xx(b)].
```

Figure 3: Example of a Prover9 refutation of the clause set

In our approach each (nontrivial) step is translated into a series of commands to the internal prover. The series of commands starts by initializing the prover with only those clauses contributing to the current inference and then schedules the resolution derivation. The last command of the series inserts the proof of the resolution step into the already replayed derivation tree.

In the example above, all steps except the last one are trivial steps and TAP returns exactly the same inferences. For the last step the following command is created:

```
List(ReplayCommand(List(0, 3, 22, 22), 24, ([], [])), InsertResolvent())
```

which says that from the reflexivity predicate 0, clauses 3 and variants of 22 it should derive clause 24, i.e. the empty clause.

A full output of TAP for this example is too big to fit nicely on a page.[5] Since the only interesting case is the last step, Figure 4 displays the corresponding generated resolution derivation – in this case of the empty clause.

4 Conclusion

In this paper we described GAPT's new feature of replaying refutations output by ATPs. Our approach is based on interpreting coarse-grained, under-specified and non-standard inference rules as streams of commands for GAPT's built-in prover TAP. By executing these commands, TAP generates resolution refutations containing only inference rules that are fine-grained and standard enough for GAPT's purposes. This approach is simpler to implement and more robust. The drawback is that its reliance on proof search by a non-optimized prover (TAP) makes replaying less efficient than a direct translation.

In the future, we plan to add support for the TSTP proof format, in order to benefit not only from Prover9 but from any prover using this format. As GAPT's algorithms for proof analysis and proof compression mature, we expect them to be of value in post-processing the proofs obtained by ATPs.

[5]The full output can be found here: http://code.google.com/p/gapt/wiki/PxTP2012

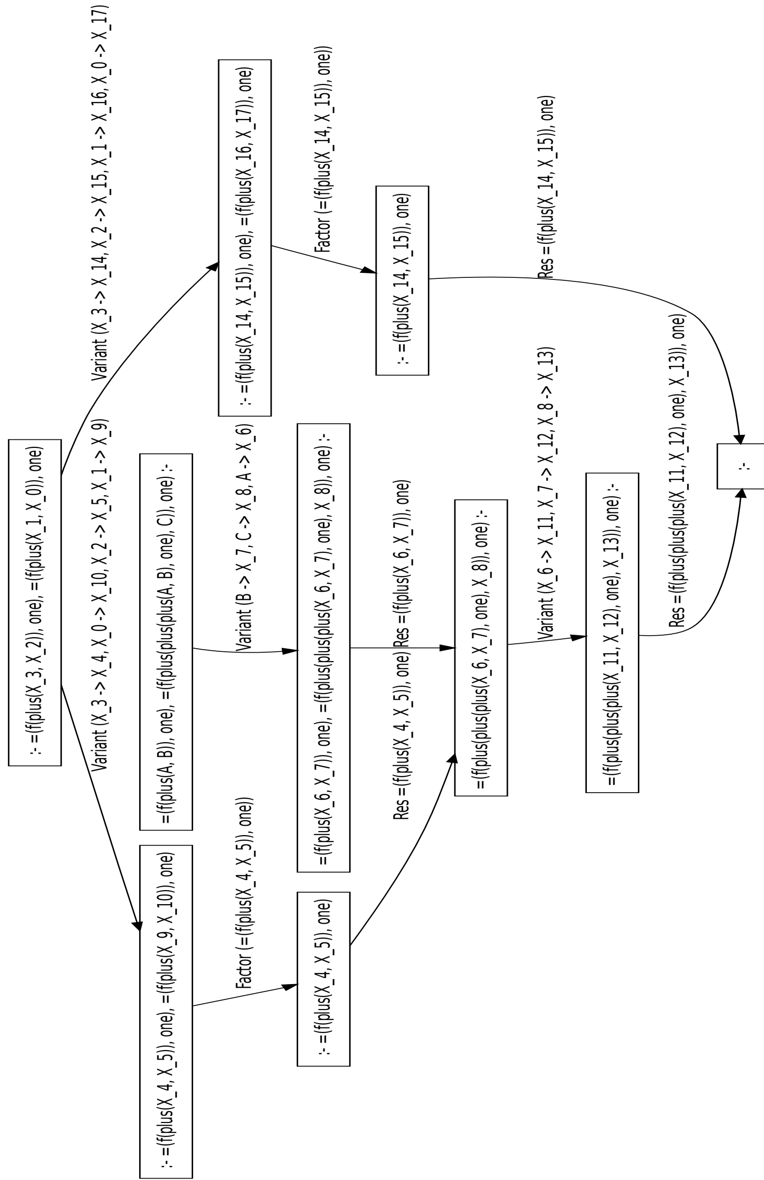


Figure 4: Replayed resolution tree of the last step of the example

References

[Amj08] Hasan Amjad. Data compression for proof replay. *Journal of Automated Reasoning*, 41:193–218, 2008.

[BHL+06] Matthias Baaz, Stefan Hetzl, Alexander Leitsch, Clemens Richter, and Hendrik Spohr. Proof transformation by CERES. In Jonathan M. Borwein and William M. Farmer,

editors, *Mathematical Knowledge Management (MKM) 2006*, volume 4108 of *Lecture Notes in Artificial Intelligence*, pages 82–93. Springer, 2006.

[BL00] Matthias Baaz and Alexander Leitsch. Cut-elimination and redundancy-elimination by resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.

[BW11] Sascha Böhme and Tjark Weber. Designing proof formats: A user's perspective. In *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving*, 2011.

[DLRW12] C. Dunchev, A. Leitsch, M. Rukhaia, and D. Weller. About Schemata And Proofs web page, 2010–2012. `http://www.logic.at/asap`.

[Fuc97] Marc Fuchs. Flexible proof-replay with heuristics. In Ernesto Coasta and Amilcar Cardoso, editors, *Progress in Artificial Intelligence*, volume 1323 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 1997.

[HLW11] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. CERES in higher-order logic. *Annals of Pure and Applied Logic*, 162(12):1001–1034, 2011.

[HLW12] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. Towards algorithmic cut-introduction. In *LPAR*, pages 228–242, 2012.

[HLWWP08] Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo. Herbrand sequent extraction. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 462–477. Springer Berlin, 2008.

[Lee67] Char-Tung Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms.* PhD thesis, 1967. AAI6810359.

[McC10a] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[McC10b] W. McCune. Prover9 and mace4 manual - output files, 2005–2010. `https://www.cs.unm.edu/~mccune/mace4/manual/2009-11A/output.html`.

[Mei00] Andreas Meier. System description: TRAMP - Transformation of machine-found proofs into natural deduction proofs at the assertion level. In *Proceedings of the 17th International Conference on Automated Deduction, number 1831 in Lecture Notes in Artificial Intelligence*, pages 460–464. Springer-Verlag, 2000.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima, Inc., 2nd edition, 2010.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Eugenia Ternovska, and Stephan Schulz, editors, *Proceedings of the 8th International Workshop on the Implementation of Logics*, 2010.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of Vampire. *AI Commun.*, 15(2,3):91–110, 2002.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Sut11] G. Sutcliffe. TPTP Syntax EBNF, 2011. `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`.

[ZMSZ04] J. Zimmer, A. Meier, G. Sutcliffe, and Y. Zhang. Integrated proof transformation services. In *Workshop on Computer-Supported Mathematical Theory Development*, 2004.

Walking through the Forest: Fast EUF Proof-Checking Algorithms

Frédéric Besson, Pierre-Emmanuel Cornilleau and Ronan Saillard

Inria Rennes – Bretagne Atlantique, France

Abstract

The quantifier-free logic of Equality with Uninterpreted Function symbols (EUF) is at the core of Satisfiability Modulo Theory (SMT) solvers. There exist several competing proof formats for EUF proofs. We propose original proof formats obtained from *proof forests* that are the artifacts proposed by Nieuwenhuis and Oliveras to extract efficiently EUF unsatisfiable cores. Our proof formats can be generated by SMT solvers for almost free. Moreover, our preliminary experiments show that our novel verifiers outperform other existing EUF verifiers and that our proof formats appear to be more concise than existing EUF proofs.

1 Introduction

SMT solvers (*e.g.*, Z3 [15], Yices [16], CVC3 [4], veriT [10]) are the cornerstone of software verification tools (*e.g.*, Dafny [18], Why/Krakatoa/Caduceus [17], VCC [11]). They are capable of discharging proof obligations of impressive size and make possible verification tasks that would otherwise be unfeasible. Scalability matters, but when it comes to verifying critical software, soundness is mandatory. SMT solvers are based on highly optimised decision procedures and proving the correctness of their implementation is probably not a viable option. To sidestep this difficulty, several SMT solvers are generating proof witnesses that can be validated by external verifiers. In order to minimise the Trusted Computing Base (TCB), an ambitious approach consists in validating SMT proofs using a general purpose proof verifier *i.e.,* a proof-assistant such as Isabelle/HOL or Coq. Recent works [9, 2] show that SMT proofs are big objects and that the bottleneck is usually the proof-assistant. Ideally, SMT proofs should be i) generated by SMT solvers with little overhead; ii) verified quickly by proof assistants.

Even for the simplest logic such as the quantifier-free logic of equality with uninterpreted function symbols (EUF) there exist competing proof formats generated by SMT solvers. Several of those formats have been validated by proof-assistants: Z3 proofs can be reconstructed in Isabelle/HOL and HOL4 [9]; veriT proofs can be reconstructed in Coq [2]. Even though the results are impressive, certain SMT proofs cannot be verified because they are too big.

We propose novel proof formats for EUF and compare their efficiency in terms of i) generation time; ii) checking time; iii) and proof size. Our comparisons are empirical and do not compare the formats in terms of proof-complexity. Instead, we have implemented the verifiers using the functional language of Coq and compared their asymptotic behaviours experimentally over families of handcrafted benchmarks. The contributions of the paper are efficient Coq verifiers for two novel EUF proof formats and their comparison with existing verifiers. The code of the verifiers is available [21]. Our first proof format is made of proof forests that are the artifact proposed by Nieuwenhuis and Oliveras to extract efficiently unsatisfiable cores [20]. Our second proof format is a trimmed down version of the proof forest reduced only to the edges responsible for triggering a congruence. For the sake of comparison, we have also implemented a Coq verifier for the EUF proof format of Z3 [14] and compared with the existing Coq verifier for the EUF proof format of veriT [2]. The main result of our preliminary experiments is that

 David Pichardie, Tjark Weber (eds.); PxTP 2012, pp. 58–64

the running time of all the verifiers is negligible *w.r.t.* the type-checking of the EUF proofs. Another result of our experiments is that proof forests verifiers are very fast and that (trimmed down) proof forests appear to be more concise than existing EUF proofs. Another advantage is that proof forests can be generated for almost free by SMT solvers based on proof forests.

The rest of the paper is organised as follows. Section 2 provides basic definitions and known facts about the EUF logic. Section 3 recalls the nature of proof forests and explains the workings of our novel EUF proof verifiers. Section 4 is devoted to experiments. Section 5 concludes.

2 Background

We write $\mathcal{T}(\Sigma, \mathcal{V})$ the smallest set of terms built upon a set of variables $\mathcal{V}$ and a ranked alphabet Σ of uninterpreted function symbols. For EUF, an atomic formula is of the form $t = t'$ for t, $t' \in \mathcal{T}(\Sigma, \emptyset)$ and a literal is an atomic formula or its negation. Equality (=) is reflexive, symmetric, transitive and closed under congruence:

$$\frac{}{x = x} \qquad \frac{x = y}{y = x} \qquad \frac{x = y, y = z}{x = z} \qquad \frac{x_1 = y_1, \dots, x_n = y_n}{f(x_1, \dots, x_n) = f(y_1, \dots, y_n)}$$

Ackermann has proved using a reduction approach that the EUF logic is decidable [1]. The reduction consists in introducing for each term $f(\vec{x})$ a boolean variable $f_{\vec{x}}$ and encoding the congruence rule by adding the boolean formula $x_1 = y_1 \wedge \dots \wedge x_n = y_n \Rightarrow f_{\vec{x}} = f_{\vec{y}}$ for each pair of terms $f(\vec{x})$, $f(\vec{y})$. This encoding is responsible for a quadratic blow up of the formula. Nelson has proposed an efficient decision procedure for deciding the satisfiability of a set of EUF literals using a congruence closure algorithm [19]. Nieuwenhuis and Oliveras have shown how to efficiently generate unsatisfiable cores by instrumenting the congruence closure algorithm with a proof forest [20] gathering all the necessary information for generating proofs (see Section 3.1).

3 Walking through the proof forest

In the following, we assume *w.l.o.g.* that EUF terms are flat and curried [20, Section 3.1] *i.e.,* are of the form a or $f(a_1, a_2)$ where f represents an explicit application symbol and a, a_1 and a_2 are constants. In the rest, we drop the f and write $a_1(a_2)$ for $f(a_1, a_2)$. These transformations can be performed in linear time and simplify the decision procedure.

3.1 Proof forest

Nieuwenhuis and Oliveras have proposed a *proof-producing* congruence closure algorithm for deciding EUF [20]. Their main contribution is an efficient Explain operation that outputs a (small) set of input equations E needed to deduce an equality, say $a = b$. If $a \neq b$ is also part of the input, $E \cup a \neq b$ is a (small) unsatisfiable core that is used by the SMT solver for backtracking. As a result, SMT solvers using congruence closure run a variant of the Explain algorithm – whether or not they are proof producing.

The Explain algorithm is based on a specific data structure: the proof forest. A proof forest is a collection of trees and each edge $a \to b$ in the proof forest is labelled by a *reason* justifying why the equality $a = b$ holds. A reason is either an input equation $a = b$ or a pair of input equations $a_1(a_2) = a$ and $b_1(b_2) = b$. For the second case, there must be justifications in the forest for $a_1 = b_1$ and $a_2 = b_2$.

We give in Figure 1 a modified version of the original Explain algorithm. The NearestAncestor and Parent functions return (if it exists) respectively the nearest common ancestor of two nodes

```
let ExplainAlongPath(a,c) :=
    a:=HighestNode(a);
    c:=HighestNode(c);
    if a = c then return
    else
      b:=Parent(a);
      if edge has form a --(a=b)--> b
      then  Output (a = b)
      else{ /*edge has form a --(b1(b2)=b / a1(a2)=a)--> b*/
          Output a1(a2)=a and b1(b2)=b ;
          Explain(a1,b1); Explain(a2,b2)};
      Union(a,b); ExplainAlongPath(b,c)
```

(a) ExplainAlongPath

```
let Explain(a,b) :=
  c:=NearestAncestor(a,b);
  ExplainAlongPath(a,c);
  ExplainAlongPath(b,c)
```

(b) Explain

Figure 1: Recursive Explain algorithm

in the proof forest and the parent of a node in the proof forest. A union-find data structure [23] is also used: Union(a,b) merges the equivalence classes of a and b and Find(a) returns the current representive of the class of a. The HighestNode(c) operation returns the highest node (*i.e.,* the node with minimal depth in the proof forest) belonging to the equivalence class of c.

Contrary to the original version, our algorithm is recursive. The Union operation has also been moved. As a consequence output equations are ordered differently. This will ease our futur proof production. Another difference is that the original algorithm always terminates but does not detect certain ill-formed proof forests *e.g.,* $a \xrightarrow[a=f(a)]{b=f(b)} b$. In this case, the recursive algorithm does not terminate (this issue is dealt with in section 4.1).

3.2 Proof forest verifier

The Explain algorithm of Figure 1 can be turned into a EUF proof verifier. The verifier is a version of Explain augmented with additional checks to ensure that the edges obtained from the SMT solver correspond to a well-formed proof forest. For instance, the verifier checks that edges are only labelled by input equations. Moreover, for edges of the form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$, the recursive calls to Explain ensure that $a_1 = b_1$ and $a_2 = b_2$ have proofs in the proof forest *i.e.,* a_1 (resp. a_2) is connected with b_1 (resp. b_2) by some valid path in the proof forest. For efficiency and simplicity, the *least common ancestors* are not computed by the verifier but used as untrusted hints. The soundness of the verifier does not depend on the validity of this information as the proposed *least common ancestor* is just used to guide the proof. If the return node is not a common ancestor, the verifier will simply fail.

For the verifier, a EUF proof is a pruned proof forest corresponding to the edges walked through during a preliminary run of Explain. As the SMT solver needs to traverse the proof forest to extract unsatisfiable core, we argue that the proof forest is a EUF proof that requires no extra-work from the SMT solver.

```
let UFchecker input edges (x,y) :=
  for (a = b) ∈ input do Union(a,b) done
  for a --b1(b2)=b / a1(a2)=a--> b in edges do
    if (a1(a2)=a) ∈ input & (b1(b2)=b) ∈ input
              & isEqual(a1,b1) & isEqual(a2,b2)
    then Union(a,b)  else fail
  done
  return isEqual(x,y)
```

Figure 2: Verifier algorithm for trimmed forests

3.3 A verifier using trimmed forests

To avoid traversing the same edge several times, the Explain algorithm and its verifier are using a union-find data structure. Therefore, the Explain verifier implicitly embeds a *decision procedure* for the theory of equality. Our optimised verifier fully exploits this observation and starts by feeding all the input equalities of the form $a = b$ into its union-find. For the decision procedure, new equalities are obtained by applying the congruence rule and efficiency crucially depends on a clever indexing of the equations. The verifier does not require this costly machinery and takes as argument a trimmed down proof forest reduced to the list of edges of the forest of form $a \xrightarrow[a_1(a_2)=a]{b_1(b_2)=b} b$. The edge labels indicate the equations that need to be paired to derive $a = b$ by the congruence rule. The algorithm of the verifier is given in Figure 2 where the predicate isEqual(a,b) checks whether a and b have the same representative in the union-find *i.e.*, Find(a) = Find(b). Once again, a preliminary run of Explain is sufficient to generate a proof for this optimised verifier.

4 Implementation and experiments

4.1 EUF verifiers in Coq

Our verifiers are implemented using the native version of Coq [8] which features *persistent* arrays [13]. Persistent arrays are purely functional data-structures that ensure constant time accesses and updates of the array as soon as it is used in a monadic way. For maximum efficiency, all the verifiers make a pervasive use of those arrays that allow for an efficient union-find implementation: union and find have their logarithmic asymptotic complexity.

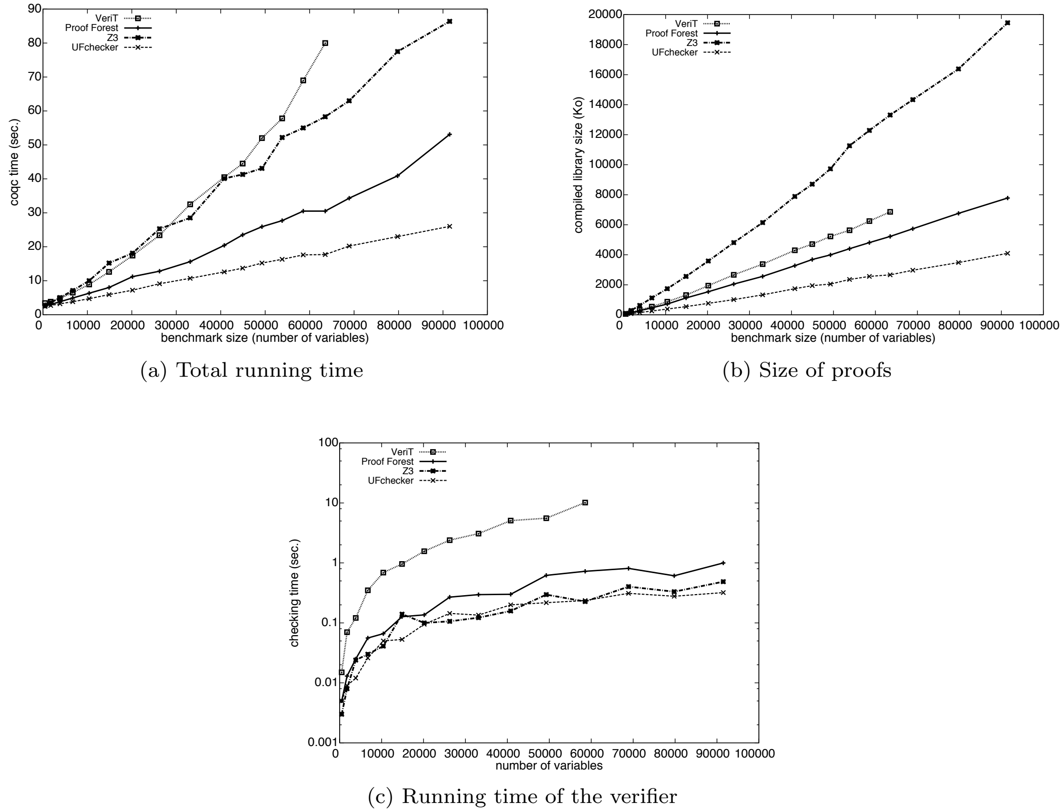
Compared to other languages, a constraint imposed by Coq is that all programs must be terminating. The UFchecker (see Section 3.3) is trivially terminating. Termination of the proof-forest verifier is more intricate because the Explain algorithm (see Figure 3.2) does not terminate if the proof forest is ill-formed *e.g.*, has cycles. However, if the proof forest is well-formed, an edge is only traversed once. As a result, at each recursive call, our verifier decrements an integer initialised to the size of the proof forest. In Coq, the verifier fails after exhausting the maximum number of allowed recursive calls.

For the sake of comparison, we have also implemented the EUF proof format of Z3. Z3 refutations are also generated using Explain [14, Section 3.4.2]. Unlike our verifiers, Z3 proofs use explicit boolean reasoning and *modus ponens*. As a consequence formulae do not have

a constant size. As already noticed by others [9, 2], efficient verifiers require a careful handling of sharing. Our terms and formulae are *hash-consed*; sharing is therefore maximum and comparison of terms or formulae is a constant-time operation.

4.2 Benchmarks

We have assessed the efficiency of our EUF verifiers on several families of handcrafted conjunctive EUF benchmarks. The benchmarks are large and all the literals are necessary to prove unsatisfiability. The formulae are not representative of conflict clauses obtained from SMT-LIB [3] benchmarks which are usually very small [7] and would not stress the scalability of the verifiers. For all our benchmarks the running time of the verifiers is negligible especially compared to the time spent type-checking the EUF proofs and the proof size is linear in the size of the formulae.



(a) Total running time

(b) Size of proofs

(c) Running time of the verifier

Figure 3: Comparison of the verifiers

Figure 3 shows our experimental results for a family F of formulae of the general form

$$F_j = \begin{cases} x_0 = x_1 \qquad x_0 \neq x_{(j+1)\cdot j} \\ f(x_{i\cdot j}, x_{i\cdot j}) = x_{i\cdot j+1} = \ldots = x_{i\cdot j+j} \quad \text{for} \quad i \in \{0 \ldots j\} \end{cases}$$

The benchmarks are indexed by the number of EUF variables and the results are obtained using a Linux laptop with a processor Intel Core 2 Duo T9600 (2.80GHz) and 4GB of Ram.

Figure 3a shows the time needed to construct and compile Coq proof terms. Figure 3b shows the size of the compiled proof terms. Figure 3c is focusing on the running time of the verifiers excluding the time needed to construct and dump proof terms.

For all our benchmarks, the UFchecker shows a noticeable advantage over the other verifiers. We can also remark that its behaviour is more predicable. The veriT verifier [2] is using proofs almost as small as proof forests but the traces generated by veriT are sometimes two orders of magnitude bigger. In the timings, the pre-processing needed to perform this impressive proof reduction is accounted for and might explain why the veriT verifier gets slower as the benchmark size grows. Remark also that for the biggest benchmarks, veriT fails to generate proofs.

Our Z3 verifier scales well despite being slightly but constantly outrun by the verifiers based on proof-forests. The running time of the different verifiers is given Figure 3c using a logarithmic scale. This emphases the fact that, except for the veriT verifier, the running time of the all the verifiers is below the second and is therefore not the limiting factor of the verification. Not surprisingly, the UFchecker requires smaller proofs and therefore its global checking time is also smaller. For big benchmarks, its running time also tends to be smaller than the running time of other verifiers.

5 Conclusion

Proof generating SMT solvers such as Z3 [15], CVC3 [4] or veriT [10] routinely generate EUF proofs in various formats. Those formats have in common that proof-checking is essentially linear in the size of the proof. In theory, our EUF proofs cannot be checked in linear time and the verifiers need to embed a union-find algorithm. In practice, our experiments conclude that our UFchecker verifier outperforms existing verifiers for EUF proofs and that succinct EUF proofs are the key for scalability. Embedding union-find in EUF proofs was previously proposed by Conchon *et al.,* [12]. However, in their approach, union-find computations are interleaved with logic inferences that are responsible for an overhead that is absent from our verifiers. The importance of succinct proofs has been recognised by Stump [22] and its Logical Framework with Side Conditions (LFSC). Unlike LFSC proofs, our proofs are less flexible but purely computational and use the principle of proof by *reflection* [5, Chapter 16]. As future work, we intend to integrate the UFchecker into the SMT proof verifier developed by several of the current authors [6] and extend its scope to the logic of constructors.

References

[1] W. Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

[3] C. Barret, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0, 2010.

[4] C. Barrett and C. Tinelli. CVC3. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004.

[6] F. Besson, P-E. Cornilleau, and D. Pichardie. Modular SMT Proofs for Fast Reflexive Checking Inside Coq. In *CPP*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011.

[7] F. Besson, P-E. Cornilleau, and D. Pichardie. A nelson-oppen based proof system using theory specific proof systems. In *PxTP 2011*, 2011.

[8] M. Boespflug, M. Dénès, and B. Grégoire. Full Reduction at Full Throttle. In *CPP*, volume 7086 of *LNCS*, pages 362–377. Springer, 2011.

[9] S. Böhme and T. Weber. Fast LCF-style Proof Reconstruction for Z3. In *Proc. of ITP 2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[10] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Proc. of CADE 2009*, LNCS. Springer, 2009.

[11] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[12] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. Lightweight integration of the ergo theorem prover inside a proof assistant. In *Proceedings of the second workshop on Automated formal methods*, AFM '07, pages 55–59. ACM, 2007.

[13] S. Conchon and J-C. Filliâtre. Semi-persistent Data Structures. In *ESOP*, volume 4960 of *LNCS*, pages 322–336. Springer, 2008.

[14] L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *Proc. of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants*, volume 418. CEUR-WS.org, 2008.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[16] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[17] J-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007.

[18] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[19] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[20] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *Proc. of RTA 2005*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.

[21] R. Saillard. EUF Verifiers in Coq. `http://www.irisa.fr/celtique/ext/euf`.

[22] A. Stump. Proof checking technology for satisfiability modulo theories. *Electron. Notes Theor. Comput. Sci.*, 228:121–133, January 2009.

[23] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.