EPTCS 337

Proceedings of the

Sixteenth Workshop on

Logical Frameworks and Meta-Languages: Theory and Practice

Pittsburgh, USA, 16th July 2021

Edited by: Elaine Pimentel and Enrico Tassi

Published: 16th July 2021
DOI: 10.4204/EPTCS.337
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains a selection of papers presented at LFMTP 21, the 16th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP), held on July 16, 2021 using the Zoom video conferencing tool due to COVID restrictions. The workshop was affiliated with CADE-28.

Logical frameworks and meta-languages form a common substrate for representing, implementing and reasoning about a wide variety of deductive systems of interest in logic and computer science. Their design, implementation and their use in reasoning tasks, ranging from the correctness of software to the properties of formal systems, have been the focus of considerable research over the last two decades. This workshop will bring together designers, implementors and practitioners to discuss various aspects impinging on the structure and utility of logical frameworks, including the treatment of variable binding, inductive and co-inductive reasoning techniques and the expressiveness and lucidity of the reasoning process.

We received 11 submissions in total, which were reviewed by 3 program committee members. Out of those, 3 were selected as work in progress reports for presentation and 6 were selected for presentation and publication. Each submission selected for publication was reviewed again by 3 program committee members and included in these proceedings.

In addition to the submission presentations, the program included 2 invited talks by Giselle Reis (CMU, Qatar) "Facilitating Meta-Theory Reasoning" and Matthieu Sozeau (Inria, France) "The Meta-Coq Project'.

We want to thank the members of the Program Committee for their efforts in providing timely reviews and for the useful suggestions and participation on the decisions. Moreover, we want to thank the authors and the invited speakers, since their contributions and presentations at the meeting stimulated interesting discussions with the attendees, making the event a success. We are also very grateful to the organization of CADE-28 for providing the infrastructure and coordination with other events.

Program Committee of LFMTP 2021:

- David Baelde (LSV, ENS Paris-Saclay & Inria Paris)
- Roberto Blanco (MPI-SP)
- Alberto Ciaffaglione (University of Udine)
- Claudio Sacerdoti Coen (University of Bologna)
- Marina Lenisa (Universitá degli Studi di Udine)
- Dennis Müller (Friedrich-Alexander-University)
- Michael Norrish (CSIRO)
- Elaine Pimentel (Universidade Federal do Rio Grande do Norte) co- chair
- Ulrich Schöpp (fortiss GmbH)
- Kathrin Stark (Princeton University)
- Aaron Stump (The University of Iowa)

E. Pimentel and E. Tassi (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. ii–iii, doi:10.4204/EPTCS.337.0

- Nora Szasz (Universidad ORT Uruguay)
- Enrico Tassi (Inria) co-chair
- Alwen Tiu (The Australian National University)
- Tjark Weber (Uppsala University)

We are grateful to the external referees Furio Honsell and Pietro Di Gianantonio for their excellent work in reviewing and selecting the submitted papers.

16th of July, 2021

Elaine Pimentel and Enrico Tassi

Facilitating Meta-Theory Reasoning (Invited Paper)

Giselle Reis
Carnegie Mellon University in Qatar
giselle@cmu.edu

Structural proof theory is praised for being a symbolic approach to reasoning and proofs, in which one can define schemas for reasoning steps and manipulate proofs as a mathematical structure. For this to be possible, proof systems must be designed as a set of rules such that proofs using those rules are correct *by construction*. Therefore, one must consider all ways these rules can interact and prove that they satisfy certain properties which makes them "well-behaved". This is called the *meta-theory* of a proof system.

Meta-theory proofs typically involve many cases on structures with lots of symbols. The majority of cases are usually quite similar, and when a proof fails, it might be because of a sub-case on a very specific configuration of rules. Developing these proofs by hand is tedious and error-prone, and their combinatorial nature suggests they could be automated.

There are various approaches on how to automate, either partially or completely, meta-theory proofs. In this paper, I will present some techniques that I have been involved in for facilitating meta-theory reasoning.

1 Introduction

Structural proof theory is a branch of logic that studies proofs as mathematical objects, understanding the kinds of operations and transformations that can be done with them. To do that, one must represent proofs using a regular and unambiguous structure, which is constructed from a fixed set of rules. This set of rules is called a *proof calculus*, and there are many different ones. One of the most popular calculi used nowadays is *sequent calculus* [15] (and its variations).

In its simplest form, a sequent is written $\Gamma \vdash \Delta$, where Γ and Δ are sets or multisets of formulas (depending on the logic). The interpretation of a sequent is that the conjunction of formulas in Γ implies the disjunction of formulas in Δ. Rules in a sequent calculus are written:

$$\frac{P_1 \quad \dots \quad P_n}{C}\ \mathsf{name}$$

where C is the *conclusion* sequent, and P_i are the *premise* sequents. When there are no premises, the rule is called an *axiom*. Sequent calculus proofs are trees where each node is an instance of a rule. A set of rules is considered a "good" or "well-behaved" sequent calculus system if it satisfies a few properties. Among the most important ones are:

1. identity expansion: the system is able to prove $A \vdash A$ for any formula A; and
2. cut elimination: if $\Gamma \vdash \Delta, A$ and $\Gamma, A \vdash \Delta$, then $\Gamma \vdash \Delta$.

A corollary of cut elimination is the calculus' *consistency*, i.e. it cannot derive false. But there are also other properties which are interesting to show, such as rule invertibility and permutability. All

E. Pimentel and E. Tassi (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 1–12, doi:10.4204/EPTCS.337.1

these properties are called the *meta-theory* of a proof system, and they can be used as lemmas in each others' proof, or justifications for sound proof transformations or proof search optimizations. Meta-theory proofs are typically done via structural induction on proof trees, formulas, or both. The number of cases is combinatorial on the number of rules and/or connectives, but cases tend to follow the same argument, with only a few more involved ones.

The appeal of sequent calculus is its versatility and uniformity: there are sequent calculi for a great number of logics, and they are formed by rules which are usually very similar. This similarity is very convenient, specially when it comes to meta-theory proofs. Since different logics share the same rules, parts of proofs can be reused from one system to the other. At first, meta-theory proofs are mostly developed by hand. After seeing the same cases over and over again, we become more and more confident that they will work out[1] and skip more and more steps. Coupled with the sheer complexity and size of such proofs, we end up missing cases and making mistakes, which need to be corrected later.

A proof of cut-elimination for full intuitionistic linear logic (FILL) was shown to have a mistake in [1], and the authors of the proof have later published a full corrected version [2]. A proof of cut-elimination for the sequent calculus GLS_V for the provability logic GL was the source of much controversy until this was resolved in [16] and formalized in [7] using Isabelle/HOL. More recently, another proof of cut elimination for the provability logic GLS was proposed [3], but the inductive measures used were not appropriate. Upon formalizing the proof [17], other researchers not only found the mistake, but were able to get a more self-contained proof. Several sequent calculi proposed for bi-intuitionistic logic were "proved" to enjoy cut-elimination when, in fact, they did not. The mistake is analysed and fixed in [29]. An error in the cut-elimination proof for modal logic nested systems was corrected in [21].

This situation has led many researchers to look for easier and less error prone methods for proving cut elimination. In this paper, I am going to discuss three different approaches: logical frameworks, formalization, and user-friendly implementations. I will focus on those developments where I have been involved in, but I will also mention relevant (though non-exhaustive) related work.

2 Logical Frameworks

In its most general form, a logical framework can be defined as a specification language with a reasoning engine, which is capable of reasoning about specifications written in the language. Formalizing meta-theory proofs in a logical framework involves writing the proof system in the specification language, and expressing properties such as cut elimination in a sentence that can be decided by the reasoning engine.

A classic example is the proof of cut elimination for the intuitionistic sequent calculus LJ in the logical framework LF [19]. LF's specification language is a type theory, so LJ is specified by writing each of its rules as an appropriate type. LJ's cut elimination proof, in its turn, is written by using another type for each proof case. LF's reasoning engine is thus able to infer coverage and termination of the proof. If one trusts LF's checker, then one can be sure that the specified cut elimination proof holds. Unfortunately, LF's method does not translate so elegantly to other logics, and fitting a sequent calculus system and its cut elimination proof in LF's type theory can be quite an involved exercise.

The L-framework uses an implementation of rewriting logic (Maude) to check meta-theory properties of sequent calculus systems [28]. Each inference rule is specified as a rewriting rule, and meta-properties are represented as reachability goals. Different meta-properties are parametrized by different rewriting rules. These include a number of rewriting rules representing the proof transformations relevant to that meta-property, and also the rewriting rules corresponding to the sequent calculus system. The rewriting

[1] If you have read Kahneman's book, system 1 takes over.

$$\frac{\vdash \Gamma;\Delta,A \quad \vdash \Gamma;\Delta,B}{\vdash \Gamma;\Delta,A\,\&\,B}\ \&\qquad \frac{\vdash \Gamma;\Delta,A,B}{\vdash \Gamma;\Delta,A\parr B}\ \parr\qquad \frac{}{\vdash \Gamma;\Delta,\top}\ \top\qquad \frac{\vdash \Gamma;\Delta}{\vdash \Gamma;\Delta,\bot}\ \bot$$

$$\frac{\vdash \Gamma;\Delta_1,A \quad \vdash \Gamma;\Delta_2,B}{\vdash \Gamma;\Delta_1,\Delta_2,A\otimes B}\ \otimes\qquad \frac{\vdash \Gamma;\Delta,A_i}{\vdash \Gamma;\Delta,A_1\oplus A_2}\ \oplus_i\qquad \frac{}{\vdash \Gamma;1}\ 1\qquad \frac{}{\vdash \Gamma;a,a^{\perp}}\ \mathsf{init}$$

$$\frac{\vdash \Gamma,A;\Delta}{\vdash \Gamma;\Delta,?A}\ ?\qquad \frac{\vdash \Gamma;A}{\vdash \Gamma;!A}\ !\qquad \frac{\vdash \Gamma,A;\Delta,A}{\vdash \Gamma,A;\Delta}\ \mathsf{copy}$$

Figure 1: One-sided dyadic sequent calculus for classical linear logic.

logic implementation is responsible for solving the reachability goal, and thus establishing whether the meta-property holds. This technique is sound but not complete. If the meta-theory proof follows a more exotic strategy, then it is likely that the pre-determined set of transformations in the L-framework will not be enough to decide reachability. In spite of this (expected) limitation, the L-framework was used to show a number of meta-properties (invertibility, permutability, cut-elimination, etc) of various sequent calculus systems, including single and multi conclusion intuitionistic logic, classical logic, linear logic, and modal logics.

Linear logic was proposed as a framework for reasoning about sequent calculus systems in [23]. In this technique, each rule in sequent calculus is encoded as a linear logic formula. Meta-properties such as cut-elimination and identity expansion are equivalent to (decidable) properties of the encoding. Therefore, given a set of linear logic formulas representing the encoding of a sequent calculus system, cut elimination can be decided using bounded proof search in linear logic. This method was used to prove meta-properties of linear, classical, and intuitionistic logics. It turns out that the linear logic framework cannot capture so easily sequent calculus systems with rules that have side conditions in the context, such as:

$$\frac{\Box\Gamma \vdash A}{\Box\Gamma,\Gamma' \vdash \Box A,\Delta}\ \Box_R$$

To be able to specify rules like this, we can use *subexponential linear logic* (SELL).

2.1 (Subexponential) Linear Logic

Linear logic is a refinement of classical logic in the sense that it allows a finer control over structural rules. In this logic, formulas that can be contracted or weakened (on the right side) are marked with the exponential operator ?, called question mark. The dual of ? is !, called bang. As a result, there are two kinds of conjunction and disjunction: an additive and a multiplicative, which differ on how the context is split between premises. The one-sided calculus for classical linear logic is depicted in Figure 1. The context is formed by Γ, containing formulas that can be contracted and weakened (i.e. those that were under ?), and Δ, containing the other formulas. A relevant rule for what follows is !, also called *promotion*. Observe that a formula under ! can only be used if no other formulas exist in Δ.

Subexponential linear logic (SELL) extends linear logic by allowing multiple *indexed* exponential operators $!^a, ?^a$ [24]. Each index may or may not allow the rules of contraction and weakening, and they are organized in a pre-order $\preceq$. Since there are formulas under different $?^a$, the sequent in SELL is composed of several contexts, one for each subexponential index. Assuming subexponential indices 1 to n, the rules involving subexponentials are modified as follows. Rule ? stores the formula in the

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_R \qquad \rightsquigarrow \qquad \begin{array}{c} \vdash \mathscr{T}, \lfloor \Gamma \rfloor, \lceil A \rceil \quad \vdash \mathscr{T}, \lfloor \Gamma \rfloor, \lceil B \rceil \\ \vdots \\ \vdash \mathscr{T}, \lfloor \Gamma \rfloor, \lceil A \wedge B \rceil \end{array}$$

Figure 2: The encoding of the object logic rule $\wedge_R$ is a linear logic formula whose derivation results on the right side tree.

appropriate context:

$$\frac{\vdash \Gamma_1; ...; \Gamma_i, A; ...; \Gamma_n; \Delta}{\vdash \Gamma_1; ...; \Gamma_i; ...; \Gamma_n; \Delta, ?^i A}\ ?$$

Formulas under $!^i$ can only be used if all contexts corresponding to indices k such that $i \not\leq k$ and Δ are empty:

$$\frac{\vdash \Gamma_1; ...; \Gamma_n; A}{\vdash \Gamma_1; ...; \Gamma_n; !^i A}\ ! \text{ if } \Gamma_k = \emptyset \text{ for } i \not\leq k$$

Formulas in Γ_i will only retain a copy in the context if i is an index that allows contraction:

$$\frac{\vdash \Gamma_1; ...; \Gamma_i, A; ...; \Gamma_n; \Delta, A}{\vdash \Gamma_1; ...; \Gamma_i, A; ...; \Gamma_n; \Delta}\ \text{copy if } i \text{ allows contraction}$$

$$\frac{\vdash \Gamma_1; ...; \Gamma_i; ...; \Gamma_n; \Delta, A}{\vdash \Gamma_1; ...; \Gamma_i, A; ...; \Gamma_n; \Delta}\ \text{use if } i \text{ does not allow contraction}$$

2.2 Encoding

Using the new promotion rule, and the flexibility in choosing the subexponential indices and their properties, we can encode rules such as:

$$\frac{\Box\Gamma \vdash A}{\Box\Gamma, \Gamma' \vdash \Box A, \Delta}\ \Box_R$$

The encoding in linear logic uses two predicate symbols that map object logic formulas into linear logic predicates: $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$. $\lfloor A \rfloor$ indicates that A is an object logic formula that is on the left side of the object logic sequent. $\lceil A \rceil$ indicates that A is an object logic formula that is on the right side of the object logic sequent. Therefore, the sequent $A_1, ..., A_n \vdash B_1, ..., B_m$ in the object logic is encoded, roughly, as the linear logic sequent: $\vdash \lfloor A_1 \rfloor, ..., \lfloor A_n \rfloor, \lceil B_1 \rceil, ..., \lceil B_m \rceil$. In addition to these formulas, the linear logic sequent also includes a set $\mathscr{T}$ of LL formulas that encode the sequent calculus rules of the considered logic. The derivation of a formula in $\mathscr{T}$ mimics the application of the corresponding rule in the object logic. This general scheme is depicted in Figure 2.

The main idea behind the encoding is to choose indices such that the structure of the SELL sequent $\vdash \Gamma_1; ...; \Gamma_n; \Delta$ contains the parts of the context that need to be distinguished in the object logic. For the example above, we have a subexponential index to store boxed formulas on the left (called $\Box_l$), one to store all other formulas on the left (called l), and one to store all formulas on the right (called r) [25]. Using the appropriate pre-order relation between those, we can force the deletion of formulas in l and r, when deriving the formula corresponding to the encoding of $\Box_R$.

The rule $\Box_R$ above is encoded as the following SELL formula (which resembles a Horn clause):

$$\lceil \Box A \rceil^{\perp} \otimes !^{\Box_l} ?^r \lceil A \rceil$$

If we use a pre-order where indices $\Box_l$, l, and r are not related and allow contraction and weakening, the derivation of the formula above in SELL is:

$$\dfrac{\dfrac{\dfrac{}{\vdash \lfloor\Gamma_l\rfloor;\lfloor\Gamma_{\Box_l}\rfloor;\lceil\Delta\rceil,\lceil\Box A\rceil;\lceil\Box A\rceil,\lceil\Box A\rceil^\perp}\,\mathsf{init}}{\vdash \lfloor\Gamma_l\rfloor;\lfloor\Gamma_{\Box_l}\rfloor;\lceil\Delta\rceil,\lceil\Box A\rceil;\lceil\Box A\rceil^\perp}\,\mathsf{copy} \qquad \dfrac{\dfrac{\vdash \cdot;\lfloor\Gamma_{\Box_l}\rfloor;\lceil A\rceil;\cdot}{\vdash \cdot;\lfloor\Gamma_{\Box_l}\rfloor;\cdot;?^r\lceil A\rceil}\,?}{\vdash \lfloor\Gamma_l\rfloor;\lfloor\Gamma_{\Box_l}\rfloor;\lceil\Delta\rceil,\lceil\Box A\rceil;!^{\Box_l}?^r\lceil A\rceil}\,!}{\vdash \lfloor\Gamma_l\rfloor;\lfloor\Gamma_{\Box_l}\rfloor;\lceil\Delta\rceil,\lceil\Box A\rceil;\lceil\Box A\rceil^\perp\otimes!^{\Box_l}?^r\lceil A\rceil}\,\otimes$$

Observe how the application of rule ! removes precisely the formulas that are weakened in the $\Box_R$ rule, and how the open premise corresponds to the premise of the rule.

2.3 Meta-theory

Given a set of linear logic formulas encoding rules of a sequent calculus system, [23] defined decidable conditions on the formulas that translate into the meta-properties identity expansion and cut elimination. The same criteria could be used for identity expansion of sequent calculus systems encoded in SELL, but not for cut elimination.

Showing cut elimination of systems encoded in LL is split into two parts:

1. show that cuts can be reduced to atomic cuts;
2. show that atomic cuts can be eliminated.

Step 1 relies on the fact that specifications of dual rules are dual LL formulas, and on cut elimination in LL itself. Step 2 is shown by mimicking Gentzen's reduction rules that permute atomic cuts until init rules, and then remove the cuts.

Cut is encoded as the linear logic formula $\lceil A\rceil\otimes\lfloor A\rfloor$, so its permutation in the object logic is equivalent to he permutation of this formula's derivation in LL. It is shown in [23] that this permutation is possible.

In the case of systems encoded in SELL, the general shape of the cut rule is:

$$!^a?^b\lceil A\rceil\otimes!^c?^d\lfloor A\rfloor$$

where $!^a$ and $!^c$ may or may not occur.

The presence of subexponentials is necessary, since sequent calculus systems with more complicated contexts may have cut rules that impose restrictions on the context, or may need more than one cut rule. As a result, the cut elimination argument for those systems may be more involved. They may require proof transformations to be done on a specific order (e.g. permute the cut on the left branch before the right), or may involve other transformations (e.g. permuting rules down the proof instead of permuting the cut up), or may use more complicated induction measures (e.g. a distinction between a "light" and "heavy" cut).

It is thus not a surprise that the elegant cut elimination criteria from [23] does not translate so nicely to SELL encodings. The presence of subexponentials on the formula corresponding to the cut rule may prevent the derivation of this formula from permuting, so the steps that require permutation of cut need to be looked at carefully. Therefore, showing cut elimination of systems encoded in SELL is split into three parts:

1. show that the cut can become *principal* (i.e. the rules immediately above the cut operate on the cut formula);

2. show that principal cuts can become atomic cuts;

3. show that atomic cuts can be eliminated.

Step 2 is shown as in [23], relying on the duality of formulas for dual rules and cut elimination in SELL. We have identified simple conditions for when step 3 can be performed, based on which subexponentials indices occur on the encoding of the rules and how they are related in the pre-order. However, step 1 turned out to be quite complicated. The reason was already alluded to before: making cuts principal may involve permutations of the cut rule using particular strategies, permutations of other rules, or transformations of one cut into another. We have identified conditions that allow for some of these transformations in [25], but it is unlikely that a general criteria that encompasses this variety of operations exists.

Permutation lemmas Step 1 above may involve a number of permutation lemmas between rules, so we focused our efforts in finding an automated method to check for these transformations. Using answer set programming (ASP) and the encoding of rules in SELL, we were able to enumerate all cases in the proof of a permutation lemma, and to decide which of these cases work [26]. The cases in such proofs are the different ways one rule can be applied over another. Using context constraints for each SELL rule, the possible derivations of a formula is computed by a logic program that finds all models that satisfy a set of constraints. Once all possible derivations of a rule over another is found, another logic program computes whether provability of some premises imply provability of others. The check is sound, but not complete.

Implementations The goal of this work was to provide automated ways to check for some meta-properties of sequent calculus systems, so it is only natural that we have implemented the solutions. Initial expansion and cut elimination for systems encoded in SELL are implemented in the tool Tatu[2]. Permutation lemma for systems encoded in SELL is implemented in the tool Quati[3]. Both tools require the user to input only the encoding, and all checks are done with the click of a button. Tatu also features a nice interface that shows the encoded sequent calculus rules and cases for permutation lemmas in LaTeX [27].

Extensions This framework was adapted to the linear nested sequent setting, and it was shown that it can more naturally capture a range of systems with context side conditions [20].

The works on using SELL as a logical frameworks [25, 26] were successful in finding decidable conditions on encodings that translate into meta-properties of the encoded systems. These conditions can be checked completely automatically, as witnessed by their implementations. It is not surprising, however, that the more a meta-theory proof deviates from the "standard" procedure, the fewer cases can be checked automatically. But probably the biggest issue with using SELL to encode systems is coming up with the encoding in the first place. It turns out that, for encoding one system, different subexponential configurations can be used. Each choice might influence on which meta-properties can and cannot be proved (even if the encoded system is correct), and figuring this out requires patience and experience.

[2] http://tatu.gisellereis.com/

[3] http://quati.gisellereis.com/

3 Formalizations in Proof Assistants

Proof assistants are incredibly expressive and powerful tools for programming proofs. Specifications are written usually in a relational or functional fashion, and proofs about such specifications are written as proof scripts. Those scripts basically describe the proof steps, and each step is validated by a proof checker. If the proof assistant is able to execute the proof, and its implementation is trustworthy, then this is a strong guarantee that the proof is correct.

One of the issues when developing proofs of meta-properties by hand is the sheer complexity and number of cases. By implementing these proofs in proof assistants, the computer will not let us skip cases or overlook details.

There are several works that implement different calculi and proofs of meta-properties in proof assistants. We mention a few, though this is far from an exhaustive list. Dawson and Goré proposed a generic method for formalizing sequent calculi in Isabelle/HOL, and implemented meta-properties parametrized by a set of rules [7]. This implementation was used to prove cut elimination of the provability logic GLS_V. Recently, D'Abrera, Dawson, and Goré reimplemented this framework in Coq and used it to implement and prove meta-properties of a linear nested sequent calculus [6]. Tews formalized a proof of cut elimination for coalgebraic logics in Coq, which uncovered a few mistakes in the original pen and paper proof [31]. Graham-Lengrand formalized in Coq completeness of focusing, a proof search discipline for linear logic [18]. Urban and Zhu formalized strong normalization of cut elimination for classical logic in Isabelle/HOL [32].

The fact that each of these works is a publication (or collection of publications) itself is evidence that formalizing meta-theory is far from trivial work and cannot be done as a matter of fact. Even though one would think that specifying sequent calculi on a functional or relational language would be "natural", there is a lot of room for design choices that influence how proofs can be implemented. I have been involved on the formalization of linear logic and its meta-theory in two proof assistants: Abella [4] and Coq [33]. The Abella formalization includes various fragments of linear logic and different calculi, and the meta-theorems proved were identity expansion, cut elimination, and invertibility lemmas when needed. The Coq formalization was done for first order classical LL, and includes the meta-theorems of cut elimination, focusing, and structural properties when needed. The choice of linear logic is strategic: this logic's context is not a set, so we cannot leverage the context of the proof assistant to store the object logic's formulas. Many substructural logics would have similar restrictions, so a solution for linear logic can probably be leveraged for other logics as well. We discuss now the main challenges and insights of those formalizations.

3.1 Specification of Contexts

As mentioned, we could not use Abella or Coq's set-based context to store linear logic formulas, since these need to be stored in a multiset. As a result, the context needs to be explicit in the specification of the sequent calculus. Proof assistants typically have a really good support for lists, so one choice would be to encode contexts as lists. In this case, we would need to show that exchange is height-preserving admissible, and use this lemma each time we need to apply a rule on a formula at the "wrong position". To avoid this extra bureaucracy, and to have specifications and proofs that resemble more what is done on pen and paper, we need to use a multiset library.

For the Abella development we implemented this library from scratch. Multi-sets can be specified in a number of ways, with different operations as its basic constructor. The implementation, operations, and lemmas proved about multisets highly influence the amount of bureaucracy in the meta-theory proofs.

Our implementation uses an `adj` operation on an element and a multiset to add the element to the multiset (akin to list cons). Using `adj` we could define `perm` (equality up to permutation) and `merge` (multiset union). It is possible that our implementation can be further simplified, but we run the risk of over-fitting it for the linear logic case.

Coq has a multiset library where multisets are implemented as bags of type `A -> nat`. It turns out that this implementation of multisets complicates reasoning, so a multiset library was again implemented from scratch. In this case, a multiset is defined as a list, and multiset operations including equality are defined inside a Coq module.

3.2 Handling Binders

Linear logic quantifiers are *binders*. Encoding object level binders has been the topic of extensive research during the last decades [13, 12, 11, 14, 22].

In the Coq formalization, LL quantifiers were encoded using the technique of *parametric* HOAS [5], so substitution and freshness conditions come for free. However, substitution lemmas and structural preservation under substitutions must be assumed as axioms.

In the Abella formalization, the HOAS technique was also employed, but object level binders can be modelled using Abella's nominal quantifier ∇.

To have a more smooth treatment for binders, the work in [33] was adapted to use the Hybrid framework [10, 9]. This has allowed the formalization of a completely internal proof of cut elimination for focused linear logic. Moreover, the encoded linear logic was used as a meta-language for encoding and proving properties of other systems, effectively providing formal proofs for the theorems in the previously discussed work [23].

3.3 Proof Development

The proofs in both Abella and Coq were carried out faithfully to what is usually done with pen and paper. The technique used in Abella was structural induction on proof trees and formulas, wheread Coq's proofs are done mostly on induction on the derivation's height (which is explicit on the specification).

As expected, a lot of details and non-cases that are usually dismissed on paper need to be properly discharged on a proof assistant. As a result, the proof includes some bureaucracy and requires a lot of patience and attention to detail. Using Coq's powerful tactics language Ltac, the work [33] included the implementation of tailored tactics to solve recurring proof goals. The amount of work pays off though, since a formalized proof is a more trustworthy proof.

Ultimately we are looking for a "canonical" way for specifying proof systems on proof assistants, such that meta-property proofs can be done more easily, hopefully using pieces from other proofs (much like it is done on pen and paper). I believe it is safe to say we are not there yet. In the current state of the art, formalizations of meta-theory on proof assistants serve more to increase trust than to facilitate meta-reasoning. But at each new formalization we learn new techniques, and maybe in the future we could use those to implement a libraries that actually facilitate meta-reasoning.

4 User-friendly Implementations

The two previous techniques for meta-reasoning aimed at complete automation, or complete trust. But we do not have to restrict ourselves to extremes. Considering the operations that need to be done during

meta-reasoning, there are a number of them which could be delegated to a computer, without requiring a lot of effort or expertise. Some examples are: checking if two sequents are the same; enumerating all possible ways two sequents can be unified; enumerating all possible ways a rule can be applied; checking if a structure is smaller than another one; computing and applying substitutions; etc. You might even have implemented a couple of those in some context. Most of these tasks have well-established, sound and complete, algorithms, so they could be implemented in an easy-to-use tool to help logicians with boring meta-reasoning proofs. I have been involved in at least two such projects: GAPT [8] and Sequoia [30].

4.1 GAPT

GAPT[4] stands for General Architecture for Proof Theory, and it is a software for investigating, implementing, and visualizing proof transformations. This system grew from years of proof theorists collaborating with the implementation of the algorithms they were studying at the time. Coupled with an organized and systematic software engineering, it was possible to build a common basis for all the different proof systems and transformations.

Terms, formulas, and sequents are among GAPT's built-in datatypes. In addition to that, it contains implementations of the sequent calculus for classical logic LK, Gentzen's natural deduction system, and resolution. GAPT is able to import proofs from various automated theorem provers, but it also includes its own prover. Users have the option of building their own LK proof using GAPT's tactic language: gaptic. The imported proofs can be manipulated using GAPT's API, and visualized using the GUI (which also provides limited manipulation options).

Among the proof transformations implemented in GAPT, we highlight some well-known ones:

- Gentzen's cut elimination;
- skolemization: removing positive occurrences of $\forall$ and negative occurrences of $\exists$;
- interpolation: given a proof of $\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$, find I such that $\Gamma_1 \vdash \Delta_1, I$ and $\Gamma_2, I \vdash \Delta_2$ are provable and I contains only predicates that appear in both partitions;
- translation from LK to natural deduction;

Extending GAPT with another calculus or proof transformation requires one to delve into its development, understanding the code and its modules. However, most of the infrastructure is already there, and there are many implementations to take inspiration from. GAPT was not built for meta-reasoning specifically, but given the amount of proof transformations involved in meta-theory, it is a good platform for experimenting with how a calculus behaves under these transformations. It is not far fetched to think of implementing, for example, the enumeration of ways a rule can be applied on a sequent, given that the notion of sequent and rules already exist.

4.2 Sequoia

Sequoia[5] is a web-based tool for helping with meta-theory of sequent calculi. It was built to be as user-friendly as possible. Users input their calculi in LaTeX, and are able to build proofs in it by clicking on a sequent and on the rule to be applied. If the rule can be applied in more than one way, the system computes all possibilities and prompts the user to choose one option. The proof tree is rendered in LaTeX on the fly, and users can undo steps if they need to backtrack.

[4] `https://www.logic.at/gapt/`

[5] `https://logic.qatar.cmu.edu/sequoia/`

When it comes to meta-reasoning, sequoia helps by listing all cases it can automatically deduce if the proof follows the "usual" strategy. For example, for identity expansion sequoia will try to build small derivations of size at most 2 for each connective, and check that the open premises could be closed with identity on smaller formulas. If it is able to do so, it will print the small derivation in LaTeX so that the user can check for themselves. The meta-properties sequoia supports are: identity expansion, weakening admissibility, permutability of rules, cut elimination, and rule invertibility.

5 Conclusion

I have discussed three different methods for reasoning about meta-properties of (mostly sequent calculus) proof systems. This is a biased view from my own experience, and should not be taken as the only ways to do meta-reasoning on the computer. Each method has its own advantages and disadvantages, which makes them incomparable.

The solution using (subexponential) linear logic as a framework has the great advantage that all checks can be completely automated, and a "yes" means the property holds, once and for all. However, if the meta-theory proof follows a more esoteric strategy, it is unlikely that this method will work. A "no" is actually "don't know", and the logician is left on their own to check the proof by hand. Another challenge with this technique is the fact that the user needs to be quite familiar with linear logic to be able to come up with a reasonable encoding.

Formalizations in proof assistants have the advantage that one needs to know simply the specification and scripting languages. A familiarity with available libraries and techniques is helpful, but not crucial. Similar to the SELL solution, if the system checks the proof, then one can be sure the property holds. The flexibility in writing proofs allows for the more esoteric proofs to be checked, but the cost of this is less automation. In contrast to the previous approach, the check cannot be done with the click of a button, and requires the logician to go through the long process of implementing the proof and all its cases and details.

The last solution is the least ambitious one, but probably the most realizable in the short term. We can use the computer to aid in parts of meta-reasoning, while leaving the most complicated cases for us to think about. We have seen how a well-designed framework can serve as a platform to test various proof transformations, and how a user-friendly system can allow some easy cases to be computed automatically.

In the end, all solutions have their limitations, and we are still far from a situation where meta-theory proofs can be developed in a more trustworthy fashion. There is still a lot of work to be done into making each of these approaches easier to use and broader. We are slowly making progress.

References

[1] G.M. Bierman (1996): *A note on full intuitionistic linear logic*. *Annals of Pure and Applied Logic* 79(3), pp. 281 – 287, doi:10.1016/0168-0072(96)00004-8.

[2] Torben Braüner & Valeria de Paiva (1996): *Cut-Elimination for Full Intuitionistic Linear Logic*. Technical Report BRICS-RS-96-10, BRICS, Aarhus, Danemark. Also available as Technical Report 395, Computer Laboratory, University of Cambridge.

[3] Jude Brighton (2015): *Cut Elimination for GLS Using the Terminability of its Regress Process*. *Journal of Philosophical Logic* 45, doi:10.1007/s10992-015-9368-4.

[4] Kaustuv Chaudhuri, Leonardo Lima & Giselle Reis (2017): *Formalized Meta-Theory of Sequent Calculi for Substructural Logics*. Electronic Notes in Theoretical Computer Science 332, pp. 57 – 73, doi:10.1016/j.entcs.2017.04.005. LSFA 2016 - 11th Workshop on Logical and Semantic Frameworks with Applications (LSFA).

[5] Adam Chlipala (2008): *Parametric Higher-Order Abstract Syntax for Mechanized Semantics*. SIGPLAN Not. 43(9), p. 143–156, doi:10.1145/1411203.1411226.

[6] Caitlin D'Abrera, Jeremy Dawson & Rajeev Goré (2021): *A formally verified cut-elimination procedure for linear nested sequents for tense logic*. In: 28^{th} International Conference on Automated Deduction (CADE-28). Accepted for publication.

[7] Jeremy E. Dawson & Rajeev Goré (2010): *Generic Methods for Formalising Sequent Calculi Applied to Provability Logic*. In: Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings, pp. 263–277, doi:10.1007/978-3-642-16242-8_19.

[8] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner & Sebastian Zivota (2016): *System Description: GAPT 2.0*. In: 8^{th} International Joint Conference on Automated Reasoning, (IJCAR), pp. 293–301, doi:10.1007/978-3-319-40229-1_20.

[9] Amy Felty, Carlos Olarte & Bruno Xavier (2021): *A Focused Linear Logical Framework and its Application to Metatheory of Object Logics*. Submitted to MSCS.

[10] Amy P. Felty & Alberto Momigliano (2012): *Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. J. Autom. Reason. 48(1), pp. 43–105, doi:10.1007/s10817-010-9194-x.

[11] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey*. J. Autom. Reason. 55(4), pp. 307–372, doi:10.1007/s10817-015-9327-3.

[12] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1-A Common Infrastructure for Benchmarks*. CoRR abs/1503.06095.

[13] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2018): *Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions*. Math. Struct. Comput. Sci. 28(9), pp. 1507–1540, doi:10.1017/S0960129517000093.

[14] Amy P. Felty & Brigitte Pientka (2010): *Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison*. In Matt Kaufmann & Lawrence C. Paulson, editors: Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings, Lecture Notes in Computer Science 6172, Springer, pp. 227–242, doi:10.1007/978-3-642-14052-5_17.

[15] Gerhard Gentzen (1969): *Investigations into Logical Deduction*. In M. E. Szabo, editor: The Collected Papers of Gerhard Gentzen, North-Holland, Amsterdam, pp. 68–131. Translation of articles that appeared in 1934-35.

[16] Rajeev Goré & Revantha Ramanayake (2012): *Valentini's cut-elimination for provability logic resolved*. The Review of Symbolic Logic 5, pp. 212–238, doi:10.1017/S1755020311000323. Available at `http://journals.cambridge.org/article_S1755020311000323`.

[17] Rajeev Goré, Revantha Ramanayake & Ian Shillito (2021): *Cut-elimination for provability logic by terminating proof-search: formalised and deconstructed using Coq*. In: 28^{th} International Conference on Automated Deduction (CADE-28). Accepted for publication.

[18] Stéphane Graham-Lengrand (2014): *Polarities & Focussing: a journey from Realisability to Automated Reasoning*. Habilitation thesis, Université Paris-Sud.

[19] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. J. ACM 40(1), p. 143–184, doi:10.1145/138027.138060.

[20] Bjoern Lellmann, Carlos Olarte & Elaine Pimentel (2017): *A uniform framework for substructural logics with modalities*. In Thomas Eiter & David Sands, editors: LPAR-21. 21st International Conference on Logic for

Programming, Artificial Intelligence and Reasoning, *EPiC Series in Computing* 46, EasyChair, pp. 435–455, doi:10.29007/93qg.

[21] Sonia Marin & Lutz Straßburger (2014): *Label-free Modular Systems for Classical and Intuitionistic Modal Logics*. In: *Advances in Modal Logic 10, invited and contributed papers from the tenth conference on "Advances in Modal Logic," held in Groningen, The Netherlands, August 5-8, 2014*, pp. 387–406.

[22] Raymond C. McDowell & Dale A. Miller (2002): *Reasoning with Higher-Order Abstract Syntax in a Logical Framework*. *ACM Trans. Comput. Logic* 3(1), p. 80–136, doi:10.1145/504077.504080.

[23] Dale Miller & Elaine Pimentel (2013): *A formal framework for specifying sequent calculus proof systems*. *Theoretical Computer Science* 474, pp. 98–116, doi:10.1016/j.tcs.2012.12.008.

[24] Vivek Nigam (2009): *Exploiting non-canonicity in the Sequent Calculus*. Ph.D. thesis, Ecole Polytechnique.

[25] Vivek Nigam, Elaine Pimentel & Giselle Reis (2016): *An extended framework for specifying and reasoning about proof systems*. *Journal of Logic and Computation* 26(2), pp. 539–576, doi:10.1093/logcom/exu029.

[26] Vivek Nigam, Giselle Reis & Leonardo Lima (2013): *Checking Proof Transformations with ASP*. In: *29th International Conference on Logic Programming (ICLP)*, 13.

[27] Vivek Nigam, Giselle Reis & Leonardo Lima (2014): *Quati: An Automated Tool for Proving Permutation Lemmas*. In: *7th International Joint Conference on Automated Reasoning (IJCAR 2014)*, pp. 255–261, doi:10.1007/978-3-319-08587-6_18.

[28] Carlos Olarte, Elaine Pimentel & Camilo Rocha (2018): *Proving Structural Properties of Sequent Systems in Rewriting Logic*. In Vlad Rusu, editor: *Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, Proceedings*, *Lecture Notes in Computer Science* 11152, Springer, pp. 115–135, doi:10.1007/978-3-319-99840-4_7.

[29] Luís Pinto & Tarmo Uustalu (2009): *Proof Search and Counter-Model Construction for Bi-intuitionistic Propositional Logic with Labelled Sequents*. In Martin Giese & Arild Waaler, editors: *Automated Reasoning with Analytic Tableaux and Related Methods: 18th International Conference, TABLEAUX 2009, Oslo, Norway, July 6-10, 2009. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 295–309, doi:10.1007/978-3-642-02716-1_22.

[30] Giselle Reis, Zan Naeem & Mohammed Hashim (2020): *Sequoia: A Playground for Logicians*. In Nicolas Peltier & Viorica Sofronie-Stokkermans, editors: *10th International Joint Conference on Automated Reasoning, (IJCAR)*, Springer International Publishing, pp. 480–488, doi:10.1007/978-3-030-51054-1_32.

[31] Hendrik Tews (2013): *Formalizing Cut Elimination of Coalgebraic Logics in Coq*. In Didier Galmiche & Dominique Larchey-Wendling, editors: *Automated Reasoning with Analytic Tableaux and Related Methods: 22nd International Conference, TABLEAUX 2013, Nancy, France, September 16-19, 2013, Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 257–272, doi:10.1007/978-3-642-40537-2_22.

[32] Christian Urban & Bozhi Zhu (2008): *Revisiting Cut-Elimination: One Difficult Proof Is Really a Proof*. In Andrei Voronkov, editor: *Rewriting Techniques and Applications: 19th International Conference, RTA 2008 Hagenberg, Austria, July 15-17, 2008 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 409–424, doi:10.1007/978-3-540-70590-1_28.

[33] Bruno Xavier, Carlos Olarte, Giselle Reis & Vivek Nigam (2018): *Mechanizing Focused Linear Logic in Coq*. *Electronic Notes in Theoretical Computer Science* 338, pp. 219 – 236, doi:10.1016/j.entcs.2018.10.014. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

Touring the MetaCoq Project (Invited Paper)

Matthieu Sozeau
Inria & LS2N, Université de Nantes
France
matthieu.sozeau@inria.fr

Proof assistants are getting more widespread use in research and industry to provide certified and independently checkable guarantees about theories, designs, systems and implementations. However, proof assistant implementations themselves are seldom verified, although they take a major share of the trusted code base in any such certification effort. In this area, proof assistants based on Higher-Order Logic enjoy stronger guarantees, as self-certified implementations have been available for some years. One cause of this difference is the inherent complexity of dependent type theories together with their extensions with inductive types, universe polymorphism and complex sort systems, and the gap between theory on paper and practical implementations in efficient programming languages. MetaCoq is a collaborative project that aims to tackle these difficulties to provide the first fully-certified realistic implementation of a type checker for the full calculus underlying the Coq proof assistant. To achieve this, we refined the sometimes blurry, if not incorrect, specification and implementation of the system. We show how theoretical tools from this community such as bidirectional type-checking, Tait-Martin-Löf/Takahashi's confluence proof technique and monadic and dependently-typed programming can help construct the following artifacts:

- a specification of Coq's syntax and type theory, the Polymorphic Cumulative Calculus of (Co)-Inductive Constructions (PCUIC);
- a monad for the manipulation of raw syntax and interaction with the Coq system;
- a verification of PCUIC's metatheory, whose main results are the confluence of reduction, type preservation and principality of typing;
- a realistic, correct and complete type-checker for PCUIC;
- a sound type and proof erasure procedure from PCUIC to untyped λ-calculus, i.e., the core of the extraction mechanism of Coq.

1 Introduction

Proof assistants have become invaluable tools in the hands of mathematicians, computer scientists and proof engineers that aim to build certified theories, software, systems and hardware, as evidenced by successful, large formalization projects ranging from famously hard mathematical results ([20],[23]) to realistic compilers ([33], [50]), program logics ([28, 5]), operating systems ([30, 21]) and even hardware design ([2, 36, 14]). Ultimately, however, all these formalizations rely on a Trusted Theory Base (TTB), that consists of the mathematical foundations of the proof-assistant -most often a variant of Higher-Order Logic, Set Theory or Type Theory- and a Trusted Code Base (TCB): its actual implementation in a general purpose programming language. To obtain the highest guarantees on the proof assistants results, one should in principle also verify the consistency of the foundations, usually by building models in a Trusted Theory (Zermelo-Fraenkel Set Theory being the most common one), the adequacy of the implementation with this theory, and the correct compilation of this implementation.

E. Pimentel and E. Tassi (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 13–29, doi:10.4204/EPTCS.337.2

1.1 A little history

To date, only the HOL family of provers have benefitted from such a guarantee, due to the seminal work of Kummar *et al* [31], who built a self-formalization of Higher-Order Logic modeled by set theory (ZF) in HOL Light (building on Harrison's work [24]), implemented itself in CakeML. In contrast, for dependent type theories at the basis of proof assistants like Coq, Agda, Idris or Lean, self-certification, or more informally type-theory eating itself [13] is a long-standing open problem. A rather large fragment of the theory at the basis of Coq, the Calculus of Constructions with universes, was formalized in Coq by Barras during his PhD [7] and extended in his habilitation thesis [8], culminating in a proof of Strong Normalization (SN) and hence relative consistency with IZF with a number of inaccessible cardinals for this theory. His development includes a proof of subject reduction and a model-theoretic proof using a specification of Intuitionistic Zermelo Fraenkel Set Theory in Type Theory, resulting from a long line of work relating the two [3, 54]. Due to Gödel's second incompleteness theorem, one can only hope to prove the consistency of the theory with n universes in a theory with $n+1$ universes. These results prove that a quite large fragment of Coq can be shown relatively consistent to a standard foundation. Since Barras' work, both pen-and-paper and formalized model-theoretic proofs have been constructed for many variants of dependent type theories, from decidability of type-checking for a type theory with universes [1] or canonicity [26] and normalization [48] for cubical type theories. We hence consider Coq's core type theory to be well-studied, a most recent reference for a consistency proof of the calculus with inductive types, universe polymorphism and cumulativity is [53, 52].

1.2 Goals of the project

The theory behind Coq's calculus, the Polymorphic Cumulative Calculus of (Co-)Inductive Constructions (PCUIC), is rather well-studied and well-tested now: most important fragments have accepted consistency proofs and no inconsistency was found in the rules that have been implemented in the last 30 years. That is, until a new axiom like Univalence breaks implicit assumptions in the calculus as happened recently for the guard checking algorithm. More worryingly, blatant inconsistencies (accepted proofs of False), are reported at least once a year, due to bugs in the implementation[1]. The source of these bugs falls generally under the category of programming errors and unforeseen interactions between subsystems. Coq is indeed a complex piece of software resulting from 37 years of research and development, incorporating many features in a single system. Its architecture was last revised in the V7 series (1999-2004) by Jean-Christophe Filliâtre [17, 16], following the de Bruijn criterion. It means that COQ does have a well-delimited, trustable proof-checking kernel, so these errors are not to be attributed to bad design in general. Rather, the problem is that this "small" kernel already comprises around 20kLoC of complex OCAML code + around 10kLoC of C code implementing a virtual machine for conversion. The system also relies on the whole OCaml compiler when used with the `native_compute` tactic for fast computation/conversion. To be fair (and grateful!), one should note that we never had to blame OCaml for a miscompilation resulting in an inconsistency. In conclusion, to avoid these errors, we should rather apply program verification to COQ's implementation.

This is a slightly different endeavor than the above works. Indeed, mechanized or not, the aforementionned proofs are concerned with idealized versions of the calculus that do not correspond to the actual implementation of PCUIC in OCaml, nor do they have any bearing on its compiled version, *a priori*. The METACOQ[2] project's goal is to bridge the gap between the model theoretic justification of the theory and

[1] https://github.com/coq/coq/blob/master/dev/doc/critical-bugs

[2] https://metacoq.github.io

the actual implementation of the Coq kernel. To do so, we need to answer the following questions in a formal, concrete way:

- What calculus is implemented by Coq exactly?
- Which meta-theoretical properties hold on the implementation?

To answer these questions, we develop a *specification* of Coq's type theory (§2), a Coq definition of a type-checker and conversion procedure that corresponds to the current Coq implementation, and verify both the sanity of the specification and correctness and completeness of the implementation.

Plan of the article. To verify the sanity of the specification we develop the meta-theory (§3) of the PCUIC calculus and show that it enjoys type preservation (§3.2.2) and principality (§3.2.4) of types, along with the expected confluence of reduction (§3.2.3) and proof that conversion is a congruence. We can then construct a corresponding type checker (§4) that is shown correct and complete with respect to the specification. Finally, to be able to execute this type-checker on large examples, we can extract it to OCaml and compare it to the type-checker implemented in Coq. Ideally, this last step should also be verified: we present a verified erasure procedure (§5) that takes a Coq environment and definition and produces a program in an (untyped) weak call-by-value λ-calculus extended with a dummy $\Box$ constructor ($\lambda_\Box$). We prove that erasure preserves observations of values, starting from a well-typed closed term. In combination with the CertiCoq compiler [4] from $\lambda_\Box$ to `C-light` and the CompCert [18] compiler from `C-light` to assembly, this will provide an end-to-end verified implementation of Coq's kernel. We discuss the remaining hurdles to realize this and further extensions in section 6.

Attribution The results surveyed in this article are due to MetaCoq team as a whole: Abhishek Anand, Dannil Annenkov, Simon Boulier, Cyril Cohen, Yannick Forster, Meven Lennon-Bertrand, Gregory Malecha, Jakob Botsch Nielsen, Matthieu Sozeau, Nicolas Tabareau and Théo Winterhalter.

Link to the formal development This article is best read online, as links in the text point to the formal development definition, which are generally too large to include in the presentation.

2 Syntax and Semantics

The MetaCoq project initially started as an extension of Gregory Malecha's Template-Coq plugin, developed during his PhD on efficient and extensible reflection in dependent type theory [37]. Template-Coq provided a *reification* of Coq's term and environment structures in Coq itself, i.e. Coq `Inductive` datatype declarations that model the syntax of terms and global declarations of definitions and inductive types, as close as possible to the OCaml definitions used in Coq's kernel. In addition, it implemented in an OCaml Coq plugin the meta-definitions of quoting and unquoting of terms between the two representations, similarly to Agda's reflection mechanism. The syntax of terms is given in figure 1. It corresponds closely to Coq's internal syntax, handling local variables (tRel, with a de Bruijn index), free named variables (tVar), existential variables (tEvar), sorts (tSort), the type-cast construct (tCast), dependent products, lambda-abstraction, let-ins and n-ary application (tApp), application of global references: constants, inductives, and constructors, a dependent case analysis construct (tCase, see 3.5 for more details) and primitive projections (tProj), fixpoints and co-fixpoints and finally, primitive integers and floating-point values (a recent [11] addition to Coq).

```
Inductive term : Type :=
| tRel (n : nat)
| tVar (id : ident) (* For free variables (e.g. in a goal) *)
| tEvar (ev : nat) (args : list term)
| tSort (s : Universe.t)
| tCast (t : term) (kind : cast_kind) (v : term)
| tProd (na : aname) (ty : term) (body : term)
| tLambda (na : aname) (ty : term) (body : term)
| tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
| tApp (f : term) (args : list term)
| tConst (c : kername) (u : Instance.t)
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info:predicate term)
        (discr:term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
| tInt (i : Int63.int)
| tFloat (f : PrimFloat.float).
```

Figure 1: Term syntax

On top of the term syntax, the MetaCoq.Template library also defines the type of local contexts which are lists of typing assumptions or local definitions, and global contexts: associative lists of global reference names to constant/axiom or inductive and constructors declarations.

2.1 The Template Monad

On top of these syntactic objects, one can define an API much like Coq's OCaml API to interact with the kernel of Coq: adding and looking up global definitions, calling the type-checker or higher-level primitives like proof-search for specific type-class instances. In [44], we show how this can be organized by defining a general TemplateMonad type-class that describes the structure of programs interacting with Coq's kernel. We need a monadic abstraction to encapsulate the handling of the global state of the Coq system that is being manipulated: updates to the global environment but also the obligation handling machinery of Program [43]. An interpreter for actions in this monad (e.g. adding a new definition with a given name and proof term) is meta-programmed in OCaml, using continuations to handle the interactive aspect of execution and maintaining coherence of the system state, similarly to the proof engine monad of the MTac2 [29] tactic language.

Using this monad, one can meta-program plugins in Coq that take input from the user, ask for user to provide definitions or proofs of particular types and update the environment from computed definitions. This can be used to define user-extensible translations [44, §4], or to derive lenses for a record definition [44, §5]. The TemplateMonad has a variant that also allows *extraction* of the monadic programs so they can be run efficiently in OCaml, putting it on par with the development of plugins for Coq directly in OCaml. MetaCoq plugins deriving parametricity theorems [44, §4.3.1] and induction principles and subterm relations from inductive definitions [12] can be defined this way, opening the possibility to verify their implementations. For parametricity for example, one can show that there is a procedure to construct from any well-typed term from Coq a proof that the parametricity predicate derived from its type holds on the term. As shown by Pédrot and Tabareau in [40], this can in turn be used to build internal syntactic models of richer, effectful type theories.

In the rest of this article, we will review how we built certified foundations needed for such efforts,

that is the typing specification and the type inference algorithm used to check well-typedness.

2.2 Typing, Reduction and Conversion

The calculus at the basis of Coq is the Polymorphic Cumulative Calculus of (Co-)Inductive Constructions (PCUIC). PCUIC is a general dependently-typed programming language, with pattern-matching and (co-)recursive definitions, universe polymorphic global declarations (constants or inductive types). Its origin is the Calculus of Constructions of Coquand and Huet [15] with $\beta\eta$-conversion, extended by the (Co-)Inductive type definition scheme of Paulin-Mohring [38], guarded (co-)fixpoint definitions [19], universe polymorphism [46] and cumulative inductive types [53]. The latest additions to the core calculus are a definitionally proof-irrelevant sort `SProp` and the addition of primitive types [6]. While they are supported in the syntax, they are not yet supported by our specification.

The sort system includes an infinite hierarchy of predicative universes `Type@{i}` ($i \in \mathbb{N}$) and an impredicative sort `Prop`. We consider `Set` to be a synonym for `Type@{0}`, hence its interpretation is always predicative[3]. A specificity of Coq is the treatment of `Prop` and in particular the singleton elimination rule that allows to eliminate propositional content into computational content, if it is trivial (a proof of absurdity, an equality, conjunction or accessibility proof): we will see in the section on erasure (5) how that is justified from the computational standpoint.

2.2.1 Conversion, Cumulativity

In dependent type theory, conversion and typing are usually intertwined, through the conversion rule which states that a term of type T can be seen as a term of type U for any (well-formed) typed U convertible to T. PCUIC is presented in the style of Pure Type Systems, where this conversion relation is untyped and can be defined independently on raw terms as the reflexive, symmetric and transitive closure of one-step reduction. We hence first define a reduction relation as an inductive predicate that includes all reduction rules of PCUIC: β for application, ι for cases, ζ for let-ins, `fix` and `cofix`, `delta` for constants and congruence rules allowing to apply reductions under any context (under lambdas, dependent products, etc). We take its closure with an additional twist: rather than a strict reflexivity rule, we define an α-equivalence relation that ignores the name annotations of binders (they are included in the core syntax for printing purposes only). Moreover, this relation is parameterized by a relation on universes to implement syntactic cumulativity of universes and inductive types. Indeed in Coq's theory we have:

$$\frac{i \leq j}{\texttt{Type@\{}i\texttt{\}} \leq \texttt{Type@\{}j\texttt{\}}}$$

A similar rule applies to cumulative inductive types. We call this relation α-cumulativity when instantiated with the large inequality of universes, in which case it is only a preorder. In case we instantiate it with equality of universes, we recover the usual α-conversion relation, which is an equivalence.

Two terms are hence in the cumulativity relation if they can be linked by reductions *or expansions* up-to α-cumulativity.

2.2.2 Typing

The typing relation of PCUIC is a fairly standard inductively defined relation that mostly corresponds to usual "on paper" treatments (e.g. Coq's reference manual [51]):

[3] Coq supports an `-impredicative-set` flag to switch to an impredicative interpretation, but it is seldom used today

```
Inductive typing (Σ : global_env_ext) (Γ : context) : term → term → Type
" Σ ; Γ ⊢ t : T " := (typing Σ Γ t T).
```

Figure 2: Type signature and notation for typing

The typing judgement is a predicate taking as parameters the global context extended with a universe declaration, a local context of assumptions and two terms corresponding to the subject and type of the typing rules. Derivations are defined in the Type sort to allow for easy inductions on the size of derivations. The typing rules are explained in detail in [44, §2] and [45][§2.2]. The typing rules are syntax-directed, i.e. there is one rule per head constructor of `term`, *except* for the cumulativity rule which can apply anywhere in a derivation. Note that we use a standard de Bruijn encoding for local variables, along with lifting and parallel substitution operations. As can be seen from the definition of figure 1, we used a nested datatype definition (`list term`, `list (branch term)`), hence some care must be taken to define recursive definitions and user-friendly induction principles on terms and derivations, lifting a predicate on terms to lists in the appropriate way. This is done by defining first a size measure on terms and then using well-founded induction on sizes of terms and derivations to derive easy to use induction principles.

Global environments Typing extends straightforwardly to local and global contexts. We formalize in particular which invariants should hold on the definition of inductive types, including strict positivity and the invariants enjoyed by cumulative inductive types. This is one point where we depart from the pen-and-paper treatments: indeed in [53], the theory that is studied is an idealization of Coq's implementation where equality is a judgement and inductive declarations do not carry parameters. In contrast, the implementation cannot rely on typing conditions to decide the cumulativity judgement and subtyping is rather defined on two different instances of the *same* inductive type, e..g `list@{Set} nat` and `list@{i} nat`. We hence had to retro-engineer, from Coq's OCaml code, a proper treatment of cumulative inductive types. We'll see in section 3.5 that this was a non-trivial endeavor.

2.3 Translation from Template to PCUIC

The tApp constructor represents n-ary application of a term f to a list of arguments *args*. This follows rather closely Coq's implementation, where the application node takes an array of arguments, for an even more compact representation of applications. Immediate access to the head of applications is an important optimization in practice, but this representation, while memory-efficient, imposes a hidden invariant on the term representation: the term f should not itself be an application, and the list of arguments should also always be non-empty. The application typing rule is likewise complicated as we have to consider application to a spine of arguments, rather than a single argument.

In Coq's kernel, this is handled by making the term type abstract and using smart constructors to enforce these invariants. Replicating this in Coq is tedious, as we have to either:

- work everywhere with an abstract/subset type of terms, precluding the use of primitive fixpoint and case-analysis
- or work with the raw syntax and add well-formedness preconditions everywhere

Our solution is to interface with Coq using the raw Template term syntax, keeping close to the implementation. To avoid dealing with well-formedness assumptions, we define a translation from this

syntax to the PCUIC term syntax where application is a binary constructor. We define similar reduction and typing judgments on the simplified syntax and show the equivalence of the two systems Template and PCUIC. This crucially relies on well-founded induction on the size of derivations to "reassociate" between binary applications and n-ary ones. The metatheory hereafter is developed on the more proof-friendly PCUIC syntax, but its theorems also apply to the original system. We simplify the PCUIC syntax further by removing the tCast constructor and translating it by an application of the identity function: this is observationally equivalent. The cast syntax in COQ is solely used to mark a place where a specific conversion algorithm should be used to convert the inferred type of a term with a specified expected type. This is used to call `vm_compute` or `native_compute` to perform the cumulativity check, rather than COQ's standard "call-by-need" conversion algorithm. As we do not formalize these fast conversion checks, this is not a loss. Note also that this has no bearing on the typeability of the term, *in theory*. Only *in practice* performing conversion with the default algorithm might be infeasible.

3 Metatheory

Now armed with the definition of typing and reduction in PCUIC, we can start proving the usual metatheoretical results of dependent type theories. We first derive the theory of our binding structures: the usual lemmas about de Bruijn operations of lifting and substitution are proven easily.

3.1 Digression on binding

Unfortunately, at the time we started the project (circa 2017), the Autosubst framework [41] could not be used to automatically derive this theory for us, due to the use of nested lists. We however learned from their work [47] and developed the more expressive σ-calculus, defined from operations of renaming (for a renaming $\mathbb{N} \to \mathbb{N}$) and instantiation (for a function $\mathbb{N} \to$ term), which provide a more proof-friendly interface to reason on the de Bruijn representation. We show that COQ's kernel functions of lifting and substitution (carrying just the number of crossed binders) are simulated with specific renaming and instantiation operations. Both variants are still of interest: it would be out of the question to use the σ-calculus operations which build closures while traversing terms in COQ's implementation. However the structured nature of the σ-calculus and its amenability to automation, having a decidable equational theory [47], is a clear advantage in practice.

One example where this shines is the treatment of (dependent) let-bindings in the calculus. Dependent let-bindings are an entirely different beast than ML-like let-bindings which can be simulated with abstraction and application. In particular, three rules of reduction can apply on local definitions:

$$\begin{array}{llll} \Gamma \vdash \texttt{let}\ x := t\ \texttt{in}\ b & \rightsquigarrow & b[t/x] & \zeta \\ \Gamma \vdash \texttt{let}\ x := t\ \texttt{in}\ b & \rightsquigarrow & \texttt{let}\ x := t\ \texttt{in}\ b' \quad \texttt{when}\ \Gamma, x := t \vdash b \rightsquigarrow b' & \texttt{cong-let-body} \\ \Gamma, x := t, \Delta \vdash x & \rightsquigarrow & \uparrow^{|\Delta|+1}(t) & \delta \end{array}$$

Here $\uparrow^n (t)$ represents the shifting of indices of the free variables of t by n, and $b[t/x]$ the usual capture-avoiding substitution. The first rule is usual let-reduction, the second is a congruence rule allowing to reduce under a `let` and the last allows to expand a local definition from the context. In the course of the metatheoretical development we must prove lemmas that allow to "squeeze" or smash the let-bindings in a context. This results in a reduced context with no let-bindings anymore and a corresponding substitution that mixes the properly substituted bodies corresponding to the removed let-bindings and regular variables, to apply to terms in the original context. This involves interchanging δ,

zeta and `cong-let-body` rules combined with the proper liftings and substitutions. This kind of reasoning appears in particular as soon as we want to invert an application judgment as let-bindings can appear anywhere in the type of the functional argument. Using σ-calculus reasoning and building the right abstractions tremendously helped simplify the proofs which would otherwise easily become indecipherable algebraic rewritings with the low level indices in liftings and substitutions.

3.2 Properties

3.2.1 Structural Properties

The usual Weakening and Substitution theorems can be proven straightforwardly by induction on the derivations. We use a beefed-up eliminator that automatically lifts the typing property we want to prove to well-formedness of the global environment, which also contains typing derivations. Likewise, we always prove properties simulataneously on well-formed local contexts and type derivations, so our theorems provide a conjunction of properties. When we moved to the σ-calculus represenation, we factorized these proofs by first stating renaming and instantiation lemmas, from which weakening and substitution follow as corrolaries. We also verify that typing is invariant by alpha-conversion, so name annotations on binders are indeed irrelevant.

3.2.2 Type Preservation

Proving subject reduction (a.k.a. type preservation) for dependent type theories can be rather difficult in a setting where definitional equality is typed, as it usually requires a logical relation argument/model construction, see e.g. [1]. However, the syntactic theory is relatively well-understood for PTS: one can first independently prove context conversion/cumulativity and injectivity of Π-types (i.e. $\Pi x : A.B \equiv \Pi x : A'.B' \rightarrow A \equiv A' \wedge B \equiv B'$), to prove type preservation in the application case. Similarly, we have injectivity of inductive type applications, up-to the cumulativity relation on universes.

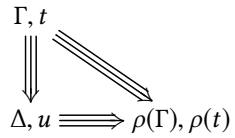
However, two other difficulties arise for PCUIC. First, we are considering a realistic type theory, will full-blown inductive family declarations, with a distinction between parameters and indices (that can contain let-bindings), and cumulative, universe polymorphic inductive types. To our knowledge, nobody ever attempted to formalize the proof at this level of detail before, even on paper. There is a good reason for that: the level of complexity is very high. Showing that the dependent case analysis reduction rule is sound mixes features such as let-bindings, the de Bruijn representation of binding adding various liftings and substitutions and the usual indigestible nesting of quantifications on indices in the typing rules of inductive types, constructors and branches. This is also ample reason to verify the code: many critical bugs a propos let-binding and inductive types were reported over the years. To tame this complexity, we tried to modularize the proof and provide the most abstract inversion lemmas on typing of inductive values and `match` constructs.

Second, COQ's theory is known to be broken regarding positive co-inductive types and subject reduction. We hence parameterize the proof: subject reduction holds only on judgments where no dependent case analysis is performed on co-inductive types. Negative co-inductive types implemented with primitive projections however can be show to enjoy subject reduction without restriction.

3.2.3 Confluence

To support inversion lemmas such as Π-type injectivity, we need to show that reduction is confluent. From this proof, it follows that the abstract, undirected conversion relation $T \equiv U$ is equivalent to reduc-

tion of the two sides to terms T' and U' that are in the syntactic α-cumulativity relation. We extend here Takahashi's refinement [49] of Tait/Martin-Löf's seminal proof of confluence for λ-calculus. The basic idea of the confluence proof is to consider parallel reduction instead of one-step reduction and prove the following triangle lemma:



The ρ function here is an "optimal" reduction function that reduces simultaneously all parallel redexes in a term or context. The fact that we need to consider contexts is due to let-bindings again: in one step of ρ, we might reduce a local definition to an abstraction, expand it in function position of an application and reduce the produced beta-redex. Using the triangle property twice, it is trivial to derive the diamond lemma and hence confluence for parallel reduction. By inclusion of one step reduction in parallel reduction, and parallel reduction in the transitive closure of one-step reduction (the "squashing" lemma), reduction is also confluent. This last part of reasoning is done abstractly, but accounting for the generalization to reduction in pairs of a context and a term. It then suffices to show commutation lemmas for reduction and α-cumulativity to show the equivalence of reduction up-to α-cumulativity and the undirected cumulativity relation. Confluence is crucial to show transitivity of the directed version.

Using this characterization of cumulativity, we can easily show that it is a congruence and that it enjoys expected inversion lemmas: if two Π-types are convertible, then they both reduce to Π-types that are in the α-cumulativity relation, so their domains are convertible and their codomains are in the cumulativity relation. Similarly, Π-types cannot be convertible to sorts or inductive types: there is no confusion between distinct type constructors.

3.2.4 Principality

As PCUIC is based on the notion of cumulativity rather than mere conversion, type uniqueness does not hold in the system. However, the refined property of principality does: for any context Γ and well-typed term t, there is a unique type T such that any other typing of t, $\Gamma \vdash t : U$ we have $T \leq U$. This property is essential for deciding type-checking: to check $\Gamma \vdash t : U$, it suffices to infer the principal type of t and check cumulativity. Principal typing is also used by the erasure procedure to take sound decisions based on the principal type of a given term.

3.3 Strengthening

The last expected structural property of the system is strengthening (a.k.a. thinning), which can be stated as:

$$\Gamma, x : A, \Delta \vdash t : T \rightarrow x \notin \mathtt{FV}(\Delta) \cup \mathtt{FV}(t) \cup \mathtt{FV}(T) \rightarrow \Gamma, \Delta \vdash t : T$$

This property ensures that typing is not influenced by unused variables, and is at the basis of the `clear` tactic of COQ, a quite essential building block in proof scripts. Unfortunately, this property *cannot* be proven by a simple induction on the typing derivation: the free-floating conversion rule allows to go through types mentioning the variables to clear, even if they do not appear in the term and type in the conclusion of the whole derivation.

3.4 Bidirectional Type Checking To The Rescue

This unfortunate situation can be resolved by reifying the principality property and type checking strategy as a bidirectional typing inductive definition. This variant of typing explicitly keeps track of the information flow in typing rules. In practice it separates the syntax-directed rules in a synthesis (a.k.a. inference) judgment (the term position is an input, while the type is an output) from the non-directed ones as checking rules (both positions are input). In [32], Meven Lennon-Bertrand develops a bidirectional variant for PCUIC, show equivalent to the orignal PCUIC, in which strengthening and principality become trivial to derive.

The crux of these argument is that bidirectional typing derivations are "canonical" for a given term, and move uses of the conversion rule to the top of the derivation, where they are strictly necessary. In addition, multiple cumulativity rules get "compressed" into a single change-of-phase rule, relying on transitivity of cumulativity. In a bidirectional synthesis derivation, if a variable does not appear in the term position, then it cannot appear in the inferred type. Simultaneously, in a checking derivation, if a variable does not appear in the term and type, then it cannot appear in these positions in any of the subderivations.

3.5 Case In Point

This detour through bidirectional typechecking is not accidental. In [45], we only proved the soundness of a typechecking algorithm for PCUIC (§4). It is in the course of formalizing the completeness of the type-checker (§4) that we discovered a problem in the typing rules of COQ. The problem appears in the dependent case analysis construct `match`. The gist of the typing rule was to typecheck the scrutinee at some unspecified inductive type $\mathsf{X}@\{\vec{u}\}\ \vec{p}\ \vec{i}$, where $\vec{u}$ is a universe instance,$\vec{p}$ the parameters and $\vec{i}$ the indices of the inductive family. The `match` construct also takes an elimination predicate, expected to be of type:

$$\Pi(\overrightarrow{x : I@[\vec{v}]}), \mathsf{X}@\{\vec{v}\}\ \vec{p'}\ \vec{x} \to \mathtt{Type}$$

Looking at this type, we would extract the universe instance $\vec{v}$ and parameters p' of the inductive X assumption. The typing rule checked that the universe instance of the scrutinee $\vec{u}$ was convertible to $\vec{v}$, rather than only in the cumulativity relation according to the subtyping rules for cumulative inductive types. It also compared the parameters $\vec{p}$ and $\vec{p'}$ for convertibility, by first lifting $\vec{p}$ in a context extended with the $\overrightarrow{x : I@[\vec{v}]}$ bindings, but these two instances did not necessarily live in the same type!

These mistakes lead to a loss of subject reduction, if cumulativity is used to lower the universes of the scrutinee, making the whole pattern-matching untypeable[4]. The problem appeared immediately while trying to prove completeness of type-checking, at the stage of designing the bidirectional typing rules: the flow of information was unclear and led us to the bug. We also realized that, to justify the comparison of the parameters, we would need to verify that $\vec{x} \notin \mathtt{FV}(\vec{p'})$ and apply strengthening, which as we have just seen is not directly provable on undirected typing rules. This motivated us to push for a change in the term representation of `match` in COQ[5] that solves both problems at once, by storing at the `match` node the universe instance and parameters that define the eliminator, and doing a sound cumulativity test of the infered type of the scrutinee and the (reconstructed) expected type of the eliminator. We are currently finishing to update the whole METACOQ development to handle this change of representation[6].

[4] https://github.com/coq/coq/issues/13495

[5] CEP 34 by Hugo Herbelin, Coq PR 13563 by Pierre-Marie Pédrot, integrated in Coq 8.14

[6] https://github.com/MetaCoq/metacoq/pull/534

4 A Type-Checker for PCUIC

4.1 Cumulativity

In [45, §3], we present a sound type-checker for PCUIC. To implement type-checking, we had to first develop a reasonably efficient reduction machine and algorithms to decide cumulativity. There are three separate algorithms at play to implement the cumulativity test.

Universes in COQ are floating variables subject to constraints [25], not directly natural numbers. To ensure consistency, one hence needs to verify that the constraints always have a valuation in the natural numbers. This boils down to deciding the (in-)equational theory of the tropical algebra $(\mathbb{N}, \mathtt{max}, +k, \leq)$. We develop a longest-simple-paths algorithm to check consistency of universe constraints: the valuation of each variable can be read off as the weight of its longest simple path from the source (`Type@{0}`). This is a naïve model and implementation of the state-of-the-art algorithm implemented in COQ, which derives from an incremental cycle detection algorithm [10] and whose formal verification is a work-in-progress [22]. Our specification is more expressive than COQ's current implementation, as it is able to handle arbitrary $\ell + k \leq \ell' + k'$ constraints between universe expressions, avoiding to distinguish so-called *algebraic* universes and implement costly universe refreshing operations when building terms. We hope to integrate this generalization back in COQ's implementation. Using this consistency check, it is easy to define α-cumulativity by structural recursion on terms.

Reduction We implement a weak-head reduction stack machine that can efficiently find the weak-head normal form of a term. To define this function, we must assume an axiom of strong normalization, which implies that reduction is well-founded on well-typed terms. This is the only axiom used in the development.

Conversion COQ uses a smart, mostly call-by-name, conversion algorithm, that uses performance-critical heuristics to decide which constants to unfold and when. COQ's algorithm does not naïvely reduce both terms to normal form to compare them, but rather incrementally reduces them to whnfs (without δ-reduction), compare their heads and recursively calls itself. When faced with the same defined constant on both sides, it first tries to unify their arguments before resorting to unfolding, resulting in faster successes but also potentially costly backtracking.

The main difficulty in the development of the conversion algorithm is that its termination and correctness are intertwined, so it is developed as a dependently-typed program that takes well-typed terms as arguments (ensuring termination of recursive calls assuming SN) and returns a proof of their convertibility (or non-convertibility). In other words it is proven sound and complete by construction. The design of the termination measure also involves a delicate construction of a dependent lexicographic ordering on terms in a stack due to Théo Winterhalter [55].

4.2 Type Checking

On top of conversion, implementing a type inference algorithm is straightforward: it is a simple structurally recursive function that takes well-formed global and local contexts and a raw term. It simply checks if the rule determined by the head of the term can apply. Figure 3 shows the type and beginning of the inference algorithm.

```
Program Fixpoint infer (Γ : context) (HΓ : ∥ wf_local Σ Γ ∥) (t : term) {struct t}
  : typing_result ({ A : term & ∥ Σ ;;; Γ |- t : A ∥ }) :=
  match t with
  | tRel n ⇒
        match nth_error Γ n with
        | Some c ⇒ ret ((lift0 (S n)) (decl_type c); _)
        | None ⇒ raise (UnboundRel n)
        end
```

Figure 3: Type inference function excerpt

Again, the function is strongly typed: its result lives in the typing_result monad, which is an exception monad, returning (ret) a sigma-type of an inferred type A and a "squashed" proof that the term has this type or failing with type error (raise). As all our derivations are in Type by default, we use explicit squashing into Prop when writing programs manipulating terms:

```
Record squash (A : Type) : Prop := sq { _ : A }.

Notation "∥ T ∥" := (squash T) (at level 10).
```

The elimination rules for propositional inductives ensures that our programs cannot make computational choices based on the shape of the squashed derivations, and that extraction will remove these arguments. The extracted version of infer hence only takes a context (assumed to be well-formed) and a term and returns an error or an infered type, just like in Coq's implementation.

Using the bidirectional presentation of the system, we can simplify the correctness and completeness proof in [45] as the algorithm really follows the same structure as bidirectional derivations. Typechecking is simply defined as type inference followed by a conversion test, as usual.

4.3 Verifying Global Environments

Once the type-checker for terms is defined, we can lift it to verify global environment declarations. For constants and axioms, this is straightforward. However, declarations of inductive types are more complex and require to first define a sound context cumulativity test, a strict positivity check and to turn the universe constraints into a graph structure. This is again done using a monad EnvCheck:

```
Program Fixpoint check_wf_env (Σ : global_env)
  : EnvCheck (∑ G, (is_graph_of_uctx G (global_uctx Σ) ∧ ∥ wf Σ ∥))
```

Given a global environment Σ, this produces either an error or a pair of a graph and a proof that the universe graph models the constraints in Σ and a (squashed) proof that the environment is well-formed.

5 Erasure from PCUIC to λ-calculus

The type-checker can be extracted to OCaml and run on reasonably large programs. For example it can be used to successfully check the prelude of the HoTT library [9], including a large proof that adjoint equivalences can be promoted to homotopy equivalences. However, our first attempt to extraction was unsuccessful: we had to change the Coq definitions so that OCaml could typecheck the generated code, as we hit a limitation of the extraction mechanism in presence of dependently-typed variadic functions. The obvious next step was hence to verify the erasure procedure itself!

In [45, §4], we present a sound erasure procedure from PCUIC to untyped, call-by-value λ-calculus. This corresponds to the first part of COQ's extraction mechanism [34], which additionally tries to maintain simple types corresponding to the original program. Erausre is performed by a single traversal of the term, expected to be well-typed. It checks if the given subterm is a type (its type is a sort `Prop` or `Type`) or if it is a proof of a proposition (its type P has sort `Prop`), in which case it returns a dummy $\Box$ term, and otherwise proceeds recursively, copying the structure of the original term. The result of erasure hence contains no type information anymore, and all propositional content is replaced with $\Box$.

We can prove the following correctness statement on this procedure:

```
Lemma erases_correct Σ t T t' v Σ' :
  extraction_pre Σ →
  Σ;;; [] |- t : T →
  Σ;;; [] |- t ⇝ℰ t' →
   erases_global Σ Σ' →
  Σ;;; [] |- t ▷ v →
  ∃ v', Σ;;; [] |- v ⇝ℰ v' ∧ Σ' ⊢ t' ▷ v'.
```

Our correctness theorem shows that if we have a well-typed term t and t erases to t', then if t reduces to a *value* v using weak-cbv reduction, then the erased term t' also reduces to an observationally equivalent value v'. The `extraction_pre` precondition enforces that the environment is well-formed. The proof follows Letouzey's original pen-and-paper proof closely [35]. Since [45], we proved two additional verified passes of optimization on the erased term and environment:

- We remove from Σ' the definitions that are not used for evaluation, pruning the environment from useless declarations that are no longer needed for the computation.
- We remove dummy pattern-matchings on $\Box$ terms, that should always trivially reduce to their single branch.

The end result of erasure is an untyped term that contains only the raw computational content of the original definition. It can be further compiled with the CERTICOQ compiler and COMPCERT to produce certified assembly code from COQ definitions.

6 Going further

We have presented the whole METACOQ project, which spans from the reification of COQ terms down to the erasure of well-typed terms to untyped λ-calculs. The whole project weights about 100kLoC of OCAML and COQ code, and is still under active development. We think this is a convincing example that we can move from a Trusted Code Base consisting of COQ's unverified kernel down to a Trusted Theory Base that consists of the formalized typing rules of PCUIC and its axiom of Strong Normalization.

The METACOQ (and CERTICOQ) projects are both ongoing work subject to limitations, we summarize here the current state of affairs.

6.1 Limitations

While PCUIC models a large part of COQ's implementation, it still misses a few impotant features of the theory:

- The η-conversion rule is not supported in our formalization, preventing us to check most of the standard library. Dealing with η rules in an untyped conversion setting is a notoriously hard

issue. We are however hopeful that we found a solution to this problem by quotienting definitional equality with η-reduction, and hope to present this result soon.

- Similarly, we do not handle the new `SProp` sort of Coq. Our methodology for η-conversion should however also apply for this case.

- We do not formalize yet the guard-checking of fixpoint and co-fixpoint definitions, relying instead on oracles. Our strong normalization assumption hence includes an assumption of correctness of the guard checkers. We are currently working on integrating a definition of the guard checking algorithm and verifying its basic metatheory (invariance by renaming, substitution, etc.).

- We did not consider the module system of Coq, which is mostly orthogonal to the core typing algorithm but represents a significant share of Coq's kernel implementation, we leave this to future work.

- We leave out the so-called "template"-polymorphism feature of Coq, which is a somewhat fragile (i.e. prone to bugs), non-modular alternative to cumulative inductive types. This prevents us from checking most of the Coq standard library today as it makes intensive use of this feature. We are working with the Coq development team to move the standard library to universe polymorphism to sidestep this issue.

6.2 Conclusion and Perspectives

There are many directions in which we consider to extend the project:

- On the specification side we would like to link our typing judgment to the "Coq en Coq" formalization of Barras [8], which provides the Strong Normalization proof we are assuming, for a slightly different variant of the calculus. This formalization is based on a sized-typing discipling for inductive types, which will require to show an equivalence with Pcuic's guardness checker, or an update of Pcuic to handle sized typing altogether.

- Proving that the theory is equivalent to a variant where conversion is typed, i.e. definitional equality is a judgment would also make our theory closer to categorical models of type theory, e.g., Categories with Families. This can build on recent results in this direction by Siles and Herbelin [42], updating them to handle cumulativity.

- In addition to the parametricity translation that we would like to prove correct, many syntactic models of type theory, extending it with side-effects [39] or forcing [27] have recently been developed. MetaCoq is the right setting to mechanize these results.

- We have concentrated our verification efforts on the core type-checking algorithm of Coq, but higher-level components like unification, elaboration and the proof engine would also benefit from formal treatment. We hope to tackle these components in the future.

- Finally, on the user side, we are still at the beginning of the exploration of the meta-programming features of MetaCoq. It could be used to justify for example the foundations of the MTac 2 language [29], to turn the typed tactic language into a definitional extension of Coq's theory.

7 Bibliography

References

[1] Andreas Abel, Joakim Öhman & Andrea Vezzosi (2018): *Decidability of conversion for type theory in type theory*. *PACMPL* 2(POPL), pp. 23:1–23:29, doi:10.1145/3158111.

[2] Behzad Akbarpour, Amr T. Abdel-Hamid, Sofiène Tahar & John Harrison (2010): *Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL*. *Comput. J.* 53(4), pp. 465–488, doi:10.1093/comjnl/bxp023.

[3] Thorsten Altenkirch (1993): *Constructions, Inductive Types and Strong Normalization*. Ph.D. thesis, University of Edinburgh.

[4] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau & Matthew Weaver (2017): *CertiCoq: A verified compiler for Coq*. In: *CoqPL*, Paris, France.

[5] Andrew W. Appel (2014): *Program Logics - for Certified Compilers*. Cambridge University Press, doi:10.1017/CBO9781107256552.

[6] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack & Laurent Théry (2010): *Extending Coq with Imperative Features and Its Application to SAT Verification*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving*, Springer, pp. 83–98, doi:10.1016/j.jal.2007.07.003.

[7] Bruno Barras (1999): *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7.

[8] Bruno Barras (2012): *Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families*. Unpublished.

[9] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau & Bas Spitters (2017): *The HoTT library: a formalization of homotopy type theory in Coq*. In Yves Bertot & Viktor Vafeiadis, editors: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, ACM, pp. 164–172, doi:10.1145/3018610.3018615.

[10] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert & Robert E. Tarjan (2016): *A New Approach to Incremental Cycle Detection and Related Problems*. *ACM Trans. Algorithms* 12(2), pp. 14:1–14:22, doi:10.1145/2756553.

[11] Guillaume Bertholon, Érik Martin-Dorel & Pierre Roux (2019): *Primitive Floats in Coq*. In John Harrison, John O'Leary & Andrew Tolmach, editors: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, *LIPIcs* 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 7:1–7:20, doi:10.4230/LIPIcs.ITP.2019.7.

[12] Marcel Ullrich Bohdan Liesnikov & Yannick Forster (2020): *Generating induction principles and subterm relations for inductive types using MetaCoq*. *The Coq Workshop 2020*.

[13] James Chapman (2009): *Type Theory Should Eat Itself*. *Electron. Notes Theor. Comput. Sci.* 228, pp. 21–36, doi:10.1016/j.entcs.2008.12.114.

[14] Adam Chlipala (2020): *Proof assistants at the hardware-software interface (invited talk)*. In Jasmin Blanchette & Catalin Hritcu, editors: *CPP 2020*, ACM, p. 2, doi:10.1145/3372885.3378575.

[15] Thierry Coquand & Gérard Huet (1988): *The Calculus of Constructions*. *Information and Computation* 76(2–3), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[16] Jean-Christophe Filliâtre (2000): *Design of a proof assistant: Coq version 7*. Research Report, Université Paris-Sud.

[17] Jean-Christophe Filliâtre (2020): *A Coq retrospective, at the heart of Coq architecture, the genesis of version 7.0*. Invited talk at the Coq Workshop 2020.

[18] Gallium, Marelle, CEDRIC & PPS (2008): *The CompCert project*. Compilers You Can *Formally* Trust.

[19] Eduardo Giménez (1998): *Structural Recursive Definitions in Type Theory*. In Kim Guldstrand Larsen, Sven Skyum & Glynn Winskel, editors: *ICALP*, *LNCS* 1443, Springer, pp. 397–408.

[20] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *ITP 2013*, *LNCS* 7998, Springer, pp. 163–179, doi:10.1007/978-3-642-39634-2_14.

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg & David Costanzo (2016): *CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels*. In Kimberly Keeton & Timothy Roscoe, editors: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, USENIX Association, pp. 653–669, doi:10.5555/3026877.3026928.

[22] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud & François Pottier (2019): *Formal Proof and Analysis of an Incremental Cycle Detection Algorithm*. In: *ITP 2019 - 10th Conference on Interactive Theorem Proving*, Portland, United States.

[23] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu & Roland Zumkeller (2015): *A formal proof of the Kepler conjecture*. *CoRR* abs/1501.02155.

[24] John Harrison (2006): *Towards self-verification of HOL Light*. In Ulrich Furbach & Natarajan Shankar, editors: *Proceedings of the third International Joint Conference, IJCAR 2006*, *LNCS* 4130, Springer-Verlag, Seattle, WA, pp. 177–191.

[25] Hugo Herbelin (2005): *Type Inference with Algebraic Universes in the Calculus of Inductive Constructions*. Manuscript.

[26] Simon Huber (2019): *Canonicity for Cubical Type Theory*. *Journal of Automated Reasoning* 63(2), pp. 173–210, doi:10.1007/s10817-018-9469-1.

[27] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau & Nicolas Tabareau (2016): *The Definitional Side of the Forcing*. In Martin Grohe, Eric Koskinen & Natarajan Shankar, editors: *LICS '16*, ACM, pp. 367–376, doi:10.1145/2933575.2935320.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers & Derek Dreyer (2021): *Safe systems programming in Rust*. *Commun. ACM* 64(4), pp. 144–152, doi:10.1145/3418295.

[29] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas & Derek Dreyer (2018): *Mtac2: typed tactics for backward reasoning in Coq*. *PACMPL* 2(ICFP), pp. 78:1–78:31, doi:10.1145/3236773.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: formal verification of an OS kernel*. In Jeanna Neefe Matthews & Thomas E. Anderson, editors: *SOSP*, ACM, pp. 207–220, doi:10.1145/1629575.1629596.

[31] Ramana Kumar, Rob Arthan, Magnus O. Myreen & Scott Owens (2016): *Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation*. *J. Autom. Reason.* 56(3), pp. 221–259, doi:10.1007/s10817-015-9357-x.

[32] Meven Lennon-Bertrand (2021): *Complete Bidirectional Typing for the Calculus of Inductive Constructions*. In Liron Cohen & Cezary Kaliszyk, editors: *ITP 2021*, *LIPIcs* 193, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:19, doi:10.4230/LIPIcs.ITP.2021.24.

[33] Xavier Leroy (2006): *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*. In: *33rd symposium Principles of Programming Languages*, ACM Press, pp. 42–54.

[34] Pierre Letouzey (2002): *A New Extraction for Coq*. In Herman Geuvers & Freek Wiedijk, editors: *TYPES'02*, *LNCS* 2646, Springer, pp. 200–219.

[35] Pierre Letouzey (2004): *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud.

[36] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson & Anthony Fox (2019): *Verified Compilation on a Verified Processor*. In: *PLDI 2019*, ACM, New York, NY, USA, pp. 1041–1053, doi:10.1145/3314221.3314622.

[37] Gregory Michael Malecha (2014): *Extensible Proof Engineering in Intensional Type Theory*. Ph.D. thesis, Harvard University.

[38] Christine Paulin-Mohring (1993): *Inductive Definitions in the System Coq - Rules and Properties*. In Marc Bezem & Jan Friso Groote, editors: *Typed Lambda Calculi and Applications*, doi:10.1007/BFb0037116.

[39] Pierre-Marie Pédrot & Nicolas Tabareau (2017): *An effectful way to eliminate addiction to dependence*. In: *LICS 2017*, IEEE Computer Society, pp. 1–12, doi:10.1109/LICS.2017.8005113.

[40] Pierre-Marie Pédrot & Nicolas Tabareau (2020): *The fire triangle: how to mix substitution, dependent elimination, and effects*. *Proc. ACM Program. Lang.* 4(POPL), pp. 58:1–58:28, doi:10.1145/3371126.

[41] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*. In Christian Urban & Xingyuan Zhang, editors: *ITP 2015*, *LNCS* 9236, Springer, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[42] Vincent Siles & Hugo Herbelin (2012): *Pure Type System conversion is always typable*. *J. Funct. Program.* 22(2), pp. 153–180, doi:10.1017/S0956796812000044.

[43] Matthieu Sozeau (2007): *Subset Coercions in Coq*. In Thorsten Altenkirch & Conor McBride, editors: *TYPES'06*, *LNCS* 4502, Springer, pp. 237–252, doi:10.1007/978-3-540-74464-1_16.

[44] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau & Théo Winterhalter (2020): *The MetaCoq Project*. *Journal of Automated Reasoning* 64(5), pp. 947–999, doi:10.1007/s10817-019-09540-0.

[45] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau & Théo Winterhalter (2020): *Coq Coq Correct! Verifying Typechecking and Erasure for Coq, in Coq*. *Proceedings of the ACM on Programming Languages* 4(POPL), doi:10.1145/3371076.

[46] Matthieu Sozeau & Nicolas Tabareau (2014): *Universe Polymorphism in Coq*. In Gerwin Klein & Ruben Gamboa, editors: *ITP 2014*, *LNCS* 8558, Springer, pp. 499–514, doi:10.1007/978-3-319-08970-6_32.

[47] Kathrin Stark, Steven Schäfer & Jonas Kaiser (2019): *Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions*. In Assia Mahboubi & Magnus O. Myreen, editors: *CPP 2019*, ACM, pp. 166–180, doi:10.1145/3293880.3294101.

[48] Jonathan Sterling & Carlo Angiuli (2021): *Normalization for Cubical Type Theory*. *CoRR* abs/2101.11479.

[49] Masako Takahashi (1995): *Parallel Reductions in lambda-Calculus*. *Inf. Comput.* 118(1), pp. 120–127, doi:10.1006/inco.1995.1057.

[50] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens & Michael Norrish (2019): *The verified CakeML compiler backend*. *J. Funct. Program.* 29, p. e2, doi:10.1017/S0956796818000229.

[51] The Coq Development Team (2021): *The Coq Proof Assistant*, doi:10.5281/zenodo.4501022.

[52] Amin Timany & Matthieu Sozeau (2017): *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*. Research Report RR-9105, KU Leuven, Belgium ; Inria Paris.

[53] Amin Timany & Matthieu Sozeau (2018): *Cumulative Inductive Types In Coq*. In Hélène Kirchner, editor: *FSCD*, *LIPIcs* 108, pp. 29:1–29:16, doi:10.4230/LIPIcs.FSCD.2018.29.

[54] Benjamin Werner (1997): *Sets in types, types in sets*. In Martín Abadi & Takayasu Ito, editors: *Theoretical Aspects of Computer Software*, Springer, pp. 530–546, doi:10.1007/BFb0014566.

[55] Théo Winterhalter (2020): *Formalisation and Meta-Theory of Type Theory*. Ph.D. thesis, Université de Nantes. 2020NANT4012.

Interacting Safely with an Unsafe Environment

Gilles Dowek
Inria and ENS Paris-Saclay
gilles.dowek@ens-paris-saclay.fr

We give a presentation of Pure type systems where contexts need not be well-formed and show that this presentation is equivalent to the usual one. The main motivation for this presentation is that, when we extend Pure type systems with computation rules, like in the logical framework DEDUKTI, we want to declare the constants before the computation rules that are needed to check the well-typedness of their type.

1 Introduction

In the simply typed lambda-calculus, to assign a type to a term, we first need to assign a type to its free variables. For instance, if we assign the type $nat \to nat \to nat$ to the variable f and the type nat to the variable x, then we can assign the type $nat \to nat$ to the term $\lambda y : nat\ (f\ x\ y)$.

Whether a type is assigned to f before or after one is assigned to x is immaterial, so the context $\{f : nat \to nat \to nat, x : nat\}$ does not need to be ordered.

1.1 Well-formed Contexts

In systems, such as the Calculus of constructions, where atomic types are variables of a special type $*$, contexts are ordered and, for instance, the term $\lambda y : nat\ (f\ x\ y)$ is assigned the type $nat \to nat$ in the context $nat : *, f : nat \to nat \to nat, x : nat$ but not in the context $x : nat, nat : *, f : nat \to nat \to nat$, that is not well-formed.

In a well-formed context, the declarations are ordered in such a way that the type of a variable only contains variables declared to its left. For instance, the context $nat : *, z : nat, array : nat \to *, nil : (array\ z)$ is well-formed, but the context $nat : *, z : nat, nil : (array\ z), array : nat \to *$ is not. Moreover, in such a well-formed context, each type is itself well-typed in the context formed with the variable declarations to its left. For instance, the context $nat : *, array : nat \to *, z : nat, nil : (array\ z\ z)$ is not well-formed. So, a context $x_1 : A_1, ..., x_n : A_n$ is said to be well-formed if, for each i, $x_1 : A_1, ..., x_i : A_i \vdash A_{i+1} : s$ is derivable for some sort s in $\{*, \Box\}$.

The original formulation of the Calculus of constructions of Coquand and Huet [4] has two forms of judgements: one expressing that a context Γ is well-formed and another expressing that a term t has a type A in a context Γ. Two rules define when a context is well-formed

$$\frac{}{[\,]\ \text{well-formed}}\ (\text{empty})$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A\ \text{well-formed}}\ (\text{decl})\ s \in \{*, \Box\}$$

and one enables the assignment of a type to a variable, in a well-formed context

$$\frac{\Gamma, x : A, \Gamma'\ \text{well-formed}}{\Gamma, x : A, \Gamma' \vdash x : A}\ (\text{var})$$

E. Pimentel and E. Tassi (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 30–38, doi:10.4204/EPTCS.337.3

These three rules together with five others—(sort), (prod), (abs), (app), and (conv)—form a eight-rule presentation of the Calculus of constructions, and more generally of Pure type systems. Because the rule (var) requires the context $\Gamma, x : A, \Gamma'$ to be well-formed, a variable can only be assigned a type in a well-formed context and this property extends to all terms, as it is an invariant of the typing rules.

This system was simplified by Geuvers and Nederhof [7] and Barendregt [1], who use a single form of judgement expressing that a term t has a type A in a context Γ. First, they drop the context Γ' in the rule (var) simplifying it to

$$\frac{\Gamma, x : A \text{ well-formed}}{\Gamma, x : A \vdash x : A}$$

and add a weakening rule

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma, x : B \vdash t : A} \text{(weak)}$$

to extend the judgement $\Gamma, x : A \vdash x : A$ to $\Gamma, x : A, \Gamma' \vdash x : A$. Then, they exploit the fact that the conclusion of the rule (decl) is now identical to the premise of the rule (var), to coin a derived rule

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \text{(start) } s \in \{*, \Box\}$$

Now that the variables can be typed without using a judgement of the form Γ well-formed, such judgements can be dropped, together with the rules (empty), (decl), and (var). So the two rules (start) and (weak), together with the five other rules form an equivalent seven-rule formulation of the Calculus of constructions, and more generally of Pure type systems.

1.2 Interacting Safely with an Unsafe Environment

When a judgement of the form $\Gamma \vdash x : A$ is derived, the well-typedness of the term A needs to be checked. But it can be checked either when the variable x is added to the context or when it is used in the derivation of the judgement $\Gamma \vdash x : A$. In the system with the rules (decl) and (var), it is checked in the rule (decl), that is when the variable is added to the context. When the rule (var) is replaced with the rule (start), it is checked when the variable is used. These two systems illustrate two approaches to safety: the first is to build a safe environment, the second is to interact safely with a possibly unsafe environment.

In the formulation of Geuvers and Nederhof and Barendregt, it is still possible to define a notion of well-formed context: the context $x_1 : A_1, ..., x_n : A_n$ is well-formed if for each i, $x_1 : A_1, ..., x_i : A_i \vdash A_{i+1} : s$ is derivable. With such a definition, it is possible to prove that if the judgement $\Gamma \vdash t : A$ is derivable, then Γ is well-formed. In this proof, the second premise of the rule (weak), $\Gamma \vdash B : s$, is instrumental, as its only purpose is to preserve the well-formedness of the context.

We can go further with the idea of interacting safely with an unsafe environment and drop this second premise, leading to the weakening rule

$$\frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t : A}$$

Then, in the judgement $\Gamma \vdash x : A$, nothing prevents the context Γ from being non well-formed, but the term A is still well-typed because the rule (start), unlike the rule (var), has a premise $\Gamma \vdash A : s$. In such a system, the judgement $nat : *, array : nat \to *, z : nat, nil : (array\ z\ z) \vdash z : nat$ is derivable, although the term $(array\ z\ z)$ is not well-typed, but the judgement $nat : *, array : nat \to *, z : nat, nil : (array\ z\ z) \vdash nil : (array\ z\ z)$ is not because this term $(array\ z\ z)$ is not well-typed.

$$\frac{}{\Gamma \vdash s_1 : s_2}\,(\text{sort'}) \quad \langle s_1, s_2\rangle \in \mathscr{A}$$

$$\frac{\Gamma, x : A, \Gamma' \vdash A : s}{\Gamma, x : A, \Gamma' \vdash x : A}\,(\text{var'}) \quad x \in \mathscr{V}_s$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (x : A) \to B : s_3}\,(\text{prod}) \quad \langle s_1, s_2, s_3\rangle \in \mathscr{R}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\ t : (x : A) \to B}\,(\text{abs}) \quad \langle s_1, s_2, s_3\rangle \in \mathscr{R}$$

$$\frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : (u/x)B}\,(\text{app})$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\,(\text{conv}) \quad A \equiv B$$

Figure 1: Pure type systems with arbitrary contexts

Yet, with the rule (start) and this strong weakening rule, the judgement $nat : *, z : nat, nil : (array\ z)$, $array : nat \to * \vdash nil : (array\ z)$ is not derivable, because the judgement $nat : *, z : nat \vdash (array\ z) : *$ is not derivable. Thus, to make this judgement derivable, we should not use a weakening rule that erases all the declarations to the right of the declaration of *nil* and the rule (start). But we should instead use a rule that keeps the full context to type the term $(array\ z)$. Yet, like the rule (start), this rule should not check that the context is well-formed, but that the type of the variable is a well-typed term

$$\frac{\Gamma, x : A, \Gamma' \vdash A : s}{\Gamma, x : A, \Gamma' \vdash x : A}\,(\text{var'})$$

This leads to the six-rule system described in Figure 1.

As the order of declarations in a context is now immaterial, contexts can indifferently be defined as sequences or as sets of declarations.

1.3 Previous Work

There are several reasons for using arbitrary contexts. One of them is that, as already noticed by Sacerdoti Coen [3], when we have two contexts Γ and Γ', for instance developed by different teams in different places, and we want to merge them, we should not have to make a choice between Γ, Γ', and Γ', Γ. We should just be able to consider the unordered context $\Gamma \cup \Gamma'$, provided it is a context, that is if $x : A$ is declared in Γ and $x : A'$ is declared in Γ' then $A = A'$.

Another is that, when we extend Pure type systems with computation rules, like in the logical framework DEDUKTI, we additionally want to declare constants in a signature Σ and then add computation rules. For instance, we want to be able to declare constants in a signature $\Sigma = nat : *, a : nat, b : nat, P : nat \to *, Q : (P\ a) \to *, e : (P\ b), h : (Q\ e), c : nat$ and then computation rules $a \longrightarrow c$, $b \longrightarrow c$. Because, unlike in [5], the term $(Q\ e)$ is not well-typed without the computation rules, we cannot check the that the signature is well-formed before we declare the rules. But, because the rules use the constants declared in Σ, we cannot declare the rules before the signature, in particular the rules do not make sense in the part of the signature to the left of the declaration of h, that is in $nat : *, a : nat, b : nat, P : nat \to *, Q : (P\ a) \to$

$*, e : (P\ b)$, where the constant c is missing. And, because we sometimes want to consider rules $l \longrightarrow r$ where l and r are not well-typed terms [2], we cannot interleave constant declarations and computation rules. Note that in Blanqui's Calculus of algebraic constructions [2], the contexts are required to be well-formed, but the signatures are not.

Another source of inspiration is the presentation of Pure type systems without explicit contexts [6], where Geuvers, Krebbers, McKinna, and Wiedijk completely drop contexts in the presentation of Pure type systems. In particular, Theorem 3.1 below is similar to their Theorem 19. The presentation of Figure 1 is however milder than their Pure type systems without explicit contexts, as it does not change the syntax of terms, avoiding, for instance, the question of the convertibility of x^B and $x^{(\lambda \dot{A}:*\dot{A})\ B}$. In particular, if $\Gamma \vdash t : A$ is derivable in the usual formulation of Pure type systems, it is also derivable in the system of Figure 1.

We show, in this note, that the system presented in Figure 1 indeed allows to interact safely with an unsafe environment, in the sense that if a judgement $\Gamma \vdash t : A$ is derivable in this system, then there exists Δ, such that $\Delta \subseteq \Gamma$ and $\Delta \vdash t : A$ is derivable with the usual Pure type system rules. The intuition is that, because of the rule (var'), the structure of a derivation tree induces a dependency between the used variables of Γ that is a partial order, and as already noticed by Sacerdoti Coen [3], a topological sorting of the used variables yields a linear context Δ. Topological sorting is the key of Lemma 3.4.

So this paper build upon the work of Coquand and Huet [4], Geuvers and Nederhof [7], Barendregt [1], Blanqui [2], Sacerdoti Coen [3], and Geuvers, Krebbers, McKinna, and Wiedijk [6]. Its main contribution is to show that Pure type systems can be defined with six rules only, without a primitive notion of well-formed context, and without changing the syntax of terms.

2 Pure Type Systems

Let us first recall a usual definition of (functional) Pure type systems [7, 1]. To define the syntax of terms, we consider a set $\mathscr{S}$ of sorts and a family of $\mathscr{V}_s$ of infinite and disjoint sets of variables of sort s. The syntax is then

$$t = x \mid s \mid (x : A) \to B \mid \lambda x : A\ t \mid t\ u$$

A context Γ is a sequence $x_1 : A_1, ..., x_n : A_n$ of pairs formed with a variable and a term, such that the variables $x_1, ..., x_n$ are distinct. So, when we write the context $\Gamma, y : B$, we implicitly assume that y is not already declared in Γ.

A context Γ is said to be included into a context Γ' ($\Gamma \subseteq \Gamma'$) if every $x : A$ in Γ is also in Γ'.

Two contexts Γ and Γ' are said to be compatible if each time $x : A$ is in Γ and $x : A'$ is in Γ', then $A = A'$.

To define the typing rule, we consider a set $\mathscr{A}$ of axioms, that are pairs of sorts and a set $\mathscr{R}$ of rules, that are triple of sorts. As we restrict to functional Pure type systems, we assume that the relations $\mathscr{A}$ and $\mathscr{R}$ are functional.

Definition 2.1 (The type system $\mathscr{T}$)

$$\frac{}{\vdash s_1 : s_2}\,(sort) \quad \langle s_1, s_2 \rangle \in \mathscr{A}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\,(start) \quad x \in \mathscr{V}_s$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma, x : B \vdash t : A}\,(weak) \quad x \in \mathscr{V}_s$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (x : A) \to B : s_3}\,(prod) \quad \langle s_1, s_2, s_3 \rangle \in \mathscr{R}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\ t : (x : A) \to B}\,(abs) \quad \langle s_1, s_2, s_3 \rangle \in \mathscr{R}$$

$$\frac{\Gamma \vdash t : (x : A) \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : (u/x)B}\,(app)$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\,(conv) \quad A \equiv B$$

Example. Consider two sorts $*$ and $\Box$ and an axiom $* : \Box$. The judgement $nat : *, z : nat \vdash z : nat$ is derivable in $\mathscr{T}$. But the judgements $z : nat, nat : * \vdash z : nat$ is not because z is declared before *nat* and the judgement $nat : *, x : (* *), z : nat \vdash z : nat$ is not because $(* *)$ is not well-typed.

Definition 2.2 (Well-formed) *Well-formed contexts are inductively defined with the rules*

- *the empty context is well-formed,*
- *if* Γ *is well-formed and* $\Gamma \vdash A : s$ *is derivable in* $\mathscr{T}$*, then* $\Gamma, x : A$ *is well-formed.*

Lemma 2.1 *If* $\Gamma \vdash t : A$ *is derivable, then* Γ *is well-formed. Conversely, if* Γ *is well-formed, then there exist two terms t and A, such that* $\Gamma \vdash t : A$ *is derivable.*

Proof. We prove that Γ is well-formed, by induction on the derivation of $\Gamma \vdash t : A$. Conversely, if Γ is well-formed and s_1 and s_2 are two sorts, such that $\langle s_1, s_2 \rangle \in \mathscr{A}$ then $\Gamma \vdash s_1 : s_2$ is derivable with the rules (sort) and (weak).

We will use the two following lemmas. The first is Lemma 18 in [7] and 5.2.12 in [1] and the second Lemma 26 in [7] and 5.2.17 in [1].

Lemma 2.2 (Thinning) *If* $\Gamma \vdash t : A$ *is derivable,* $\Gamma \subseteq \Gamma'$*, and* Γ' *is well-formed, then* $\Gamma' \vdash t : A$ *is derivable.*

Lemma 2.3 (Strengthening) *If* $\Gamma, x : A, \Gamma' \vdash t : B$ *is derivable and x does not occur in* Γ'*, t, and A, then* $\Gamma, \Gamma' \vdash t : B$ *is derivable.*

Lemma 2.4 (Strengthening contexts) *If* $\Gamma_1, x : A, \Gamma_2$ *is well-formed and x does not occur in* Γ_2 *then* Γ_1, Γ_2 *is well-formed.*

Proof. By induction on the structure of Γ_2. If Γ_2 is empty, then $\Gamma_1, \Gamma_2 = \Gamma_1$ is well-formed. Otherwise, $\Gamma_2 = \Gamma_2', y : B$. By induction hypothesis, Γ_1, Γ_2' is well-formed. As $\Gamma_1, x : A, \Gamma_2', y : B$ is well-formed, $\Gamma_1, x : A, \Gamma_2' \vdash B : s$ is derivable. By Lemma 2.3, $\Gamma_1, \Gamma_2' \vdash B : s$ is derivable. Thus, $\Gamma_1, \Gamma_2', y : B$ is well-formed.

If Γ_1 and Γ_2 are two well-formed contexts with no variables in common, then the concatenation Γ_1, Γ_2 also is well-formed. This remark extend to the case where Γ_1 and Γ_2 have variables in common, but are compatible.

Lemma 2.5 (Merging) *If* Γ_1 *and* Γ_2 *are two well-formed compatible contexts, then there exists a well-formed context* Γ*, such that* $\Gamma_1 \subseteq \Gamma$*,* $\Gamma_2 \subseteq \Gamma$*, and* $\Gamma \subseteq (\Gamma_1, \Gamma_2)$.

Proof. By induction on of Γ_2.

- If Γ_2 is empty, we take $\Gamma = \Gamma_1$. The context Γ is well-formed, $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq \Gamma$, and $\Gamma \subseteq (\Gamma_1, \Gamma_2)$.

- If $\Gamma_2 = (\Gamma_2', x : A)$, then Γ_2' is well-formed and, by induction hypothesis, there exists a well-formed context Γ', such that $\Gamma_1 \subseteq \Gamma'$, $\Gamma_2' \subseteq \Gamma'$, and $\Gamma' \subseteq (\Gamma_1, \Gamma_2')$.
  - If $x : A \in \Gamma'$, then we take $\Gamma = \Gamma'$. The context Γ is well-formed, $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq \Gamma$, and $\Gamma \subseteq (\Gamma_1, \Gamma_2)$.
  - Otherwise, as Γ_1 and Γ_2 are compatible, Γ' contains no other declaration of x. We take $\Gamma = \Gamma', x : A$. We have $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq \Gamma$, $\Gamma \subseteq (\Gamma_1, \Gamma_2)$. By Lemma 2.2, as $\Gamma_2' \vdash A : s$, and $\Gamma_2' \subseteq \Gamma'$, and Γ' is well-formed, $\Gamma' \vdash A : s$ is derivable , thus Γ is well-formed.

Example. Consider two sorts $*$ and $\Box$ and an axiom $* : \Box$. If $\Gamma_1 = \mathit{nat} : *, \mathit{bool} : *, z : \mathit{nat}$ and $\Gamma_2 = \mathit{bool} : *, \mathit{true} : \mathit{bool}, \mathit{nat} : *$, the context Γ is $\mathit{nat} : *, \mathit{bool} : *, z : \mathit{nat}, \mathit{true} : \mathit{bool}$.

3 Arbitrary Contexts

Definition 3.1 (The type system $\mathscr{T}'$) *The system $\mathscr{T}'$ is formed with the rules of Figure 1. With respect to the system $\mathscr{T}$, the rule (sort) is replaced with the rule (sort'), the rule (start) is replaced with the rule (var'), and the rule (weak) is dropped.*

Example. Consider two sorts $*$ and $\Box$ and an axiom $* : \Box$. The judgement $\mathit{nat} : *, z : \mathit{nat} \vdash z : \mathit{nat}$ is derivable in $\mathscr{T}'$. So are the judgements $z : \mathit{nat}, \mathit{nat} : * \vdash z : \mathit{nat}$ and $\mathit{nat} : *, x : (* *), z : \mathit{nat} \vdash z : \mathit{nat}$.

Lemma 3.1 (Thinning) *If Γ and Γ' are two contexts, such that $\Gamma \subseteq \Gamma'$ and $\Gamma \vdash t : A$ is derivable in $\mathscr{T}'$, then $\Gamma' \vdash t : A$ is derivable in $\mathscr{T}'$.*

Proof. By induction on the derivation of $\Gamma \vdash t : A$ in $\mathscr{T}'$.

Lemma 3.2 (Key lemma) *If Γ is well-formed and $\Gamma \vdash t : A$ is derivable in $\mathscr{T}'$, then $\Gamma \vdash t : A$ is derivable in $\mathscr{T}$.*

Proof. By induction on the derivation of $\Gamma \vdash t : A$ in $\mathscr{T}'$.

- If the derivation ends with the rule (sort'), then $t = s_1$, $A = s_2$, and $\langle s_1, s_2 \rangle \in \mathscr{A}$. As Γ is well-formed, $\Gamma \vdash s_1 : s_2$ is derivable in $\mathscr{T}$ with the rules (sort) and (weak).
- If the derivation ends with the rule (var'), then t is a variable x, $\Gamma = \Gamma_1, x : A, \Gamma_2$, and $\Gamma \vdash A : s$ is derivable in the system $\mathscr{T}'$. As Γ is well-formed, $\Gamma_1 \vdash A : s'$ is derivable in $\mathscr{T}$. Thus, $\Gamma_1, x : A \vdash x : A$ is derivable in $\mathscr{T}$ with the rule (start). And, as Γ is well-formed, $\Gamma_1, x : A, \Gamma_2 \vdash x : A$ is derivable with the rule (weak).
- If the derivation ends with the rule (prod), then $t = (x : C) \to D$, $A = s_3$, $\Gamma \vdash C : s_1$ is derivable in $\mathscr{T}'$, $\Gamma, x : C \vdash D : s_2$ is derivable in $\mathscr{T}'$, and $\langle s_1, s_2, s_3 \rangle \in \mathscr{R}$. Then, as Γ is well-formed, by induction hypothesis, $\Gamma \vdash C : s_1$ is derivable in $\mathscr{T}$. Thus, $\Gamma, x : C$ is well-formed and, by induction hypothesis again, $\Gamma, x : C \vdash D : s_2$ is derivable in $\mathscr{T}$. So, $\Gamma \vdash (x : C) \to D : s_3$ is derivable in $\mathscr{T}$ with the rule (prod).
- If the derivation ends with the rule (abs), then $t = \lambda x : C\ u$, $A = (x : C) \to D$, $\Gamma \vdash C : s_1$ is derivable in $\mathscr{T}'$, $\Gamma, x : C \vdash D : s_2$ is derivable in $\mathscr{T}'$, $\Gamma, x : C \vdash u : D$ is derivable in $\mathscr{T}'$, and $\langle s_1, s_2, s_3 \rangle \in \mathscr{R}$. By induction hypothesis, $\Gamma \vdash C : s_1$ is derivable in $\mathscr{T}$. Thus, $\Gamma, x : C$ is well-formed and, by induction hypothesis again, $\Gamma, x : C \vdash D : s_2$ is derivable in $\mathscr{T}$ and $\Gamma, x : C \vdash u : D$ is derivable in $\mathscr{T}$. So, $\Gamma \vdash \lambda x : C\ u : (x : C) \to D$ is derivable in $\mathscr{T}$ with the rule (abs).

- If the derivation ends with the rule (app), then $t = u\ v$, $A = (v/x)D$, $\Gamma \vdash u : (x : C) \to D$ is derivable in $\mathscr{T}'$ and $\Gamma \vdash v : C$ is derivable in $\mathscr{T}'$. By induction hypothesis $\Gamma \vdash u : (x : C) \to D$ is derivable in $\mathscr{T}$ and $\Gamma \vdash v : C$ is derivable in $\mathscr{T}$. Hence $\Gamma \vdash u\ v : (v/x)D$ is derivable in $\mathscr{T}$, with the rule (app).
- If the derivation ends with the rule (conv), then $\Gamma \vdash t : C$ is derivable in $\mathscr{T}'$, $\Gamma \vdash A : s$ is derivable in $\mathscr{T}'$, and $C \equiv A$. By induction hypothesis, $\Gamma \vdash t : C$ is derivable in $\mathscr{T}$ and $\Gamma \vdash A : s$ is derivable in $\mathscr{T}$. Thus, $\Gamma \vdash t : A$ is derivable in $\mathscr{T}$, with the rule (conv).

Lemma 3.3 (Reordering) *Let Γ be a context, x a variable that does not occur in Γ, and Γ' a well-formed context, such that $\Gamma' \subseteq (\Gamma, x : C)$ and $\Gamma' \vdash t : A$ is derivable in $\mathscr{T}'$. Then, there exists a well-formed context Γ'', such that $\Gamma'', x : C \vdash t : A$ is derivable in $\mathscr{T}'$.*

Proof. If $x : C$ is in Γ', then we have $\Gamma' = \Gamma'_1, x : C, \Gamma'_2$, and as $\Gamma'_2 \subseteq \Gamma$, x does not occur in Γ'_2. We take $\Gamma'' = \Gamma'_1, \Gamma'_2$. By Lemma 2.4, Γ'' is well-formed and, by Lemma 3.1, $\Gamma'', x : C \vdash t : A$ is derivable in $\mathscr{T}'$.

Otherwise, we take $\Gamma'' = \Gamma'$. This context is well-formed and, by Lemma 3.1, $\Gamma'', x : C \vdash t : A$ is derivable in $\mathscr{T}'$.

Lemma 3.4 (Context curation) *If $\Gamma \vdash t : A$ is derivable in $\mathscr{T}'$, then there exists a well-formed context Δ, such that $\Delta \subseteq \Gamma$ and $\Delta \vdash t : A$ is derivable in $\mathscr{T}'$.*

Proof. By induction on the derivation of $\Gamma \vdash t : A$.

- If the derivation ends with the rule (sort'), then $t = s_1$ and $A = s_2$, such that $\langle s_1, s_2 \rangle \in \mathscr{A}$. We take the empty context for Δ, $\Delta \subseteq \Gamma$, Δ is well-formed, and $\Delta \vdash s_1 : s_2$ is derivable in $\mathscr{T}'$, with the rule (sort').
- If the derivation ends with the rule (var'), then t is a variable x, $x : A$ is an element of Γ and $\Gamma \vdash A : s$ is derivable in $\mathscr{T}'$. By induction hypothesis, there exists a well-formed context Δ_1, such that $\Delta_1 \subseteq \Gamma$ and $\Delta_1 \vdash A : s$ is derivable in $\mathscr{T}'$.

  If $x : A$ is an element of Δ_1, we take $\Delta = \Delta_1$. We have $\Delta \subseteq \Gamma$ and Δ is well-formed. Moreover $\Delta \vdash A : s$ is derivable in $\mathscr{T}'$ and Δ contains $x : A$, thus $\Delta \vdash x : A$ is derivable in $\mathscr{T}'$, with the rule (var').

  Otherwise, as $\Delta_1 \subseteq \Gamma$, Δ_1 contains no declaration of x, we take $\Delta = \Delta_1, x : A$. We have $\Delta \subseteq \Gamma$. By Lemma 3.2, $\Delta_1 \vdash A : s$ is derivable in $\mathscr{T}$, thus Δ is well-formed. Moreover, by Lemma 3.1, the judgement $\Delta \vdash A : s$ is derivable in $\mathscr{T}'$ and, as Δ contains $x : A$, $\Delta \vdash x : A$ is derivable in $\mathscr{T}'$, with the rule (var').
- If the derivation ends with the rule (prod) then $t = (x : C) \to D$, $A = s_3$, the contexts $\Gamma \vdash C : s_1$ and $\Gamma, x : C \vdash D : s_2$ are derivable in $\mathscr{T}'$, and $\langle s_1, s_2, s_3 \rangle \in \mathscr{R}$. Modulo α-equivalence, we can assume that x does not occur in Γ. By induction hypothesis, there exist two well-formed contexts Γ_1 and Γ_2, such that $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq (\Gamma, x : C)$, and the judgements $\Gamma_1 \vdash C : s_1$ and $\Gamma_2 \vdash D : s_2$ are derivable in $\mathscr{T}'$.

  By Lemma 3.3, there exists a well-formed context Γ'_2 such that $\Gamma'_2, x : C \vdash D : s_2$ is derivable in $\mathscr{T}'$. As Γ_1 and Γ'_2 contain no declaration of x, by Lemma 2.5, there exists a well-formed context Δ, such that $\Gamma_1 \subseteq \Delta$, $\Gamma'_2 \subseteq \Delta$, and Δ contains no declaration of x. We have $\Gamma_1 \subseteq \Delta$ and $\Gamma'_2, x : C \subseteq \Delta, x : C$. Thus, by Lemma 3.1, $\Delta \vdash C : s_1$ and $\Delta, x : C \vdash D : s_2$ are derivable in $\mathscr{T}'$. Thus, $\Delta \vdash (x : C) \to D : s_3$ is derivable in $\mathscr{T}'$, with the rule (prod).

- If the derivation ends with the rule (abs), then $t = \lambda x : C\ u$, $A = (x : C) \to D$, the judgements $\Gamma \vdash C : s_1$, $\Gamma, x : C \vdash D : s_2$, and $\Gamma, x : C \vdash u : D$ are derivable in $\mathscr{T}'$, and $\langle s_1, s_2, s_3 \rangle \in \mathscr{R}$. Modulo α-equivalence, we can assume that x does not occur in Γ. By induction hypothesis, there exist three well-formed contexts Γ_1, Γ_2, and Γ_3, such that $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq (\Gamma, x : C)$, $\Gamma_3 \subseteq (\Gamma, x : C)$, and the judgements $\Gamma_1 \vdash C : s_1$, $\Gamma_2 \vdash D : s_2$, and $\Gamma_3 \vdash u : D$ are derivable in $\mathscr{T}'$.

  By Lemma 3.3, there exists well-formed contexts Γ'_2 and Γ'_3, such that the judgements $\Gamma'_2, x : C \vdash D : s_2$ and $\Gamma'_3, x : C \vdash u : D$ are derivable in $\mathscr{T}'$. As Γ_1, Γ'_2, and Γ'_3 contain no declaration of x, using Lemma 2.5 twice, there exists a well-formed context Δ, such that $\Gamma_1 \subseteq \Delta$, $\Gamma'_2 \subseteq \Delta$, $\Gamma'_3 \subseteq \Delta$, and Δ contains no declaration of x. We have $\Gamma_1 \subseteq \Delta$, $\Gamma'_2, x : C \subseteq \Delta, x : C$, and $\Gamma'_3, x : C \subseteq \Delta, x : C$. Thus, by Lemma 3.1, the judgements $\Delta \vdash C : s_1$, $\Delta, x : C \vdash D : s_2$, and $\Delta, x : C \vdash u : D$ are derivable in $\mathscr{T}'$. Thus, $\Delta \vdash \lambda x : C\ u : (x : C) \to D$ is derivable in $\mathscr{T}'$, with the rule (abs).

- If the derivation ends with the rule (app) then $t = u\ v$, $A = (v/x)D$, and the judgements $\Gamma \vdash u : (x : C) \to D$ and $\Gamma \vdash v : C$ are derivable in $\mathscr{T}'$. By induction hypothesis, there exists two well-formed contexts Γ_1 and Γ_2, such that $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq \Gamma$, and the judgements $\Gamma_1 \vdash u : (x : C) \to D$ and $\Gamma_2 \vdash v : C$ are derivable in $\mathscr{T}'$.

  By Lemma 2.5, there exists a well-formed context Δ, such that $\Gamma_1 \subseteq \Delta$, and $\Gamma_2 \subseteq \Delta$. By Lemma 3.1, the judgements $\Delta \vdash u : (x : C) \to D$ and $\Delta \vdash v : C$ are derivable in $\mathscr{T}'$. Thus, $\Delta \vdash (u\ v) : (v/x)D$ is derivable in $\mathscr{T}'$, with the rule (app).

- If the derivation ends with the rule (conv) then the judgements $\Gamma \vdash t : C$ and $\Gamma \vdash A : s$ are derivable in $\mathscr{T}'$, and $C \equiv A$. By induction hypothesis, there exists two well-formed contexts Γ_1 and Γ_2, such that $\Gamma_1 \subseteq \Gamma$, $\Gamma_2 \subseteq \Gamma$, and the judgements $\Gamma_1 \vdash t : C$ and $\Gamma_2 \vdash A : s$ are derivable in $\mathscr{T}'$.

  By Lemma 2.5, there exists a well-formed context Δ, such that $\Gamma_1 \subseteq \Delta$ and $\Gamma_2 \subseteq \Delta$. By Lemma 3.1, the judgements $\Delta \vdash t : C$ and $\Delta \vdash A : s$ are derivable in $\mathscr{T}'$. Thus, $\Delta \vdash t : A$ is derivable in $\mathscr{T}'$, with the rule (conv).

Theorem 3.1 *If $\Gamma \vdash t : A$ is derivable in $\mathscr{T}'$, then there exists Δ, such that $\Delta \subseteq \Gamma$ and $\Delta \vdash t : A$ is derivable in $\mathscr{T}$.*

Proof. By Lemma 3.4, there exists a well-formed context Δ, such that $\Delta \subseteq \Gamma$ and $\Delta \vdash t : A$ is derivable in $\mathscr{T}'$. By Lemma 3.2, $\Delta \vdash t : A$ is derivable in $\mathscr{T}$.

Example. Consider two sorts $*$ and $\Box$ and an axiom $* : \Box$. From the derivation of the judgement, $z : nat, nat : * \vdash z : nat$, we extract the context $nat : *, z : nat$.

And from the derivation of $nat : *, x : (* *), z : nat \vdash z : nat$, we also extract the context $nat : *, z : nat$.

Acknowledgements

The author wants to thank Frédéric Blanqui, Herman Geuvers, and Claudio Sacerdoti Coen for useful and lively discussions about the various presentations of type theory.

References

[1] H. Barendregt (1992): *Lambda calculi with types*. In S. Abramsky, D.M. Gabbay & T.S.E. Maibaum, editors: *Handbook of Logic in Computer Science*, 2, Oxford University Press, pp. 117–309.

[2] F. Blanqui (2001): *Definitions by Rewriting in the Calculus of Constructions*. In: *Logic in Computer Science*, IEEE Computer Society, pp. 9–18, doi:10.1109/LICS.2001.932478.

[3] C. Sacerdoti Coen (2004): *Mathematical Libraries as Proof Assistant Environments*. In A. Asperti, G. Bancerek & A. Trybulec, editors: *Mathematical Knowledge Management*, *Lecture Notes in Computer Science* 3119, Springer, pp. 332–346, doi:10.1007/978-3-540-27818-4_24.

[4] T. Coquand & G. Huet (1988): *The Calculus of Constructions*. *Information and Computation* 76(2/3), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[5] D. Cousineau & G. Dowek (2007): *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo*. In S. Ronchi Della Rocca, editor: *Typed Lambda Calculi and Applications*, *Lecture Notes in Computer Science* 4583, Springer, pp. 102–117, doi:10.1007/978-3-540-73228-0_9.

[6] H. Geuvers, R. Krebbers, J. McKinna & F. Wiedijk (2010): *Pure Type Systems without Explicit Contexts*. In K. Crary & M. Miculan, editors: *Logical Frameworks and Meta-languages: Theory and Practice*, *Electronic Proceedings in Theoretical Computer Science* 34, Open Publishing Association, pp. 53–67, doi:10.4204/EPTCS.34.6.

[7] H. Geuvers & M.-J. Nederhof (1991): *Modular proof of strong normalization for the calculus of constructions*. *Journal of Functional Programming* 1(2), pp. 155–189, doi:10.1017/S0956796800020037.

Automating Induction by Reflection

Johannes Schoisswohl
University of Manchester, UK TU Wien, Austria
johannes.schoisswohl@manchester.ac.uk

Laura Kovács
TU Wien, Austria
laura.kovacs@tuwien.ac.at

Despite recent advances in automating theorem proving in full first-order theories, inductive reasoning still poses a serious challenge to state-of-the-art theorem provers. The reason for that is that in first-order logic induction requires an infinite number of axioms, which is not a feasible input to a computer-aided theorem prover requiring a finite input. Mathematical practice is to specify these infinite sets of axioms as axiom schemes. Unfortunately these schematic definitions cannot be formalized in first-order logic, and therefore not supported as inputs for first-order theorem provers.

In this work we introduce a new method, inspired by the field of axiomatic theories of truth, that allows to express schematic inductive definitions, in the standard syntax of multi-sorted first-order logic. Further we test the practical feasibility of the method with state-of-the-art theorem provers, comparing it to solvers' native techniques for handling induction.

1 Introduction

Automated reasoning has advanced tremendously in the last decades, pushing the limits of what computer programs can prove about other computer programs. Recent progress in this area features techniques such as first-order reasoning with term algebras [18], embedding programming control structures in first-order logic [17], the AVATAR architecture [27], combining theory instantiation and unification with abstraction in saturation-based proof search [23], automating proof search for higher-order logic [3, 4], and first-order logic with rank-1 polymorphism [5].

Despite the fact that first-order logic with equality can be handled very well in many practical cases, there is one fundamental mathematical concept most first-order theorem provers lack; namely inductive reasoning. Not only is induction a very interesting theoretical concept, it is also of high practical importance, since it is required to reason about software reliability, safety and security, see e.g. [13, 7, 10, 22, 11]. Such and similar software requirement typically involve properties over natural numbers, recursion, unbounded loop iterations, or recursive data structures like lists, and trees.

Many different approaches towards automating inductive reasoning have been proposed, ranging from cyclic proofs [9, 16], computable approximations of the ω-rule [2], recursion analysis [1, 8, 21], theory exploration [6, 25], inductive strengthening [20, 25], and integrating induction in saturation based theorem proving [12, 8, 24]. What all these approaches have in common is that they tackle the problem of inductive theorem proving by specializing the proof system. In the present paper, we propose a different approach. Instead of changing the proof system, we will change our input problem, replacing the infinite induction scheme by a finite conservative extension. Hence our approach is not tailored to one specific reasoner, but can be used with any automated theorem prover for first-order logic.

In order to achieve such a generic approach, we use ideas from axiomatic theories of truth [15]. As Gödel's incompleteness theorem tells us that there is a formula $Bew(x)$ in Peano Arithmetic **PA** that expresses provability in **PA**, Tarski's undefinability theorem teaches us that there is no formula $T(x)$ that expresses truth in **PA**. Extending the language of **PA**, in order to be able to express truth in **PA** is the core idea of axiomatic theories of truth. The truth theory we introduce in this paper will not be an extension of

E. Pimentel and E. Tassi (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 39–54, doi:10.4204/EPTCS.337.4

PA, but an extension of an arbitrary theory. This theory will let us express the sentence "for all formulas, the induction scheme is true", within first order logic.

As our experimental evaluation shows our technique does not outperform state of the art built-in methods for inductive reasoning in general, we will see that there are cases where an improvement can be achieved.

The contributions of our work can be summarized as follows:

- We introduce a method for conservatively extending an arbitrary theory with a truth predicate (Section 3);
- We show how our method can be used to replace the induction scheme of **PA**, and other theories involving inductive datatypes (Section 4);
- We provide a new set of benchmarks to the automated theorem proving community (Section 5);
- We conduct experiments on this set of benchmarks with state-of-the-art theorem provers. (Section 5)

2 Preliminaries

We assume familiarity with multi-sorted first-order logic and automated theorem proving; for details, we refer to [19, 5].

We consider reasoning on the object and the meta level. Therefore we will use different symbols for logic on all of these levels. We use the symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\approx$ for negation, conjunction, disjunction, implication, equivalence, and equality respectively, and write $Qx:\sigma.\phi$ with $Q \in \{\exists, \forall\}$ for existential and universal quantification over the sort σ on the object level. We will drop sort declarations for quantified variables when there is no ambiguity. Further we will write $\bigvee\!\!\!\!\bigvee_{x_1,\ldots,x_n} \phi$ for the formula $\forall x_1 \ldots \forall x_n.\phi$, and $\bigvee\!\!\!\!\bigvee \phi$ to denote the universal closure of ϕ.

On the meta-level we will use !, &, $\|$, $\Longrightarrow$, $\Longleftrightarrow$, and $=$ for negation, conjunction, disjunction, implication, equivalence, and equality and $\forall$, and $\exists$ for quantification. Meta-level logical formulas will only be used where they help to improve readability and precision, and otherwise natural language will be used.

By $\mathbf{Var}_\sigma$, $\mathbf{Term}_\sigma$, and $\mathbf{Form}$, we respectively denote the sets of variables of sort σ, terms of sort σ and object-level formulas over a signature Σ. As $\mathbf{Var}_\sigma$ is a countably infinite set, we assume without loss of generality that it is composed of the variables $\{\mathsf{x}_i^\sigma \mid i \in \mathbb{N}\}$, and leave away the sort superscript σ, if it is clear from the context.

For a function symbol f from signature Σ, we write $f :: \sigma_1 \times \ldots \times \sigma_n \rightsquigarrow \sigma \in \Sigma$, to denote that $\mathbf{dom}(f) = \sigma_1 \times \ldots \times \sigma_n$ is the domain of f, $\mathbf{codom}(f) = \sigma$ is its codomain, and $\mathbf{arity}(f) = n$ is the arity of f. We consider constants being functions of arity 0 and write $c :: \sigma$ for $c ::\rightsquigarrow \sigma$. Further we write $P : \mathsf{Pred}(\sigma_n \times \ldots \times \sigma_n)$ to denote that P is a predicate with domain $\mathbf{dom}(P) = \sigma_1 \times \ldots \times \sigma_n$ and arity $\mathbf{arity}(P) = n$. Further, we write $\mathbf{dom}(s,i)$ to refer to the ith component of the domain of s.

Given formula/term ϕ, a variable x and a term t, we write $\phi[x \mapsto t]$ to denote the formula/term resulting from replacing all occurences of x by t in ϕ. Similarly, if x is a variable and $\phi[x]$ is a formula/term, we denote the formula/term resulting from replacing all occurences of x for t by $\phi[t]$.

A formula is open if it contains free variables, and closed otherwise. We consider a theory to be a set of closed formulas. If $\mathscr{T}$ is a theory with signature Σ, by $\mathbf{Form}^{\mathscr{T}}$ we denote the set of all formulas over Σ.

The semantics of formulas and terms over a signature Σ is defined using multi-sorted first-order interpretations $\mathscr{M}$, consisting of $\langle\langle\Delta_{\sigma_1},\ldots,\Delta_{\sigma_n}\rangle,\mathscr{I}\rangle$, where: Δ_{σ_i} is the domain for sort $\sigma\in\mathbf{sorts}_\Sigma$, and $\mathscr{I}$ is an interpretation function that freely interprets variables, function symbols, and predicate symbols, respecting the sorts, and is extended to terms in the standard way. By Δ we denote $\langle\Delta_{\sigma_1},\ldots,\Delta_{\sigma_n}\rangle$. We write $\mathscr{M}\vDash\phi$ for the structure $\mathscr{M}$ satisfying the formula ϕ and say that $\mathscr{M}$ is a model of ϕ. We write $\mathscr{I}\vDash\phi$ instead of $\langle\langle\Delta_{\sigma_1},\ldots,\Delta_{\sigma_n}\rangle,\mathscr{I}\rangle\vDash\phi$, whenever the domains are clear from context. By $\mathbf{Interpret}_\Sigma$ we denote the class of all interpretation functions over a signature Σ.

For defining our approach for reflective reasoning, we need the concept of a conservative extension. A conservative extension of a theory $\mathscr{T}$ is a theory $\mathscr{T}'$, such that $\Sigma_\mathscr{T}\subseteq\Sigma_{\mathscr{T}'}$, and $\forall\phi\in\mathbf{Form}^\mathscr{T}.(\mathscr{T}\vDash\phi\iff\mathscr{T}'\vDash\phi)$

An inductive datatype $\mathscr{D}_\tau$, with respect to some signature Σ is a pair $\langle\tau,\mathbf{ctors}_\tau\rangle$, where τ is a sort and $\mathbf{ctors}_\tau$ is a set of function symbols $F\subset\Sigma$ such that $\forall f\in F.\mathbf{codom}(f)=\tau$. By the first-order structural induction scheme of $\mathscr{D}_\tau$ we denote the set of formulas:

$$\left\{\left(\bigwedge_{c\in\mathbf{ctors}_\tau} case_c\right)\to\forall x.\phi[x]\mid\phi[x]\in\mathbf{Form}\right\}\tag{$\mathbf{I}_\tau$}$$

where

$$case_c=\mathop{\Large\forall\!\!\!\forall}_{x_1,\ldots,x_n}\left(\left(\bigwedge_{i\in recursive_c}\phi[x_i]\right)\to\phi[c(x_1,\ldots,x_n)]\right)$$
$$recursive_c=\{i\mid\mathbf{dom}_\Sigma(c,i)=\tau\}$$

x the induction variable, $\bigwedge_{c\in\mathbf{ctors}_\tau}case_c$ the induction premise, and $\forall x\phi[x]$ the induction conclusion.

An example for such an inductive datatype is the type of lists $\mathscr{D}_{\mathsf{List}}=\langle\mathsf{List},\{\mathsf{nil}::\mathsf{List},\mathsf{cons}::\alpha\times\mathsf{List}\rightsquigarrow\mathsf{List}\}\rangle$. The first-order induction scheme for $\mathscr{D}_{\mathsf{List}}$ is therefore

$$\left\{case_{\mathsf{nil}}\wedge case_{\mathsf{cons}}\to\forall x.\phi[x]\mid\phi[x]\in\mathbf{Form}\right\}\qquad case_{\mathsf{nil}}=\top\to\phi[\mathsf{nil}]$$
$$case_{\mathsf{cons}}=\forall x:\alpha,xs:\mathsf{List}.\Big(\phi[xs]\to\phi[\mathsf{cons}(x,xs)]\Big)$$

3 Reflective extension

Our aim is to finitely axiomatise the induction scheme $\mathbf{I}_\tau$ with respect to some datatypes $\mathscr{D}$ and an arbitrary base theory $\mathscr{T}$. We will hence first construct a conservative extension $\dot{\mathscr{T}}$ of $\mathscr{T}$, which allow us to quantify over first-order formulas of the language of $\mathscr{T}$. In order to achieve this we will take an approach that is inspired by Horsten's theory **TC** [15]. There are however a few crucial differences between our approach and [15], as follows. While **TC** [15] is an extension of Peano Arithmetic **PA**, our work can be used for an arbitrary theory $\mathscr{T}$. Further, while **TC** [15] relies on numbers to encode formulas, our approach uses multi-sorted logic and introduces additional sorts for formulas, terms, and variables, yielding a rather straightforward definition of models and proof of consistency for the extended theory.

The basic idea of our approach is to redefine the syntax and semantics of first-order logic in first-order logic itself. As such, there will be three levels of reasoning. As usual, we have the meta level

logic (the informal logical reasoning going on in the paper), and the object level (the formal logical reasoning we reason about on the meta level). In addition we will have a logic embedded in the terms of object level logic formulas. We will refer to this level as the *reflective level*, and refer to formulas/functions/terms/expressions as reflective formulas/functions/terms/expressions.

3.1 Signature

In a first step, we extend the signature Σ and the set of sorts $\mathbf{sorts}_\Sigma$ of our base theory $\mathscr{T}$ with the vocabulary to be able to talk about variables, terms, and formulas.

Definition 1 (Reflective signature)**.** *Let Σ be an arbitrary signature. We define $\dot{\Sigma}$ to be the reflective extension of Σ.*

$$
\begin{aligned}
\mathbf{sorts}_{\dot{\Sigma}} = {} & \{\mathsf{var}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{term}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{form}, \mathsf{env}\}
\end{aligned}
$$

$$
\begin{aligned}
\dot{\Sigma} = {} & \Sigma \cup \{\mathsf{v}_0^\sigma :: \mathsf{var}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{next}_\sigma :: \mathsf{var}_\sigma \rightsquigarrow \mathsf{var}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{inj}_\sigma :: \mathsf{var}_\sigma \rightsquigarrow \mathsf{term}_\sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\dot{f} :: \mathsf{term}_{\sigma_1} \times \ldots \times \mathsf{term}_{\sigma_n} \rightsquigarrow \mathsf{term}_\sigma \mid f :: \sigma_1 \times \ldots \times \sigma_n \rightsquigarrow \sigma \in \Sigma\} \\
& \cup \{\dot{P} :: \mathsf{term}_{\sigma_1} \times \ldots \times \mathsf{term}_{\sigma_n} \rightsquigarrow \mathsf{form} \mid P :: \mathsf{Pred}(\sigma_1 \times \ldots \times \sigma_n) \in \Sigma\} \\
& \cup \{\dot{\approx}_\sigma :: \mathsf{term}_\sigma \times \mathsf{term}_\sigma \rightsquigarrow \mathsf{form} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\dot{\bot} :: \mathsf{form}, \dot{\vee} :: \mathsf{form} \times \mathsf{form} \rightsquigarrow \mathsf{form}, \dot{\neg} :: \mathsf{form} \rightsquigarrow \mathsf{form}\} \\
& \cup \{\dot{\forall}_\sigma :: \mathsf{var}_\sigma \times \mathsf{form} \rightsquigarrow \mathsf{form} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{empty} :: \mathsf{env} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{push}_\sigma :: \mathsf{env} \times \mathsf{var}_\sigma \times \sigma \rightsquigarrow \mathsf{env} \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{eval}_\sigma^v :: \mathsf{env} \times \mathsf{var}_\sigma \rightsquigarrow \sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\mathsf{eval}_\sigma :: \mathsf{env} \times \mathsf{term}_\sigma \rightsquigarrow \sigma \mid \sigma \in \mathbf{sorts}_\Sigma\} \\
& \cup \{\dot{\models} :: \mathsf{Pred}(\mathsf{env} \times \mathsf{form})\}
\end{aligned}
$$

where all newly introduced symbols, and sorts are disjoint from the ones in Σ, and $\mathbf{sorts}_\Sigma$ respectively.

As this definition is rather lengthy we will break down the intended semantics of all newly introduced symbols. We can split the definitions into two parts: (i) one formalizing the syntax and (ii) one formalizing the semantics of our reflective first-order logic.

(i) Reflective syntax. Our reflective syntax is formalized as follows.

Variables The sort var_σ is used to represent the countably infinite set of variables $\mathbf{Var}_\sigma$. The two functions v_0^σ, and next_σ that are added to the signature can be thought of as the constructors for this infinite set of variables. This means v_0^σ is intended to be interpreted as the variable x_0,

$\mathsf{next}_\sigma(\mathsf{v}_0^\sigma)$ is meant to be interpreted as x_1, and so on. We introduce the following syntactic sugar for variables:

$$\mathsf{v}_{i+1}^\sigma = \mathsf{next}_\sigma(\mathsf{v}_i^\sigma) \qquad \text{for } i \geq 0$$

Terms We use the sort term_σ to represent terms of sort $\mathbf{Term}_\sigma$. On the meta level terms are defined inductively, as follows.

The base case is a variable. Since variables and terms are of different sorts, we need the function inj_σ to turn variables into terms. This function is intended to be interpreted as the identity function.

The step case of the inductive definition is building terms out of function symbols and other terms. Therefore, we need to introduce a reflective function symbol $\dot{f}$ for every function f in the signature. The $\dot{f}$ is intended to be interpreted as the function symbol f, while f itself is interpreted as an actual function.

Formulas As for terms, formulas **Form** are defined inductively on the meta level.

For atomic formulas we introduce a reflective equality symbol $\dot{\approx}_\sigma$ for each sort σ and a reflective version $\dot{P}$ for every predicate symbol P. Even though it's not strictly necessary we introduce a nullary reflective connective $\dot{\bot}$ is intended to be interpreted as the formula $\bot$.

Complex formulas are built from atomic formulas and connectives, or quantifiers. Therefore we introduce a functionally complete set of reflective connectives, namely $\dot{\vee}$, and $\dot{\neg}$. As it will help in terms of readability, we will use infix notation for $\dot{\vee}$, and drop the parenthesis for $\dot{\neg}$ if there is no ambiguity.

In order to formalize quantification we introduce a function $\dot{\forall}_\sigma$ for each sort. We will write $\dot{\forall} x{:}\sigma.p$ for the term $\dot{\forall}_\sigma(x,p)$.

(ii) Reflective semantics. For axiomatising the meaning of formulas, we will use syntactic representations of the semantic structures needed to define the semantics of first-order logic.

Environment In order to define the meaning of a quantifier, we redefine the meaning of a variable within the scope of the quantifier. Therefore we will use a stack of variable interpretations, which we will call an environment. The idea is that a variable v_i^σ is freely interpreted in an empty environment empty, while it is interpreted as t if the tuple $\langle \mathsf{v}_i^\sigma, t\rangle$ was pushed on the stack using $\mathsf{push}_\sigma(e,\mathsf{v}_i^\sigma,t)$. This setting becomes more clear in Sections 3.2-3.3, where we axiomatise the meaning and define a model of the reflective theory.

Evaluation To make use of the environment, we need a reflective evaluation function for terms eval_σ and eval_σ^v that corresponds to interpreting terms and variables in some model $\mathscr{I}$ of the reflective theory.

Satisfaction Finally, we have our reflective satisfaction relation $\dot{\models}$. We write $e \dot{\models} p$ for $\dot{\models}(e,p)$, which can roughly be interpreted as "the interpretation $\mathscr{I}$ partially defined by e satisfies p". Our truth $\mathbf{T}$ predicate in the Tarskian sense is $\mathbf{T}(x) = (\mathsf{empty} \dot{\models} x)$.

3.2 Axiomatisation

We now formalize our semantics. We relate reflective with non-reflective function and predicate symbols, by defining the meaning of the reflective satisfaction relation $\dot{\models}$, and the meaning of the reflective evaluation functions eval_σ, and eval_σ^v. All axioms we list are implicitly universally quantified, and one instance of them will be present for every sort $\sigma,\tau \in \mathbf{sorts}_\Sigma$. Finally, the reflective extension $\dot{\mathscr{T}}$ of our base theory $\mathscr{T}$ is the union of all these axioms and $\mathscr{T}$.

Reflective variable interpretation. As already mentioned, the interpretation of variables in an empty environment empty is undefined. In contrast an environment to which a variable v, and a value x is pushed, evaluates the variable v to x. Hence,

$$\mathsf{eval}^v_\sigma(\mathsf{push}_\sigma(e,v,x),v) = x \qquad (\mathsf{Ax}_{\mathsf{eval}^v_0})$$

$$v \not\approx v' \to \mathsf{eval}^v_\sigma(\mathsf{push}_\sigma(e,v',x),v) = \mathsf{eval}^v_\sigma(e,v) \qquad (\mathsf{Ax}_{\mathsf{eval}^v_1})$$

$$\mathsf{eval}^v_\sigma(\mathsf{push}_\tau(e,w,x),v) = \mathsf{eval}^v_\sigma(e,v) \qquad \text{for } \sigma \neq \tau \qquad (\mathsf{Ax}_{\mathsf{eval}^v_2})$$

Reflective evaluation. The function symbol eval_σ defines the value of a reflective term t, and thereby maps the reflective functions $\dot{f}$ to their non-reflective counter parts f. For variables eval_σ, the evaluation to eval^v_σ is used.

$$\mathsf{eval}_\sigma(e,\mathsf{inj}_\sigma(v)) = \mathsf{eval}^v_\sigma(e,v) \qquad (\mathsf{Ax}_{\mathsf{eval}_{var}})$$

$$\mathsf{eval}_\sigma(e,\dot{f}(t_1,...,t_n)) = f(\mathsf{eval}_{\sigma_1}(e,t_1),...,\mathsf{eval}_{\sigma_n}(e,t_n)) \qquad (\mathsf{Ax}_{\mathsf{eval}_f})$$

$$\text{for } f : \sigma_1 \times ... \times \sigma_n \rightsquigarrow \sigma \in \Sigma$$

Reflective satisfaction. The predicate symbol $\dot{\models}\,\sigma$ defines the truth of a formula with respect to some variable interpretation. To this end, the meaning of the reflective connectives and the quantifiers in terms is defined by their respective object-level counterparts, as follows:

$$(e \mathrel{\dot{\models}} x \mathrel{\dot{\approx}}_\sigma y) \leftrightarrow \mathsf{eval}_\sigma(e,x) \approx \mathsf{eval}_\sigma(e,y) \qquad (\mathsf{Ax}_{\dot{\approx}})$$

$$(e \mathrel{\dot{\models}} \dot{P}(t_1,...,t_n)) \leftrightarrow P(\mathsf{eval}_{\sigma_1}(e,t_1),...,\mathsf{eval}_{\sigma_n}(e,t_n)) \qquad \text{for } P : \mathsf{Pred}(\sigma_1 \times ... \times \sigma_n) \qquad (\mathsf{Ax}_P)$$

$$(e \mathrel{\dot{\models}} \dot{\bot}) \leftrightarrow \bot \qquad (\mathsf{Ax}_{\dot{\bot}})$$

$$(e \mathrel{\dot{\models}} \dot{\neg}\,\phi) \leftrightarrow \neg(e \mathrel{\dot{\models}} \phi) \qquad (\mathsf{Ax}_{\dot{\neg}})$$

$$(e \mathrel{\dot{\models}} \phi \mathrel{\dot{\vee}} \psi) \leftrightarrow (e \mathrel{\dot{\models}} \phi) \vee (e \mathrel{\dot{\models}} \psi) \qquad (\mathsf{Ax}_{\dot{\vee}})$$

$$(e \mathrel{\dot{\models}} \dot{\forall} v{:}\sigma.\phi) \leftrightarrow \forall x : \sigma.(\mathsf{push}_\sigma(e,v,x) \mathrel{\dot{\models}} \phi) \qquad (\mathsf{Ax}_{\dot{\forall}})$$

3.3 Consistency and Conservativeness

As we have now specified our theory, we next ensure that (i) $\dot{\mathscr{T}}$ is indeed a conservative extension of $\mathscr{T}$ and (ii) $\dot{\mathscr{T}}$ is consistent. In general, we cannot ensure that $\dot{\mathscr{T}}$ is consistent, since already the base theory $\mathscr{T}$ could have been inconsistent. Hence we will show that $\dot{\mathscr{T}}$ is consistent if $\mathscr{T}$ is consistent.

In order to prove (i) and (ii), that is conservativeness and consistency of $\dot{\mathscr{T}}$, we introduce the notion of a *reflective model* $\dot{\mathscr{M}}$, that is based on a model $\mathscr{M}$ of $\mathscr{T}$. The basic idea is that $\dot{\mathscr{M}}$ interprets every symbol in the base theory $\mathscr{T}$ as it would be interpreted in $\mathscr{M}$, hence every formula in $\mathbf{Form}^{\mathscr{T}}$ is true in $\dot{\mathscr{M}}$ iff it is true in $\mathscr{M}$. Due to soundness and completeness of first-order logic we get that $\dot{\mathscr{T}}$ is indeed a conservative extension of $\mathscr{T}$. Further, due to the fact that for every model of $\mathscr{M}$ of $\mathscr{T}$ we have a model $\dot{\mathscr{M}}$ of $\dot{\mathscr{T}}$, we also have that $\dot{\mathscr{T}}$ is consistent if $\mathscr{T}$ is consistent. In order to ensure this reasoning is correct we need to ensure that $\dot{\mathscr{M}}$ also satisfies the axioms we introduced for reflective theories. This will be done by interpreting the new reflective sort form as the set of first order formulas **Form**, and interpreting the sort term_σ as terms of sort $\mathbf{Term}_\sigma$.

Definition 2 (Reflective interpretation)**.** *Let $\mathscr{M} = \langle\langle \Delta_{\sigma_1},...,\Delta_{\sigma_n}\rangle, \mathscr{I}\rangle$ be a first-order interpretation over the signature Σ. We define the reflective interpretation $\dot{\mathscr{M}}$ to be*

$$\dot{\mathscr{M}} = \langle\langle \Delta_{\sigma_1}, ..., \Delta_{\sigma_n}, \mathbf{Term}_{\sigma_1}, ... \mathbf{Term}_{\sigma_n}, \mathbf{Form}\rangle, \dot{\mathscr{I}}\rangle$$

$$\dot{\mathscr{I}}(f) : \Delta_{\sigma_1} \times ... \times \Delta_{\sigma_n} \mapsto \Delta_\sigma \qquad \text{for } f :: \sigma_1 \times ... \times \sigma_n \rightsquigarrow \sigma \in \Sigma$$
$$\dot{\mathscr{I}}(f) = \mathscr{I}(f)$$

$$\dot{\mathscr{I}}(P) : \mathscr{P}(\Delta_{\sigma_1} \times ... \times \Delta_{\sigma_n}) \qquad \text{for } P :: \mathsf{Pred}(\sigma_1 \times ... \times \sigma_n) \in \Sigma$$
$$\dot{\mathscr{I}}(P) = \mathscr{I}(P)$$

$$\dot{\mathscr{I}}(\mathsf{v}_0^\sigma) : \mathbf{Var}_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{v}_0^\sigma) = \mathsf{x}_0$$

$$\dot{\mathscr{I}}(\mathsf{next}_\sigma) : \mathbf{Var}_\sigma \mapsto \mathbf{Var}_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{next}_\sigma)(\mathsf{x}_i) = \mathsf{x}_{i+1}$$

$$\dot{\mathscr{I}}(\mathsf{inj}_\sigma) : \mathbf{Var}_\sigma \mapsto \mathbf{Term}_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{inj}_\sigma)(x) = x$$

$$\dot{\mathscr{I}}(\dot{f}) : \mathbf{Term}_{\sigma_1} \times ... \times \mathbf{Term}_{\sigma_n} \mapsto \mathbf{Term}_\sigma \qquad \text{for } f :: \sigma_1 \times ... \times \sigma_n \rightsquigarrow \sigma \in \Sigma$$
$$\dot{\mathscr{I}}(\dot{f})(t_1, ..., t_n) = f(\dot{\mathscr{I}}(t_1), ..., \dot{\mathscr{I}}(t_n))$$

$$\dot{\mathscr{I}}(\dot{P}) : \mathbf{Term}_{\sigma_1} \times ... \times \mathbf{Term}_{\sigma_n} \mapsto \mathbf{Form} \qquad \text{for } P :: \mathsf{Pred}(\sigma_1 \times ... \times \sigma_n) \in \Sigma$$
$$\dot{\mathscr{I}}(\dot{P})(t_1, ..., t_n) = P(\dot{\mathscr{I}}(t_1), ..., \dot{\mathscr{I}}(t_n))$$

$$\dot{\mathscr{I}}(\dot{\approx}_\sigma) : \mathbf{Term}_\sigma \times \mathbf{Term}_\sigma \mapsto \mathbf{Form} \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\dot{\approx}_\sigma)(s,t) = \dot{\mathscr{I}}(s) \approx \dot{\mathscr{I}}(t)$$

$$\dot{\mathscr{I}}(\dot{\bot}) : \mathbf{Form}$$
$$\dot{\mathscr{I}}(\dot{\bot}) = \bot$$

$$\dot{\mathscr{I}}(\dot{\vee}) : \mathbf{Form} \times \mathbf{Form} \mapsto \mathbf{Form}$$
$$\dot{\mathscr{I}}(\dot{\vee})(\phi, \psi) = \phi \vee \psi$$

$$\dot{\mathscr{I}}(\dot{\neg}) : \mathbf{Form} \mapsto \mathbf{Form}$$
$$\dot{\mathscr{I}}(\dot{\neg})(\phi) = \neg\phi$$

$$\dot{\mathscr{I}}(\dot{\forall}_\sigma) : \mathbf{Var}_\sigma \times \mathbf{Form} \mapsto \mathbf{Form} \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\dot{\forall}_\sigma)(\mathsf{x}_i, \phi) = \forall \mathsf{x}_i : \sigma.\phi$$

$$\dot{\mathscr{I}}(\mathsf{empty}) : \mathbf{Interpret}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{empty}) = \mathscr{I}$$

$$\dot{\mathscr{I}}(\mathsf{push}_\sigma) : \mathbf{Interpret}_\Sigma \times \mathbf{Var}_\sigma \times \sigma \mapsto \mathbf{Interpret}_\Sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{push}_\sigma)(\mathscr{J}, \mathsf{x}_i, v)(x) = \begin{cases} v & \text{if } x = \mathsf{x}_i \\ \mathscr{J}(x) & \text{otherwise} \end{cases}$$

$$\dot{\mathscr{I}}(\mathsf{eval}^v_\sigma) : \mathbf{Interpret}_\Sigma \times \mathbf{Var}_\sigma \mapsto \Delta_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{eval}^v_\sigma)(\mathscr{J}, \mathsf{x}_i) = \mathscr{J}(\mathsf{x}_i)$$

$$\dot{\mathscr{I}}(\mathsf{eval}_\sigma) : \mathbf{Interpret}_\Sigma \times \mathbf{Term}_\sigma \mapsto \Delta_\sigma \qquad \text{for } \sigma \in \mathbf{sorts}_\Sigma$$
$$\dot{\mathscr{I}}(\mathsf{eval}_\sigma)(\mathscr{J}, t) = \mathscr{J}(t)$$

$$\dot{\mathscr{I}}(\dot{\models}) : \mathscr{P}(\mathbf{Interpret}_\Sigma \times \mathbf{Form})$$
$$\dot{\mathscr{I}}(\dot{\models}) = \{\langle \mathscr{J}, \phi \rangle \in \mathbf{Interpret}_\Sigma \times \mathbf{Form} \mid \mathscr{J} \models \phi\}$$

We now need to ensure that our reflective interpretation $\dot{\mathscr{M}}$ is indeed a model of $\dot{\mathscr{T}}$ if $\mathscr{M}$ is a model of $\mathscr{T}$.

Theorem 1 (Reflective model)**.**

$$\mathscr{M} \models \mathscr{T} \iff \dot{\mathscr{M}} \models \dot{\mathscr{T}}$$

Proof. The "$\Longleftarrow$" part of the biconditional is trivial since $\mathscr{T} \subset \dot{\mathscr{T}}$, and $\dot{\mathscr{M}}$ interprets all symbols of the original signature in the same way as $\mathscr{M}$.

For the same reason as before we have that $\mathscr{M} \models \mathscr{T} \implies \dot{\mathscr{M}} \models \mathscr{T}$. Hence we are left to show that $\dot{\mathscr{M}} \vdash \dot{\mathscr{T}} \setminus \mathscr{T}$. This follows from the axioms we introduced in 3.2 in natural language, as well as from the meta level semantics of first-order logic, by also making sure that our meta level and our object level definitions match. □

3.4 Truth predicate

We showed that our theory $\dot{\mathscr{T}}$ is indeed a conservative extension of $\mathscr{T}$. Next we prove that $\dot{\mathscr{T}}$ behaves in the way we need it for axiomatising induction. That is, we need to make sure that $\dot{\mathscr{T}}$ has a truth predicate, allowing us to quantify over formulas and thus defining the induction scheme as a single formula.

As in [15], we use a Gödel encoding to state that our theory $\dot{\mathscr{T}}$ has a truth predicate. Usually, a Gödel encoding maps variables, terms, and formulas to numerals. Since our theory $\dot{\mathscr{T}}$ does not necessarily contain number symbols, we need to use a more general notion of a Gödel encoding, namely that it maps variables, terms, and formulas in our base language $\mathbf{Form}^{\mathscr{T}}$ to terms in our extended language $\mathbf{Form}^{\dot{\mathscr{T}}}$. That is, we map formulas **Form** to terms of sort form, variables $\mathbf{Var}_\sigma$ to var_σ and $\mathbf{Term}_\sigma$ to term_σ. Formally, we define our Gödel encoding as follows:

Definition 3 (Gödel encoding)**.**

$$\ulcorner \phi \vee \psi \urcorner = \ulcorner \phi \urcorner \dot{\vee} \ulcorner \psi \urcorner \qquad (\mathsf{Gdl}_{\vee})$$
$$\ulcorner \neg \phi \urcorner = \dot{\neg} \ulcorner \phi \urcorner \qquad (\mathsf{Gdl}_{\neg})$$
$$\ulcorner \bot \urcorner = \dot{\bot} \qquad (\mathsf{Gdl}_{\bot})$$
$$\ulcorner \forall \mathsf{x}_i : \sigma . \phi \urcorner = \dot{\forall} \mathsf{v}_i^{\sigma} : \sigma . \ulcorner \phi \urcorner \qquad (\mathsf{Gdl}_{\forall})$$
$$\ulcorner \mathsf{x}_n \urcorner = \mathsf{inj}_{\sigma}(\mathsf{v}_n^{\sigma}) \quad \textit{where } \mathsf{x}_n \in \mathbf{Var}_{\sigma} \qquad (\mathsf{Gdl}_{\mathsf{x}})$$
$$\ulcorner s \approx t \urcorner = \ulcorner s \urcorner \dot{\approx}_{\sigma} \ulcorner t \urcorner \quad \textit{where } s,t \in \mathbf{Term}_{\sigma} \qquad (\mathsf{Gdl}_{\approx})$$
$$\ulcorner f(t_1,...,t_n) \urcorner = \dot{f}(\ulcorner t_1 \urcorner,...,\ulcorner t_n \urcorner) \qquad (\mathsf{Gdl}_{f})$$
$$\ulcorner P(t_1,...,t_n) \urcorner = \dot{P}(\ulcorner t_1 \urcorner,...,\ulcorner t_n \urcorner) \qquad (\mathsf{Gdl}_{P})$$

With our Gödel encoding at hand, we can now show that $\dot{\mathscr{T}}$ contains a truth predicate $\mathbf{T}[\phi]$ for $\mathscr{T}$, namely the formula $\mathsf{empty} \dot{\models} \ulcorner \phi \urcorner$. To this end, we have the following result.

Theorem 2 (Truth Predicate)**.**

$$\forall \phi \in \mathbf{Form}^{\mathscr{T}} . \Big(\dot{\mathscr{T}} \models \phi \leftrightarrow (\mathsf{empty} \dot{\models} \ulcorner \phi \urcorner) \Big)$$

□

In order to prove Theorem 2, we strengthen its statements, such that it holds not only for the empty reflective interpretation empty, but also for every reflective interpretation built from terms empty, and $\lambda e.\mathsf{push}_{\sigma}(e, \mathsf{v}_i^{\sigma}, \mathsf{x}_i^{\sigma})$. This is necessary, as (i) our axiomatisation of $\dot{\forall}_{\sigma}$ defines the meaning of a quantifier in terms of a different reflective environment and (ii) we relate finitary representation of variables (based on v_0^{σ} and $\lambda x.\mathsf{next}_{\sigma}(x)$) to the infinite set of variables used for standard first-order logic syntax. For the detailed proof of Theorem 2, we refer to the extended version of this paper [26].

4 Induction by Reflection

We next show how to build a finite theory that entails the first-order induction scheme $\mathbf{I}_{\tau}$. For the sake of simplicity we will first have a look at the rather familiar case of Peano Arithmetic, and present a generalisation of the same approach in the following subsection.

4.1 Natural Numbers

In order to finitely axiomatise **PA**, we need a finite fragment of **PA** to start with. The obvious choice is **Q**, which we define as **PA** without the induction formulas. We then build the reflective extension $\dot{\mathbf{Q}}$, with the following two essential properties. First, it has a sort form of formulas, hence we can quantify over this sort. Second, we have a truth predicate $\dot{\mathbf{Q}}$ for **Q**, which means we can represent an arbitrary formula of **Q** in a single term in $\dot{\mathbf{Q}}$.

Now we can define a conservative extension $\ddot{\mathbf{Q}}$ of **PA**. Therefore we will add the following axiom to $\dot{\mathbf{Q}}$; we call this axiom as the *reflective induction axiom*:

$$\forall \phi : \mathsf{form} . \Big(\mathbf{True}[\phi, 0] \wedge \forall n : \mathsf{nat} . (\mathbf{True}[\phi, n] \rightarrow \mathbf{True}[\phi, \mathsf{s}n]) \rightarrow \forall n : \mathsf{nat} . \mathbf{True}[\phi, n] \Big) \qquad (\dot{\mathbf{I}}_{\mathsf{nat}})$$

where $\mathbf{True}[\phi, n] := (\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, n) \dot{\models} \phi)$. Thus, we define $\ddot{\mathbf{Q}}$ as

$$\ddot{\mathbf{Q}} = \dot{\mathbf{Q}} \cup \{\dot{\mathbf{I}}_{\mathsf{nat}}\}$$

Theorem 3. $\ddot{\mathbf{Q}}$ *is a conservative extension of* **PA**

Proof. We will need the following auxiliary formula:

$$\dot{\mathscr{T}} \models (\mathsf{push}_{\sigma}(e, \ulcorner \mathsf{x}_i \urcorner, t) \dot{\models} \ulcorner \phi[\mathsf{x}_i] \urcorner) \leftrightarrow (e \dot{\models} \ulcorner \phi[t] \urcorner) \tag{1}$$

This formula holds by induction over ϕ. Proving that $\ddot{\mathbf{Q}}$ is a conservative extension of **PA** reduces showing that every formula in $\mathbf{Form}^{\mathbf{PA}}$ is provable in $\ddot{\mathbf{Q}}$ iff it is provable in **PA**. We next prove both directions of this property.

(1) $\forall \phi \in \mathbf{Form}^{\mathbf{PA}}.(\mathbf{PA} \models \phi \implies \ddot{\mathbf{Q}} \models \phi)$ To this end, we show that all axioms of **PA** are derivable in $\ddot{\mathbf{Q}}$. Since **Q** is a subset of both **PA** and $\ddot{\mathbf{Q}}$, we only need to deal with the induction axioms. Let $\phi[0] \wedge \forall n.(\phi[n] \to \phi[n+1]) \to \forall n.\phi[n]$ be an arbitrary instance of the first-order mathematical induction scheme $\mathbf{I}_{\mathsf{nat}}$. Let us instantiate the reflective induction axiom ($\dot{\mathbf{I}}_{\mathsf{nat}}$) with $\ulcorner \phi[\mathsf{x}_0] \urcorner$. We obtain

$$\begin{aligned}\ddot{\mathbf{Q}} \vdash & \mathbf{True}[\ulcorner \phi[\mathsf{x}_0] \urcorner, 0] \wedge \\ & \forall n : \mathsf{nat}.(\mathbf{True}[\ulcorner \phi[\mathsf{x}_0] \urcorner, n] \to \mathbf{True}[\ulcorner \phi[\mathsf{x}_0] \urcorner, \mathsf{s}n]) \\ & \to \forall n : \mathsf{nat}.\mathbf{True}[\ulcorner \phi[\mathsf{x}_0] \urcorner, n]\end{aligned}$$

which expands to

$$\begin{aligned}\ddot{\mathbf{Q}} \vdash & (\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, 0) \dot{\models} \ulcorner \phi[\mathsf{x}_0] \urcorner) \wedge \\ & \forall n : \mathsf{nat}.((\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, n) \dot{\models} \ulcorner \phi[\mathsf{x}_0] \urcorner) \to (\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, \mathsf{s}n) \dot{\models} \ulcorner \phi[\mathsf{x}_0] \urcorner)) \\ \to & \forall n : \mathsf{nat}.(\mathsf{push}_{\mathsf{nat}}(\mathsf{empty}, \mathsf{v}_0^{\mathsf{nat}}, n) \dot{\models} \ulcorner \phi[\mathsf{x}_0] \urcorner)\end{aligned}$$

By formula (1), we can derive

$$\begin{aligned}\ddot{\mathbf{Q}} \vdash & (\mathsf{empty} \dot{\models} \ulcorner \phi[0] \urcorner) \wedge \\ & \forall n : \mathsf{nat}.((\mathsf{empty} \dot{\models} \ulcorner \phi[n] \urcorner) \to (\mathsf{empty} \dot{\models} \ulcorner \phi[\mathsf{s}n] \urcorner)) \\ \to & \forall n : \mathsf{nat}.(\mathsf{empty} \dot{\models} \ulcorner \phi[n] \urcorner)\end{aligned}$$

Applying Theorem 2, the fact that $\lambda x.(\mathsf{empty} \dot{\models} x)$ is our truth predicate, we finally get

$$\ddot{\mathbf{Q}} \vdash \phi[0] \wedge \forall n : \mathsf{nat}.(\phi[n] \to \phi[\mathsf{s}n]) \to \forall n : \mathsf{nat}.\phi[n]$$

(2) $\forall \phi.(\ddot{\mathbf{Q}} \models \phi \implies \mathbf{PA} \models \phi)$ We prove by contraposition. Suppose we have some formula ϕ such that $\mathbf{PA} \not\models \phi$. Hence there is a counter-model $\mathscr{M}$, such that $\mathscr{M} \models \mathbf{PA}$ but $\mathscr{M} \not\models \phi$. Since $\mathbf{Q} \subset \mathbf{PA}$, it holds that $\mathscr{M} \models \mathbf{Q}$. Thanks to Section 3.3 we can extend the model $\mathscr{M}$ to the reflective model $\dot{\mathscr{M}}$ such that $\dot{\mathscr{M}} \models \dot{\mathbf{Q}}$, and that $\dot{\mathscr{M}} \not\models \phi$. We are thus left with establish that $\dot{\mathscr{M}}$ is a model of $\ddot{\mathbf{Q}}$.

In $\dot{\mathscr{M}}$ the sort form is interpreted as the actual set of formulas **Form**. Therefore let $\phi[\mathsf{x}_0]$ be an arbitrary of these formulas. Since $\mathscr{M} \vDash \mathbf{PA}$, we have that $\dot{\mathscr{M}} \vDash \mathbf{PA}$, which implies that

$$\dot{\mathscr{M}} \vDash \phi[0] \wedge \forall n : \mathsf{nat}.(\phi[n] \to \phi[\mathsf{s}n]) \to \forall n : \mathsf{nat}.\phi[n]$$

By Theorem 2, we get

$$\begin{aligned}\dot{\mathscr{M}} \vDash &(\mathsf{empty} \mathrel{\dot{\vDash}} \ulcorner \phi[0] \urcorner) \wedge \\ &\forall n : \mathsf{nat}.((\mathsf{empty} \mathrel{\dot{\vDash}} \ulcorner \phi[n] \urcorner) \to (\mathsf{empty} \mathrel{\dot{\vDash}} \ulcorner \phi[\mathsf{s}n] \urcorner)) \\ \to &\forall n : \mathsf{nat}.(\mathsf{empty} \mathrel{\dot{\vDash}} \ulcorner \phi[n] \urcorner)\end{aligned}$$

which, using formula 1, can be rewritten to

$$\begin{aligned}\dot{\mathscr{M}} \vDash &\mathbf{True}[\phi[x_0], 0] \wedge \\ &\forall n : \mathsf{nat}.(\mathbf{True}[\phi[x_0], n] \to \mathbf{True}[\phi[x_0], \mathsf{s}n]) \\ \to &\forall n : \mathsf{nat}.\mathbf{True}[\phi[x_0], n]\end{aligned}$$

Since $\dot{\mathscr{M}}$ interprets form formulas exactly as the set **Form** and $\phi[x_0]$ is an arbitrary formula, we conclude that the reflective induction axiom $\dot{\mathbf{I}}_\tau$ holds for $\dot{\mathscr{M}}$. Therefore, $\dot{\mathscr{M}}$ models $\ddot{\mathbf{Q}}$ but not ϕ, which concludes our proof. □

4.2 Arbitrary datatypes

The result of Section 4.1 can be lifted to arbitrary datatypes. Therefore, we translate the meta-level definition of the induction scheme $\mathbf{I}_\tau$ for datatypes $\mathscr{D}_\tau$ to an equivalent reflective version. That is, for a theory $\mathscr{T}$, we build $\mathscr{T}'$ by adding the axiom $\dot{\mathbf{I}}_\tau$ to $\dot{\mathscr{T}}$ for every datatype $\mathscr{D}_\tau$ in the theory, as follows:

$$\forall \phi : \mathsf{form}.\Big(\bigwedge_{c \in \mathbf{ctors}} \mathit{case}_{\phi,c} \to \forall x : \tau.\mathbf{True}[\phi, x] \Big) \qquad (\dot{\mathbf{I}}_\tau)$$

where

$$\begin{aligned}\mathit{case}_{\phi,c} &:= \mathop{\Forall}_{x_1,\ldots,x_n} \Big(\bigwedge_{i \in \mathit{recursive}_c} \mathbf{True}[\phi, \mathsf{x}_i] \to \mathbf{True}[\phi, c(x_1,\ldots,x_n)] \Big) \\ \mathit{recursive}_c &:= \{i \mid \mathbf{dom}_\Sigma(c,i) = \tau\} \\ \mathbf{True}[\phi, n] &:= (\mathsf{push}_\tau(\mathsf{empty}, \mathsf{v}_0^\tau, n) \mathrel{\dot{\vDash}} \phi)\end{aligned}$$

In the case of extending **Q** to a conservative extension of **PA**, the axioms of constructor disjointness $\mathsf{Disj}_{\mathsf{nat}}$, and injectivity $\mathsf{Inj}_{\mathsf{nat}}$ were already present in **Q**. Thus, for an arbitrary inductive theory $\mathscr{T}$ with inductive datatypes $\mathscr{D}_\mathscr{T}$, we define the reflective inductive extension $\ddot{\mathscr{T}}$ as follows:

$$\ddot{\mathscr{T}} = \mathscr{T} \cup \{(\dot{\mathbf{I}}_\tau), \mathsf{Disj}_\tau, \mathsf{Inj}_\tau \mid \mathscr{D}_\tau \in \mathscr{D}_\mathscr{T}\}$$

5 Experiments

In order to evaluate the practical viability of the techniques introduced Sections 3-4, we performed two set of experiments, denoted as **Refl** and **Ind** and described next.

Setup Note that our work introduces many new function symbols, and axioms which might blow up the proof search space, even if induction is not involved at all. Therefore, in our first experiment **Refl** we wanted to evaluate the feasibility of reasoning in the reflective extension of a theory.

Refl itself consists of two groups of benchmarks $\mathbf{Refl_0}$, and $\mathbf{Refl_1}$. **Refl** is the simplest one. For every theory $\mathscr{T}$ in some set of base theories, and every axiom $\alpha \in \mathscr{T}$ we try to proof the validity of $\dot{\mathscr{T}} \vdash (\mathsf{empty} \dot{\models} \ulcorner \alpha \urcorner)$. Since we established that $\lambda x.(\mathsf{empty} \dot{\models} x)$ is the truth predicate of $\mathscr{T}$, and the fact that α is an axiom, we know that these consequence assertions indeed hold. $\mathbf{Refl_1}$ involves reasoning in the reflective extension $\dot{\mathscr{T}}$ of some theory as well. But in this case not the reflective version of the axioms, but the reflective versions of some simple consequence of $\mathscr{T}$ are to be proven.

The benchmarks in the second experiment **Ind** are a set of crafted properties that require inductive reasoning. Every problem $\mathscr{T} \models \phi$ in this set of benchmarks is addressed in two ways. Firstly, proving it directly for the solvers that support induction natively, and secondly, translating the problem to $\ddot{\mathscr{T}} \models \phi$. For a description of the exact axioms and conjectures used in each of our benchmarks we refer to the extended version of this paper [26].

All benchmarks, as well as a program for generating reflective, and reflective inductive extensions of theories, and Gödel encodings for conjectures can be found at GitHub[1]. As the different solvers we used for evaluation support different input formats, our tool supports serializing problems into these various formats.

We used two (non-disjoint) sets of solvers. Firstly, solvers that support induction natively, and secondly various general-purpose theorem provers that are able to deal with multi-sorted quantified first-order logic, hence induction using the reflective extension.

The solvers considered where the SMT-solvers CVC4 and Z3, the superposition-based first-order theorem prover VAMPIRE, the higher-order theorem prover ZIPPERPOSITION that uses a combination of superposition and term rewriting, and the inductive theorem prover ZENO, that is designed to proof inductive properties of a HASKELL-like programming language. Since VAMPIRE in many cases uses incomplete strategy, per default it was run with a complete strategy forced as well. This configuration if referred to as VAMPIRECOMPLETE. ZIPPERPOSITION supports replacing equalities by dedicated rewrite rules, which comes at the cost of the theoretical loss of some provable problems, but yields a significant gain of performance in practice. ZIPPERPOSITION with these rewrite rules enabled will be referred to as ZIPREWRITE. CVC4 allows for theory exploration which was shown to be helpful for inductive reasoning in [12]. CVC4 with this heuristic enabled is referred to as CVC4GEN.

We ran each solver with a timeout of 10 seconds per problem.

Results In the first part of Table 1 we can see the results of solvers proving reflective versions of axioms. What is striking is that the SMT solvers CVC4, and Z3, can solve all benchmarks of this category, while the problem seems to be harder for the saturation based theorem provers. Further ZIPREWRITE does pretty well in this class of benchmarks as well. A potential reason for this difference in performance between the ordinary saturation approach and ZIPREWRITE might have to do with the following: For ZIPREWRITE equalities for function definitions of the reflective extensions are translated to rewrite rules that are oriented in way that they would intuitively be oriented by a human, this means that for example the axiom ($\mathsf{Ax}_{\mathsf{eval}_f}$) can be evaluated as one would intuitively do. In contrast VAMPIRE, using superposition with the Knuth-Bendix simplification ordering will orient this equality in the wrong way, which means that it won't be able to evaluate it in the intuitive way, which might be the reason for the difference in performance.

[1] https://github.com/joe-hauns/msc-automating-induction-via-reflection

benchmark	CVC4	CVC4GEN	Z3	VAMPIRE	VAMPIRECOMPLETE	ZIPPERPOSITION	ZIPREWRITE
$\mathbf{Refl_0}$							
N+Leq+Add+Mul-ax0	✓	✓	✓	✓	✓	✓	✓
N+Leq+Add+Mul-ax1	✓	✓	✓	–	–	–	✓
N+Leq+Add+Mul-ax2	✓	✓	✓	✓	✓	–	✓
N+Leq+Add+Mul-ax3	✓	✓	✓	–	–	–	✓
N+Leq+Add+Mul-ax4	✓	✓	✓	✓	✓	–	✓
N+Leq+Add+Mul-ax5	✓	✓	✓	–	–	–	✓
N+L+Pref+App-ax0	✓	✓	✓	✓	✓	✓	✓
N+L+Pref+App-ax1	✓	✓	✓	✓	✓	–	✓
N+L+Pref+App-ax2	✓	✓	✓	–	–	–	–
N+L+Pref+App-ax3	✓	✓	✓	✓	✓	–	✓
N+L+Pref+App-ax4	✓	✓	✓	–	–	–	✓
$\mathbf{Refl_1}$							
eqRefl	✓	✓	✓	✓	✓	✓	✓
eqTrans	✓	✓	✓	–	–	–	✓
excludedMiddle-0	✓	✓	✓	✓	✓	✓	✓
excludedMiddle-1	✓	✓	✓	✓	✓	✓	✓
universalInstance	–	–	✓	✓	✓	✓	✓
contraposition-0	✓	✓	✓	✓	✓	✓	✓
contraposition-1	✓	✓	✓	–	–	–	✓
currying-0	✓	✓	✓	✓	✓	–	✓
currying-1	✓	✓	✓	–	–	–	✓
addGround-0	✓	✓	✓	✓	✓	✓	✓
addGround-1	✓	✓	–	–	–	✓	✓
addExists	–	–	–	–	–	–	✓
existsZeroAdd	–	–	–	–	–	–	–
mulGround	✓	✓	✓	–	–	–	✓
mulExists	–	–	–	–	–	–	✓
existsZeroMul	–	–	–	–	–	–	–
appendGround-0	✓	✓	✓	✓	✓	✓	✓
appendGround-1	✓	✓	✓	–	–	✓	✓
appendExists	–	–	–	–	–	–	✓
existsNil	–	–	✓	✓	✓	–	✓

Table 1: Results of the experiment **Refl**.

benchmark	CVC4	CVC4GEN	VAMPIRE	VAMPIRECOMPLETE	ZIPPERPOSITION	ZIPREWRITE	ZENO	CV̈C4	CVC4G̈EN	Z̈3	VAM̈PIRE	VAMPIREC̈OMPLETE	ZIPPERP̈OSITION	ZIPRËWRITE
addCommut	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
mulCommut	–	–	–	–	–	–	–	–	–	–	–	–	–	–
addAssoc	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
mulAssoc	–	–	–	–	–	–	–	–	–	–	–	–	–	–
addNeutral	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
addNeutral-0	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
addNeutral-1	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
mulZero	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	✓
distr-0	–	–	–	–	–	–	–	–	–	–	–	–	–	–
distr-1	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
leqTrans	–	–	–	–	–	–		–	–	–	–	–	–	–
zeroMin	✓	✓	✓	✓	✓	✓		–	–	✓	✓	–	–	✓
addMonoton-0	–	–	–	–	–	–		–	–	–	–	–	–	–
addMonoton-1	–	–	–	–	–	–		–	–	–	–	–	–	–
addCommutId	–	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
appendAssoc	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
appendMonoton	✓	✓	✓	✓	✓	✓	✓	–	–	–	–	–	–	–
allEqRefl	✓	✓	✓	✓	✓	✓	–	–	–	✓	–	–	–	–
allEqDefsEquality	✓	✓	–	–	✓	✓	–	–	–	–	–	–	–	–
revSelfInvers	–	–	–	–	✓	–	–	–	–	–	–	–	–	–
revAppend-0	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
revAppend-1	–	–	–	–	–	✓	–	–	–	–	–	–	–	–
revsEqual	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Table 2: Lists the results of running solvers on the benchmark set **Ind**. For every solver SLVR that supports full first-order logic with equality as input, there is a solver SL̈VR using the reflective inductive theory as an input instead of using the solvers native handling of induction. The greyed out cells mean that the problem cannot be translated to the solvers input format.

The second part of the table shows that the performance of the SMT-solvers drops as soon as more complex reasoning is involved. Especially the problems with conjectures involving existential quantification[2] are hardly solved by the SMT solvers. This is not surprising since SMT solvers target at solving quantifier-free fragments of first-order logic.

Table 2 lists the results of the final experiment **Ind**. As the first experiments have shown reasoning in the reflective theories is hard even for very simple conjectures, it is not surprising that it is even harder for problems that require inductive reasoning to solve. Nevertheless there are some problems that can be solved using the reflective inductive extension instead of built-in induction heuristics. The most striking result is that Z3 is able to solve benchmarks that involve induction, even though it is a SMT-solver without any support for inductive reasoning.

[2]These problem ids contain the substring “exists” in their id.

6 Conclusion

It is mathematical practice to define infinite sets of axioms as schemes of formulas. Alas these schemes of axioms are not part of standard input syntax of today's theorem proves. In order to circumvent this shortcoming, we developed a method to express these schematic definitions in the language of first-order logic by means of a conservative extension, which we called the reflective extension of a theory. We showed that this reflective extension is indeed a conservative extension of the base theory. It contains a truth predicate which allows us to quantify over formulas within the language of first-order logic.

We replaced the first-order induction scheme of **PA** by the axioms needed for the reflective extension and a single additional axiom, called the reflective induction axiom. We proved that the resulting theory is indeed a conservative extension of **PA**. Further, we demonstrated how to replace the induction scheme of a theory with arbitrary inductive datatypes. This kind of conservative extension is what we called the reflective inductive extension.

Our experiments show that reasoning in the reflective extension of a theory is hard for modern theorem provers, even for very simple problems. Despite the poor performance in general, we have a positive result serving as a proof of concept of our method, namely that the SMT-solver Z3, which does not support induction natively was able to solve problems that require inductive reasoning.

Investigating our encoding in relation with the proof systems supported by the Dedukti framework [14] is an interesting line for further work. Further we are interested to explore which different proof search heuristics can be used to make our technique feasible for practical applications.

Acknowledgements This work has been supported by the ERC consolidator grant 2020 ARTIST 101002685, the ERC starting grant 2014 SYMCAR 639270, the EPSRC grant EP/P03408X/1, and the ERC proof of concept grant 2018 SYMELS 842066.

References

[1] Raymond Aubin (1979): *Mechanizing Structural Induction Part II: Strategies*. *Theor. Comput. Sci.* 9, pp. 347–362, doi:10.1016/0304-3975(79)90035-5.

[2] Siani Baker, Andrew Ireland & Alan Smaill (1992): *On the Use of the Constructive Omega-Rule within Automated Deduction*. In: *LPAR'92*, *Lecture Notes in Computer Science* 624, Springer, pp. 214–225, doi:10.1007/BFb0013063.

[3] Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes & Uwe Waldmann (2018): *Superposition for Lambda-Free Higher-Order Logic*. In: *IJCAR*, *Lecture Notes in Computer Science* 10900, Springer, pp. 28–46, doi:10.1007/978-3-319-94205-6_3.

[4] Ahmed Bhayat & Giles Reger (2020): *A Combinator-Based Superposition Calculus for Higher-Order Logic*. In: *IJCAR*, *Lecture Notes in Computer Science* 12166, Springer, pp. 278–296, doi:10.1007/978-3-030-51074-9_16.

[5] Ahmed Bhayat & Giles Reger (2020): *A Polymorphic Vampire - (Short Paper)*. In: *IJCAR*, *Lecture Notes in Computer Science* 12167, Springer, pp. 361–368, doi:10.1007/978-3-030-51054-1_21.

[6] Koen Claessen, Moa Johansson, Dan Rosén & Nicholas Smallbone (2012): *HipSpec: Automating Inductive Proofs of Program Properties*. In: *ATx'12/WInG'12*, *EPiC Series in Computing* 17, EasyChair, pp. 16–25. Available at `https://easychair.org/publications/paper/Kb7`.

[7] Véronique Cortier, Niklas Grimm, Joseph Lallemand & Matteo Maffei (2018): *Equivalence Properties by Typing in Cryptographic Branching Protocols*. In: *POST*, *Lecture Notes in Computer Science* 10804, Springer, pp. 160–187, doi:10.1007/978-3-319-89722-6_7.

[8] Simon Cruanes (2017): *Superposition with Structural Induction*. In: *FroCoS*, *Lecture Notes in Computer Science* 10483, Springer, pp. 172–188, doi:10.1007/978-3-319-66167-4_10.

[9] Mnacho Echenim & Nicolas Peltier (2020): *Combining Induction and Saturation-Based Theorem Proving*. *J. Autom. Reason.* 64(2), pp. 253–294, doi:10.1007/s10817-019-09519-x.

[10] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham & Mooly Sagiv (2019): *Inferring Inductive Invariants from Phase Structures*. In: *CAV*, *Lecture Notes in Computer Science* 11562, Springer, pp. 405–425, doi:10.1007/978-3-030-25543-5_23.

[11] Pamina Georgiou, Bernhard Gleiss & Laura Kovács (2020): *Trace Logic for Inductive Loop Reasoning*. *CoRR* abs/2008.01387, doi:10.34727/2020/isbn.978-3-85448-042-6_33.

[12] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl & Andrei Voronkov (2020): *Induction with Generalization in Superposition Reasoning*. In: *CICM*, *Lecture Notes in Computer Science* 12236, Springer, pp. 123–137, doi:10.1007/978-3-030-53518-6_8.

[13] Krystof Hoder, Nikolaj Bjørner & Leonardo Mendonça de Moura (2011): *μZ- An Efficient Engine for Fixed Points with Constraints*. In: *CAV*, *Lecture Notes in Computer Science* 6806, Springer, pp. 457–462, doi:10.1007/978-3-642-22110-1_36.

[14] Gabriel Hondet & Frédéric Blanqui (2020): *The New Rewriting Engine of Dedukti (System Description)*. In Zena M. Ariola, editor: *FSCD*, *LIPIcs* 167, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:16, doi:10.4230/LIPIcs.FSCD.2020.35.

[15] Leon Horsten (2011): *The Tarskian Turn: Deflationism and Axiomatic Truth*. Mit Press, MIT Press, doi:10.7551/mitpress/9780262015868.001.0001.

[16] Abdelkader Kersani & Nicolas Peltier (2013): *Combining Superposition and Induction: A Practical Realization*. In: *FroCoS*, *Lecture Notes in Computer Science* 8152, Springer, pp. 7–22, doi:10.1007/978-3-642-40885-4_2.

[17] Evgenii Kotelnikov, Laura Kovács, Giles Reger & Andrei Voronkov (2016): *The vampire and the FOOL*. In: *CPP*, ACM, pp. 37–48, doi:10.1145/2854065.2854071.

[18] Laura Kovács, Simon Robillard & Andrei Voronkov (2017): *Coming to terms with quantified reasoning*. In: *POPL*, ACM, pp. 260–270, doi:10.1145/3009837.3009887.

[19] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In: *CAV*, *Lecture Notes in Computer Science* 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[20] K. Rustan M. Leino (2012): *Automating Induction with an SMT Solver*. In: *VMCAI*, *Lecture Notes in Computer Science* 7148, Springer, pp. 315–331, doi:10.1007/978-3-642-27940-9_21.

[21] J. Strother Moore (2019): *Milestones from the Pure Lisp theorem prover to ACL2*. *Formal Aspects Comput.* 31(6), pp. 699–732, doi:10.1007/s00165-019-00490-3.

[22] Lauren Pick, Grigory Fedyukovich & Aarti Gupta (2020): *Automating Modular Verification of Secure Information Flow*. In: *FMCAD*, IEEE, pp. 158–168, doi:10.34727/2020/isbn.978-3-85448-042-6_23.

[23] Giles Reger, Martin Suda & Andrei Voronkov (2018): *Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning*. In: *TACAS*, *Lecture Notes in Computer Science* 10805, Springer, pp. 3–22, doi:10.1007/978-3-319-89960-2_1.

[24] Giles Reger & Andrei Voronkov (2019): *Induction in Saturation-Based Proof Search*. In: *CADE*, *Lecture Notes in Computer Science* 11716, Springer, pp. 477–494, doi:10.1007/978-3-030-29436-6_28.

[25] Andrew Reynolds & Viktor Kuncak (2015): *Induction for SMT Solvers*. In: *VMCAI*, *Lecture Notes in Computer Science* 8931, Springer, pp. 80–98, doi:10.1007/978-3-662-46081-8_5.

[26] Johannes Schoisswohl & Laura Kovacs (2021): *Automating Induction by Reflection*. Available at `https://arxiv.org/abs/2106.05066`.

[27] Andrei Voronkov (2014): *AVATAR: The Architecture for First-Order Theorem Provers*. In: *CAV*, *Lecture Notes in Computer Science* 8559, Springer, pp. 696–710, doi:10.1007/978-3-319-08867-9_46.

Countability of Inductive Types Formalized in the Object-Logic Level

Qinxiang Cao Xiwei Wu *

Abstract: The set of integer number lists with finite length, and the set of binary trees with integer labels are both countably infinite. Many inductively defined types also have countably many elements. In this paper, we formalize the syntax of first order inductive definitions in Coq and prove them countable, under some side conditions. Instead of writing a proof generator in a meta language, we develop an axiom-free proof in the Coq object logic. In other words, our proof is a dependently typed Coq function from the syntax of the inductive definition to the countability of the type. Based on this proof, we provide a Coq tactic to automatically prove the countability of concrete inductive types. We also developed Coq libraries for countability and for the syntax of inductive definitions, which have value on their own.

Keywords: countable, Coq, dependent type, inductive type, object logic, meta logic

1 Introduction

In type theory, a system supports inductive types if it allows users to define new types from constants and functions that create terms of objects of that type. The Calculus of Inductive Constructions (CIC) is a powerful language that aims to represent both functional programs in the style of the ML language and proofs in higher-order logic [16]. Its extensions are used as the kernel language of Coq [5] and Lean [14], both of which are widely used, and are widely considered to be a great success. In this paper, we focus on a common property, countability, of all first-order inductive types, and provide a general proof in Coq's object-logic. The techniques that we use in this formalization can be useful for formally proving other properties of inductive types in the future.

Here we show some examples of inductive types that we will use in this paper :

```
Inductive natlist :=  Cons : nat → natlist → natlist
                    | Nil  : natlist.
Inductive bintree :=  Node : nat → bintree → bintree → bintree
                    | Leaf : bintree.
Inductive expr := andp   : expr → expr → expr
                | orp    : expr → expr → expr
                | impp   : expr → expr → expr
                | falsep : expr
                | varp   : nat → expr.
```

As demonstrated above, `natlist` (list of natural numbers), `bintree` (binary trees with natural numbers as labels) and `expr` (the expressions of propositional language with `nat` as variable identifiers) can be defined inductively in the Coq proof assistant [5]. Here, the word "inductive" means that `natlist`, `bintree` and `expr` are the smallest of all sets that satisfy the above typing constraints. Specifically,

*Parallel authorship, equal contribution. Corresponding author: Qinxiang Cao.

E. Pimentel and E. Tassi (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 55–70, doi:10.4204/EPTCS.337.5

`natlist` is defined as the smallest set containing "`Nil`" and closed by the "`Cons`" constructor, which allows us to define a recursive function `length` on `natlist` so that `length Nil = 0`, `length (Cons x l) = 1 + length l`, and to prove related properties by induction on the structure of `natlist`.

Countability is a basic property of sets. To state that a set is countable means that it has the same cardinality as some subset of the set of natural numbers. For example, in the Henkin style proof of FOL completeness [8], one important step is to construct a maximal consistent set Ψ by expanding a consistent set Φ of propositions:

$$\begin{array}{rcll} \Psi_0 & := & \Phi & \\ \Psi_{n+1} & := & \Psi_n \cup \{\phi_n\} & (\text{if } \Psi_n \cup \phi_n \text{ is consistent}) \\ \Psi_{n+1} & := & \Psi_n & (\text{if } \Psi_n \cup \phi_n \text{ is inconsistent}) \\ \Psi & := & \bigcup_n \Psi_n & \text{where } \phi_0, \phi_1, \ldots \text{ are all FOL propositions.} \end{array}$$

In that proof, it is critical that the set of all FOL propositions is countable. The countability property allows us to enumerate all propositions as $\phi_0, \phi_1, \ldots$. As another example, we can show the countability of computable functions by proving the countability of untyped lambda expressions. Since the set of all functions from natural numbers to natural numbers is uncountable, there must exist at least one uncomputable function. Like FO-propositions and lambda expressions, many sets can be formalized as inductive types, and we focus on the countability of the inductive types in this paper.

Proving `natlist` to be countable is straightforward. (1) The only `natlist` of length 0 is `Nil`. (2) The `natlists` of length $(n+1)$ should be countable if those `natlists` of length n are countable (because the former set is isomorphic with the Cartesian product of $\mathbb{N}$ and the latter set). (3) By induction, the set of `natlists` of length n is countable for any n. (4) The set of all `natlists` is a union of countably many countable sets, and thus is countable (because we can easily construct an bijection from $\mathbb{N}^2$ to $\mathbb{N}$: $f(x,y) = 2^x(2y+1)-1$ and the construction does not even need the choice axiom). Similarly, natural number labeled binary trees with size n are countable for any n. Thus, the elements of `bintree` are countable. It is natural to apply the same proof idea to a more complex inductive type. We define a *rank* function to generalize the length function for `natlist` and the size function for binary trees (with some slight modification). For example, the `rank` function on `natlist` and `bintree` satisfies:

```
rank Nil = 1
rank (Cons n l) = rank l + 1
rank Leaf = 1
rank (Node n l r) = rank l + rank r + 1
```

We prove that given a fixed inductive type, its elements with rank less than n are countable[1]. Then, all elements of this type are also countable since the set is a union of countably many countable sets. One could write Coq tactics, which is a meta language, to describe our proof ideas above. In contrast, our target in this paper is to formally prove one single theorem in the object language for general inductive types' countability.

Handling general inductive types in Coq's object language is hard. Coq's object language, Gallina, has built-in support for recursive functions and inductive proofs, as long as they are about concrete inductive types. For general inductive types, we do not have such support, and even simple pattern match expressions are not easy to formalize. We choose to derive recursive functions and inductive proof

[1] In the general proof, we consider elements with rank less than n, not elements with rank equal to n

principles from general recursive functions. Using concrete inductive types as an example, `natlist`'s general recursive function is:

```
natlist_rect
   : forall P : natlist → Type,  P Nil →
     (forall (n : nat) (l : natlist), P l → P (Cons n l)) →
     forall l : natlist, P l
```

It satisfies, for any (maybe dependently typed) P and F_0, F_1,

```
natlist_rect P F_0 F_1 Nil = F_0
natlist_rect P F_0 F_1 (Cons n l) = F_1 n l (natlist_rect P F_0 F_1 l)
```

We generalize the combination of `natlist_rect` and the two equalities above, and develop our proofs based on them.

Theoretically, it is more difficult to do something at the object-logic level than at the meta-logic level. Any proof formalized at the object-logic level is a Coq function from its assumptions to its conclusion, according to Curry-Howard correspondence. One can always develop a corresponding meta-language function that implements the "same" functionality. In contrast, some statements are only provable in a meta-logic, but are unprovable in the object logic. Martin Hofmann and Thomas Streicher showed that the principle of uniqueness of identity proofs is not derivable in the object logic itself [12].

Practically, our proof automation, which uses object-logic proofs, is more efficient than proof generators written in a meta-language. Our tool can prove `expr` countable in Coq in 0.089 seconds but a tactic-based proof will take 1.928 seconds to finish the proof[2] (see Coq development for more details). This result arises because our tool only requires Coq to typecheck one theorem with its arguments, but proofs, either proof scripts or proof terms, generated by a meta-language generator require Coq to typecheck every single proof step.

In this paper, we formalize the general proof of countability theorem mentioned above, using Coq, and automate our proof to avoid repeating the long proof process. There are several ways in which this goal may be achieved. One is to use external tools to generate the operations and proofs of corresponding lemmas. For example, DBGen [17] generates single-variable substitution operations, and Autosubst2 [20] can generate substitution-related definitions and Coq proof terms. Another approach is to use the internal facility of theorem provers, which is written in a built-in meta language, to generate proof terms or proof scripts. Brian Huffman and Alexander Krauss (old datatype), and Jasmin Blanchette (BNF datatype) have developed tactics, which is a meta language, to prove datatypes countable [21] in Isabelle/HOL.

In comparison, an object logic proof of "for any possible T, $P(T)$ holds" is one singleton proof term of type $\forall T, P(T)$. A meta-logic proof is a meta-level program (with probably more expressiveness power) which takes T as its input and outputs a proof term of $P(T)$, which could be huge. Intuitively, the former one directly *states* that $\forall T, P(T)$ is true, while the latter one is an oracle which can *step-by-step explain* why $P(T)$ holds for a concrete T. Arthur Azevedo de Amorim's implementation [2] is the only object-level proof of countability before our paper. He used indices to number constructors of an inductive type. In other words, he formalized the syntax of inductive types and deeply embedded the syntax (in some sense, the meta language) in Coq's object language. As a consequence, his proof involves complicated reasoning about indices' equivalence, type's equivalence and dependent type issues—if two Coq types $T_1 = T_2$, T_1's elements are not automatically recognized as T_2's elements by Coq's type checker. Our formalization shows that such proof-reflection technique is not needed for a general countability proof.

[2]Processor: 2.3 GHz 8-Core Intel Core i9; Memory: 16 GB 2400 MHz DDR4.

Contributions. Our main contributions are a Coq formalized general countability proof for first order inductive types, and an automatic tactic for proving inductive types countable. We do not need any external tool to generate definitions, proof terms, or proof scripts, and our proof itself does not involve complicated dependently typed reasoning about type equalities. We also developed Coq libraries for countability and for a syntax of inductive definitions, which have values of their own. All of our proofs are formalized axiom-free, and our proof of countability theorem can be used in the completeness proof of separation logics, a Coq formalization for an early paper [6].
Outline. In Section 2, we will clarify our Coq definition of countability and our formalization of the syntax of first order inductive type definitions. In Section 3, we present our general countability theorem and our proof. In Section 4, we introduce our automatic tactic for proving concrete inductive types countable. We discuss related works in Section 5 and conclude in Section 6.

2 Preliminaries

In this section, we present our formal definition of countable (Section 2.1), the syntax of first order inductive types (Section 2.2), and general recursive functions (Section 2.3). We will also list their important properties, that we prove in our Coq library.

2.1 Countable

We define the type T to be countable if and only if there exists an injection from T to natural numbers, which means T is either finite or countably infinite. Here, an injection is a relation that keeps the injective property and functional property. This `Countable` is the definition used in our final theorem, but we use an auxiliary definition, `SetoidCountable`, in our proof. For the countability proof of inductive type T, we need to prove that $\{x : T \mid \mathsf{rank}(x) < n\}$ is countable for any n. In Coq, an element in $\{x : T \mid \mathsf{rank}(x) < n\}$ is a dependently typed tuple (x, p), where $x \in T$ and p is a proof of $\mathsf{rank}(x) < n$. Two such dependently typed tuples (x_1, p_1) and (x_2, p_2) are equal if $x_1 = x_2$, and p_1 and p_2 are identical proof terms. Proving $\{x : T \mid \mathsf{rank}(x) < n\}$ `Countable` requires us to show whether two proofs, p_1 and p_2, of $\mathsf{rank}(x) < n$ are identical. Using `SetoidCountable` avoids that kind of reasoning about proof terms, and avoids using the "proof-irrelevance" axiom in some sense[3]. The definition of $\{x : T \mid \mathsf{rank}(x) < n\}$ being `SetoidCountable` is straightforward: there exists a function f from $\{x : T \mid \mathsf{rank}(x) < n\}$ to natural numbers, so that if $f(x_1, p_1) = f(x_2, p_2)$ then $x1 = x2$.

```
Definition image_defined {A B} (R: A → B → Prop): Prop :=
    forall a, exists b, R a b.
Definition partial_functional {A B} (R: A → B → Prop): Prop :=
    forall a b1 b2, R a b1 → R a b2 → b1 = b2.
Definition injective {A B} (R: A → B → Prop): Prop :=
    forall a1 a2 b, R a1 b → R a2 b → a1 = a2.
Record injection (A B: Type): Type := {
    inj_R:> A → B → Prop;
    im_inj: image_defined inj_R;
    pf_inj: partial_functional inj_R;
    in_inj: injective inj_R }.
```

[3] `Axiom proof_irrelevance : forall (P:Prop) (p1 p2:P), p1 = p2.`

```
Definition Countable (T : Type) := injection T nat.
Record Setoid_injection
  (A B: Type) (RA: A → A → Prop) (RB: B → B → Prop) := ...
    (* RA and RB are equivalence relations on A and B resp. *)
Definition SetoidCountable (A: Type) {RA: A → A → Prop}: Type :=
  @Setoid_injection A nat RA (@eq nat).
    (* RA is an equivalence relation on A *)
    (* @SetoidCountable A RA if the quotient set A/RA is countable. *)
```

In our countability library, we prove that products of two countable types and unions of countably many countable types are countable. We prove that the composition of two injections is still an injection (see `injective_compose` below), and if $f \circ g$ is an injection then f is an injection (see `injective_compose_rev` below). We also prove their setoid versions, but omit them here. We define "bijection" and prove some elementary properties about bijection and injection. For connections between `SetoidCountable` and `Countable`, we prove that any `Setoid_injection` on Coq's builtin equality is an `injection`, and thus any `SetoidCountable` type w.r.t. Coq's builtin equality is also `Countable`.

```
Lemma injective_compose {A B C} (R1: A → B → Prop) (R2: B → C → Prop):
    injective R1 → injective R2 → injective (compose R1 R2).
Lemma injective_compose_rev {A B C} (R1: A → B → Prop) (R2: B → C → Prop):
    image_defined R2 → injective (compose R1 R2) → injective R1.
Lemma SetoidCountable_Countable {A: Type}:
    SetoidCountable A (@eq A) → Countable A.
```

Here, we use a relation rather than a function to provide the definition of `Countable`, for better usability and extensibility. For example, if we have a bijection from A to B and we have `Countable B`, we want to show that A is also countable. We can do it easily by relation, but we cannot do it under the definition of function, because of the problem of computability.

2.2 Syntax of inductive definition

In our formalization, we only consider *first order inductive definitions*. Not all inductive types have only countably many elements. Thus we exclude definitions like the following:

```
Inductive inf_tree: Type :=
| inf_tree_leaf: inf_tree
| inf_tree_node: nat → (nat → inf_tree) → inf_tree.
```

However, we try to focus on techniques of building dependently type functions from inductive definitions to nontrivial proof terms in this work. Thus we choose to exclude mutually inductive definitions and nested inductive definitions from our formalization, although we believe that we can extend our work in the future to handle these cases.

Formally, a first-order inductive type T is defined by a list of constructors, each of which is a first-order function with result type T. The argument types of constructors should be constant base type[4] or T itself. So we formalize the syntax of an inductive definition T as a list of dependently typed pairs of typing rules and constructors: `list (sigT (fun arg ⇒ constr_type arg T))`, we will

[4]Here, constant base type means other types which are countable.

call it Constrs_type later in this paper. We usually call the "typing rule" part arg, with a type list (option Type), and call a constructor constr, whose type depends on arg and is calculated by constr_type. For example, the definition of natlist has two branches. The type of Cons is: nat → natlist → natlist. It has two arguments, one of which is of type nat and the other is the inductive type natlist itself. Thus, this typing rule can be formalized as: [Some nat; None] (Some for a based type and None for the inductive type itself) and

```
constr_type [Some nat; None] natlist = nat → natlist → natlist
```

exactly describes the type of constructor Cons. Since the definition of natlist has two branches, this definition can be described by: [_[[Some nat; None], Cons]_; _[[], Nil]_].

2.3 General recursion

For an inductive type T, Coq generates T_rect, T_ind, T_rec and T_sind, which respectively correspond to elimination principles on Type, Prop, Set and SProp[5]. For our countability proof, T_rect is enough[5]. We define rect_type to compute the type of T_rect (we call it rect later in this paper) from an inductive definition. Similar to the definition of Constrs_type, we use rect_clause_type to compute each branch. Here we show the definitions of rect_type :

```
Fixpoint rect_type
       (T: Type) (constrs: Constrs_type) (P: T → Type): Type :=
  match constrs with
  | nil ⇒ forall x: T, P x
  | _[ arg, constr ]_ :: constrs0 ⇒
      rect_clause_type arg T P constr → rect_type T constrs0 P
  end.
```

For example, natlist_rect (which is generated by Coq) has type

```
forall P, rect_type natlist [ _[ [Some nat; None], Cons ]_; _[ [], Nil ]_ ] P
```

Knowledge of the type of general recursive function alone is not sufficient for building proofs. It is important that the computation result of a recursive function coincides with the definitions in its corresponding branch. For example, as mentioned in Section 1, natlist_rect satisfies:

```
natlist_rect P F_0 F_1 Nil = F_0
natlist_rect P F_0 F_1 (Cons n l) = F_1 n l (natlist_rect P F_0 F_1 l)
```

For general inductive types, we need to consider all possible ways of filling rect's arguments. We introduce apply_rect so that apply_rect T P constrs para (rect P) x fills rect's argument (like F_0 and F_1 above) in a parameterized way defined by para and calculates the result on $x : T$. Based on that, we define rect_correct:

```
rect_correct (T: Type) (constrs: Constrs_type)
  (rect: forall P, rect_type T constrs P): Prop
```

to be the following property: for any para and x, apply_rect T P constrs para (rect P) x equals to the recursive branch defined by para and x (and the recursive function apply_rect T P constrs

[5]The constant T_ind is always generated, whereas T_rec and T_rect may be impossible to derive, for example, when the sort is Prop. However, we only focus on the inductive type so no problems are encountered.

para rect itself). Detailed definitions of apply_rect and rect_correct involve complicated dependent type issue, and we defer them to Section 3.

In summary, our proof about inductive type's countability depends on and only depends on the following arguments and hypothesis:

- the Coq type T: Type;
- the inductive definition constrs: Constrs_type;
- the general recursive function rect: forall P, rect_type T constrs P;
- the characteristic equations rect_correctness: rect_correct constrs rect.

We develop three automatic tactics gen_constrs , gen_rect , apply_rect_correctness_gen to get constrs, rect and rect_correctness above from T.

- In order to get rect, we build a virtual induction proof on T and analyze that proof term. For example, one can prove "forall l: natlist, Type" by the following tactic:

  ```
  intro l; induction l; exact bool.
  ```

  This tactic will generate the following proof term:

  ```
  fun l : natlist ⇒ natlist_rect (fun _ ⇒ Type) bool (fun _ _ _ ⇒ bool) l
  ```

  Then our tactic analyzes this proof term to get natlist_rect.

- In order to get constrs, we get rect first and analyze the syntax of its type. For example,

  ```
  natlist_rect:
      forall P : natlist → Type,
        (forall (n0 : nat) (l1 : natlist), P l1 → P (Cons n0 l1)) →
        (P Nil) →
        (forall l : natlist, P l)
  ```

  From its assumptions, our tactic can generate

  ```
  [ _[ [Some nat; None], Cons ]_; _[ [], Nil ]_ ].
  ```

- In order to get rect_correctness, we only need to unfold the definitions of rect_correct and apply_rect, and prove the conclusion by reflexivity.

3 The countability theorem

As mentioned in Section 1, the main proof idea is to define a rank function from inductive type T to nat, and prove that $T_n \triangleq \{x : T \mid \mathsf{rank}(x) < n\}$ is countable for any n. This conclusion can be proved by induction on n. Its induction step is to construct an injection from T_{n+1} to the union of different products of T_n, and the latter one is countable since T_n is countable by the induction hypothesis. Using natlist and bintree as examples, we can construct injections[6]:

[6]Here we choose not to present the correct rank function, for reasons of conciseness. The real general rank function takes more arguments to be specialized on natlist and bintree; see Section 3.1. Also, redundant "* unit" and "+ void" are introduced by a uniform recursive definition for convenience.

```
natlist_{n+1} →
  nat * (natlist_n * unit) + (unit + void)
bintree_{n+1} →
  nat * (bintree_n * (bintree_n * unit)) + (unit + void)
```

Using `natlist` as an example, this construction of injection takes three steps:

- Defining a function from `natlist` to `nat * (natlist * unit) + (unit + void)`:

```
pattern_match (l: natlist) :=
  match l with
  | Cons n0 l1 ⇒ inl (n0, (l1, tt))
                          (* inl chooses the left branch of sum type *)
                          (* tt is the only element of unit *)
  | Nil ⇒ inr (inl (tt))
                         (* inr chooses the right branch of sum type *)
  end.
```

- Well-definedness of `pattern_match`:
  We thus prove that if `rank l < S n` and `pattern_match l = inl (n0, (l1, tt))`, then `rank l1 < n`. Thus, we can define a dependently typed function of the type below based on `pattern_match`.

```
{l: natlist | rank l < S n} →
  nat * ({l: natlist | rank l < n} * unit) + (unit + void)
```

  In other words, we define

```
pattern_match_DT:
  natlist_{n+1} → nat * (natlist_n * unit) + (unit + void)
```

- Injective property:
  We prove that the `pattern_match` function we defined is an injection.

In Section 3.1, we introduce our general definition of `rank` and `pattern_match`. In Section 3.2 and 3.3, we establish the injective property above. Specifically, we first prove that `pattern_match` itself is an injection (see Section 3.2) and use that conclusion to prove our final dependently typed version injective (see Section 3.3). Finally, we summarize our main theorem in Section 3.4.

3.1 Definitions

We first define the function that calculates the union of different products named `normtype`. As shown in the beginning of section 3, we want to project T_{n+1} into the union of different products of T_n. Also, the non-dependent type version `pattern_match` is a function from T to the union of different products of T. Thus, our definition of `normtype` is polymorphic. For example,

```
normtype natlist [ _[ [Some nat; None], Cons ]_; _[ [], Nil ]_ ] X
  = nat * (X * unit) + (unit + void)
normtype bintree [ _[ [Some nat; None; None], Node ]_; _[ [], Leaf ]_ ] X
  = nat * (X * (X * unit) + (unit + void)
```

In our definition of `normtype`, we analyze the inductive definition `constrs` (which means `[ _[ [Some nat; None], cons ]_; _[ [], nil ]_ ]` for `natlist`), use product types to represent each branch, and use sum types to connect them. For each branch, we use A when we meet `Some A` and use X when we meet `None`. We can therefore generate all `normtype` types with the same syntax tree as T like T_{n+1}.

As mentioned in Section 2, we need to define all recursive function (e.g. `rank`) and pattern match expressions (e.g. `pattern_match`) based on the general recursive function. Specifically, suppose `rect` is the general recursive function of type `T` with inductive definition `constrs`, we define `rank` and `pattern_match` by filling `rect`'s arguments through `apply_rect`.

When defining `rank`, each argument of `rect` is to add recursive calls' results together. For example, `size` (see Section 1) is the `rank` function of `bintree`. We can define it by filling `bintree`'s arguments:

```
size t = bintree_rect _ (fun n0 t1 r1 t2 r2 ⇒ r1 + r2 + 1) (1)
```

Here, in the recursive branch for constructor "`Node`", `n0` is the label, `t1` and `t2` are the left and right subtrees respectively, and `r1` and `r2` are the results of recursive calls: `size t1` and `size t2`; and in the recursive branch of constructor "`Leaf`", the return value is a constant `1`. In general, we can define these arguments of `rect` based on the syntax of inductive definitions. Using the example above, the typing information of "`Node`" is described by `[Some nat; None; None]` since `Node` has type:

```
nat → bintree → bintree → bintree
```

The first element `Some nat` corresponds to `n0` above; the second element `None` corresponds to `t1` and `r1` above; and the element `None` corresponds to `t2` and `r2`.

Defining `pattern_match` is more complicated. Here is how we define the `pattern_match` function for `natlist` (see the beginning of Section 3) by filling `rect`'s arguments.

```
pattern_match t =
  natlist_rect _ (fun n0 l1 r1 ⇒ inl (n0, (l1, tt))) (inr (inl tt))
```

Here, we put `inl` in the `Cons` branch since it is the first branch, and we put `inr` ∘ `inl` in the `Nil` branch since it is the second branch. That means we cannot define these two arguments of `natlist_rect` based only on `Cons`'s and `Nil`'s typing information. Specifically, if `constrs` can be decomposed into: `constrs1 ++ _[arg2,constr2]_ :: constrs3`, then `rect`'s arguments for the `arg2-constr2`-branch also depend on the length of `constrs1` and `constrs3`. For this reason, the definition of `apply_rect` must allow `para` (the parameterized way of filling `rect`'s arguments) to take `constrs1` and `constrs3` as its arguments.

In our real definition, `apply_rect`'s type is:

```
apply_rect T constrs P para rect_P : forall (t: T), P constrs t.
```

where (we omit `para`'s type first and introduce it later)

```
P: Constrs_type → T → Type;
rect_P: rect_type T constrs (P constrs)
```

Here `P` defines the dependently typed result type and `rect_P` is a specialized `rect` for computing results of "type" `P`. The most important parameter of `apply_rect` is `para`. Its type is:

```
para: forall constrs1 arg2 constr2 constrs3,
  let constrs := rev_append constrs1 (_[arg2, constr2]_ :: constrs3) in
  rect_clause_type arg2 T (P constrs) constr2
```

That is: if constrs, all branches of inductive definitions, can be decomposed into (rev constrs1), _[arg2, constr2]_, and constrs3, then para computes rect_P's argument for the branch of arg2 and constr2. Here, the function rev_append is the auxiliary function for defining a tail-recursive list reverse provided by Coq's standard library:

```
rev_append {A} (l l' : list A) : list A :=
  match l with
  | [] ⇒ l'
  | a :: l0 ⇒ rev_append l0 (a :: l')
  end.
```

It ensures that for any l and l', rev_append l l' = rev l ++ l. It is critical for us to use

```
rev_append constrs1 (_[arg2, constr2]_ :: constrs3)
```

instead of

```
constrs1 ++ _[arg2, constr2]_ :: constrs3
```

because it is much more convenient for building dependently typed inductive proofs and dependently typed recursive functions. When we apply an inductive proof on the structure of constrs, we can easily transform

```
rev_append constrs1 (_[arg2, constr2]_ :: constrs3)
```

to `rev_append (_[arg2, constr2]_ :: constrs1) constrs3`

because they are $\beta\eta\iota$-reduction to each other, and so can pass Coq's unification checking. But it is not the case for

```
constrs1 ++ (_[arg2, constr2]_ :: constrs3)
```

and `(constrs1 ++ [ _[arg2, constr2]_ ]) ++ constrs3`

which are equal, and can be proved equal, but they cannot pass Coq's unification checking.

In the end, we define paras for rank and pattern_match, and define these two functions for generic inductive types based on corresponding paras and apply_rect.

3.2 Injective property: the simple typed version

We have defined pattern_match, a function from T to normtype T constrs T (which is a union type of different product types). We prove it to be an injection in this subsection. Later, we will use this simplified conclusion to establish our ultimate goal: a Setoid_injection from T_{n+1} to normtype T constrs T_n.

To prove pattern_match to be injective (formalized as PM_inj below), we need to perform case analysis over both a and b.

```
Definition PM_inj T constrs rect: Prop :=
  forall a b: T, pattern_match a = pattern_match b → a = b.
```

Again, case analysis proofs is nontrivial since we are now reasoning about a generic inductive type, not a concrete one. We use a specialized rect to solve the problem:

```
Definition rect_PM_inj:
  rect_type T constrs
    (fun a ⇒ forall b, pattern_match a = pattern_match b → a = b)
:=  rect _.
```

Here, `rect_PM_inj` is a proof of a big implication proposition, whose conclusion is exactly `PM_inj` and whose assumptions are those case analysis branches. Thus, the proof of the injective property can be built by filling all those arguments of `rect_PM_inj`. Specifically, we decompose `constrs` into a form of `rev_append constrs1 constrs2` (at first `constrs1 = []` and `constrs2 = constrs`) and fill those arguments by an induction over `constrs2`. We successfully avoid dependently typed type-casting since we use `rev_append` in this proof and in `apply_rect`'s types. We omit proof details here.

We now finish the definition of a "simple" typed function, `pattern_match`, and the proof of its injective property. During the proof process, we encounter some problems regarding dependent types and solve them with the help of `rev_append`. To construct an injection from T_{n+1} (defined as $\{$`x : T | rank x < S n`$\}$) to `normtype T constrs` T_n, we need to prove linear arithmetic properties about `rank`. We could prove that dependently typed injective property in a similar way, but the dependent types with irrelevant proofs will trouble us much more. So we choose a different proof strategy.

3.3 Injective property: the dependent type version

Instead of developing a similar (but more complicated) proof of the injective property for the dependently typed version of `pattern_match`, we choose to *use* our injective proof above to build our dependently typed injective proof. In order to prove `pattern_match_DT` (the dependently typed version of `pattern_match`, a mapping from T_{n+1} to `normtype T constrs` T_n) to be injective, it is sufficient to show (Fig. 1 illustrates this proof strategy):

- `pattern_match` ∘ `proj1_sig = (normtype_map proj1_sig)` ∘ `pattern_match_DT`;
- `pattern_match` is injective;
- `proj1_sig` is injective.

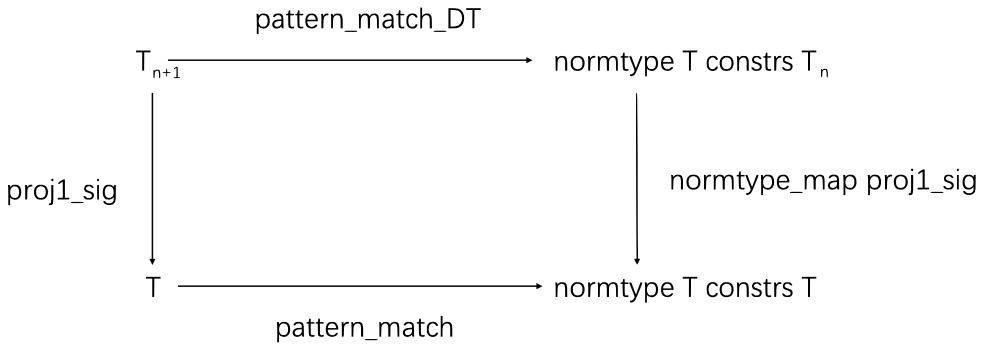


Figure 1: Proof strategy

Here, `proj1_sig` (defined by Coq standard library) removes the proof part of Coq's sigma types:

```
proj1_sig {A} {P: A → Prop} (e : sig P): A :=
  match e with exist _ a _ ⇒ a end
```

and we define `normtype_map` to apply `proj1_sig` on each part of `normtype T constrs` T_n.

The reasoning behind this proof strategy is straightforward. By the second and the third condition (`pattern_match` and `proj1_sig` are injective), we know that `pattern_match` ∘ `proj1_sig` is injective using theorem `injective_compose` (see Section 2.1). Thus, `(normtype_map proj1_sig)` ∘ `pattern_match_DT` is also injective according to the first condition (the diagram in Fig. 1 commutes).

In the end, pattern_match_DT must be an injection due to theorem injective_compose_rev (see Section 2.1).

We proved pattern_match to be injective in Section 3.3 and proj1_sig is obviously an injection; thus we only need to prove pattern_match ∘ proj1_sig = (normtype_map proj1_sig) ∘ pattern_match_DT, based on our definition of pattern_match_DT and normtype_map.

In our Coq formalization, the real type of pattern_match_DT is:

```
Definition pattern_match_DT n:
  forall t:T, rank t < S n → normtype T constrs Tn.
```

given type T, its inductive definition constrs and its general recursive function rect and rect_correctness. Using natlist as an example,

```
pattern_match_DT n (l: natlist):
   length l < S n → nat * (natlistn * unit) + (unit + void)
:= match l with
    | Cons n0 l1 ⇒
        fun H: length (Cons n0 l1) < S n ⇒
          inl (n0, (exist _ l1 SomeProof, tt))
    | Nil ⇒
        fun H: length Nil < S n ⇒
          inr (inl (tt))
    end.
```

The function above may be hard to read. It is equivalent to the following one, which is written in a less dependently typed way.

```
pattern_match_DT_demo n (l: natlist) (H: length l < S n) :=
  match l with
  | Cons n0 l1 ⇒ inl (n0, (exist _ l1 SomeProof, tt))
  | Nil ⇒ inr (inl (tt))
  end.
```

In their Cons branches, we need to provide a proof of length l1 < n at the place of SomeProof given H: length l < S n.

In order to define such a pattern_match_DT function for a generic inductive type T, we build its definition based on apply_rect and a "para" of the following type:

```
Definition pattern_match_para_DT n T constrs1 arg2 constr2 constrs3:
    let constrs := rev_append constrs1 (_[arg2, constr2]_ :: constrs3) in
    rect_clause_type arg2 T
       (fun t ⇒ rank t < S n → normtype T constrs Tn)
       constr2.
```

That is, we define pattern_match_DT by filling arguments of T's general recursive function rect. Given that constrs is decomposed into rev_append constrs1 (_[arg2, constr2]_ :: constrs3), this para above computes the arg2-constr2-argument of rect. Again, using the Cons argument of natlist_rect as an example, this para should have the following type:

```
forall n0 l1,
  length (Cons n0 l1) < S n →
  normtype natlist constrs_natlist natlistn
```

In the definition of pattern_match_para_DT and the proof of pattern_match ∘ proj1_sig = (normtype_map proj1_sig) ∘ pattern_match_DT, we repeatedly use rect_correctness to reason about rank. For the sake of space, we omit details here.

3.4 Main theorem

We have constructed the injection from T_{n+1} to normtype T constrs T_n in Coq. The definition of apply_rect helps us define most of the functions we need, and most of the intermediate lemmas can be reduced to properties of apply_rect and the corresponding para, which greatly simplifies our proof. We use rev_append to avoid type casting.

In summary, we use SetoidCountable as an auxiliary definition to prove the countability theorem. Specifically, we prove SetoidCountable T_n by induction over n and by the injection we just constructed. In the end, our main theorem is:

```
Variable (T: Type).
Variable (constrs: Constrs_type).
Variable (rect: forall P, rect_type T constrs P).
Variable (rect_correctness : rect_correct T constrs rect).
Hypothesis base_countable :
  Forall_type (fun s ⇒ (Forall_type option_Countable) (projT1 s)) constrs.
Theorem Countable_T : Countable T.
```

where the function Forall_type (provided by Coq's standard library) defines the universal predicates over lists

```
Inductive Forall_type {A : Type} (P : A → Type) : list A → Type :=
    | Forall_type_nil : Forall_type P nil
    | Forall_type_cons : forall (x : A) (l : list A),
              P x → Forall_type P l → Forall_type P (x :: l).
```

and the hypothesis base_countable says that all base types used in the inductive definition are countable. For example, this theorem proves that natlist is countable, as long as nat is countable.

4 Automatic proof systems

To make our result more practical and accessible, we developed Coq tactics to automatically prove inductive types countable. We define three tactics gen_constrs , gen_rect , apply_rect_correctness_gen to get constrs, rect and rect_correctness from T (previously mentioned in Section 2, see Coq development for more details). According to the list of conditions and hypothesis, we need in proof (as shown above in Section 3.4), the last thing that needs to be done is to prove base_countable, which means that all base types used in the definition of T are countable. This is done using a library of proved countability results and Coq assumptions. Here are two typical applications of our tactic Countable_solver:

```
Theorem Countable_expr : Countable expr.
Proof. intros. Countable_solver. Qed.
Theorem Countable_list : forall A: Type, Countable A → Countable (list A).
Proof. intros. Countable_solver. Qed.
```

5 Related work

We discuss three related studies in proving relevant properties for inductive types in Coq and one in Isabelle/HOL. What sets our formalization apart from this work is our proof at Coq's object logic level; only Deriving [2] develops object logic proofs like us. As a result, our proofs and automation instructions do not need other additional axioms, libraries, or tools, and can run with high efficiency.

- Theory Countable[1] : These researchers also used injection to represent the `Countable` relation. Their main idea was to construct an injection from data types to old data types which are countable. They could automatically prove the countability of data types which had nested and mutual recursion, and used other data types. However, the process and automatic tactics were formalized in a meta language.
- Autosubst[19] & Autosubst2[20] : Autosubst can automatically generate the substitution operations for a custom inductive type of terms, and prove the corresponding substitution lemmas. The library gives the enumerability of De Brujin substitution algebra[18]. Autosubst offers tactics that implement the normalization and decision procedure. They believe that it is hard to maintain or extend Ltac code, so that they proposed a new implementation of Autosubst which comes in the form of a code generator to generate Coq proof terms, and at the same time extends Autosubst's input language to mutual inductive sorts with multiple sorts of variables. Autosubst and Autosubst2 develop formalized proofs as tactics and external proof term generators, respectively, which can be treated as meta-level mappings from the syntax of inductive definitions to countability proof terms. In comparison, our proof is a Coq object-level mapping from inductive definitions to countability, but we do not support mutually inductive types at present.
- Undecidability[9] : Forster et. al. formalized the computational undecidability of the validity, satisfiability, and provability of first-order formulas following a synthetic approach based on the computation native to Coq's constructive type theory. They extended the library in 2020, to present a comprehensive analysis of the computational content of completeness theorems for first-order logic, considering various semantics and deduction systems[10]. They proved first-order logic's propositions countable in their completeness proof. They also formalized the syntax of inductive definitions[7], as we did, but they do not provide a general theorem of countability.
- Deriving [2] : Deriving proved inductive types countable in Coq. Deriving uses `countType` in the MathComp library, which provides a different definition of injection and `Countable`. Although this alternative definition requires injections to be Coq-computable functions, it is not a significant drawback comparing with our definitions when applying inductive type's countability. Deriving supports the proof of countability for mutually inductive types and nested inductive types. Their proof strategy is different from ours: they built an injection from every inductive type T to *finite-width trees*, which they defined as a Coq type and proved countable. More significantly, they provided two formalization of inductive definitions. One is like ours:

  ```
  constrs: list (sigT (fun arg ⇒ constr_type arg T));
  ```

  the other uses indices to number the constructors. In other words, the latter formalization is a deep embedding of the meta language (the syntax of inductive types) into Coq's object language. They

[7] This general formalization appears in their Coq development but they do not mention that in their paper.

carefully used computable functions over the deeply embedded meta language (since natural numbers' equality tests are computable, but types' equality tests are not computable) in their definitions and used the connection between these two formalizations to compute their final proof term. In their proofs, they need to reason about indices' equalities, types' equalities and relevant dependent type issues—if two Coq types $T_1 = T_2$, T_1's elements are not automatically recognized as T_2's elements by Coq's type checker. In comparison, our work shows that inductions over constructor lists do prove the conclusion using the "`rev-append` trick", and we do not need number-indexing and heavy-weighted proof reflection to bypass related difficulties in dependently type proofs.

6 Conclusions

We proved in Coq that a first-order inductive type is countable as long as all base types used in the definition are countable. Our definitions and proofs are all axiom-free. We provide an alternative way of thinking about solving dependent types at the object level. We developed very efficient tactics which use this countability theorem to prove concrete inductive types to be countable. Our formalization and automatic tactic still have room for expansion in the future. For example, our tactics do not work when applied to mutually recursive types. We believe that it is plausible to transform the mutually recursive types into primitive recursive types [11] and enhance our tactics. Our Coq development can be found at:

`https://github.com/QinxiangCao/Countable_PaperSubmission`

Ackownledgement This research is sponsored by National Natural Science foundation of China (NSFC) Grant No. 61902240 and Shanghai Pujiang Program 19PJ406000.

References

[1] Jasmin Blanchette Alexander Krauss, Brian Huffman: *This is a library of Countable Theory in Isabelle/HOL*. `https://devel.isa-afp.org/browser_info/current/HOL/HOL-Library/Countable.html`.

[2] A. A. de Amorim (2020): *Deriving Instances with Dependent Types*. *CoqPL 2020*.

[3] Abhishek Anand & Vincent Rahli (2014): *A Generic Approach to Proofs about Substitution*. In Amy P. Felty & Brigitte Pientka, editors: *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP '14, Vienna, Austria, July 17, 2014*, ACM, pp. 5: 1–5: 8, doi:10.1145/2631172.2631177.

[4] Brian Aydemir & Stephanie Weirich (2010): *LNgen: Tool support for locally nameless representations*.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin et al. (1999): *The Coq proof assistant reference manual*. *INRIA, version* 6(11).

[6] Qinxiang Cao, Santiago Cuellar & Andrew W. Appel (2017): *Bringing Order to the Separation Logic Jungle*. In Bor-Yuh Evan Chang, editor: *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, *Lecture Notes in Computer Science* 10695, Springer, pp. 190–211, doi:10.1007/978-3-319-71237-6_10.

[7] Adam Chlipala (2008): *Parametric higher-order abstract syntax for mechanized semantics*. In James Hook & Peter Thiemann, editors: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, ACM, pp. 143–156, doi:10.1145/1411204.1411226.

[8] Heinz-Dieter Ebbinghaus, Jörg Flum & Wolfgang Thomas (1994): *Mathematical logic (2. ed.)*. Undergraduate texts in mathematics, Springer, doi:10.1007/978-1-4757-2355-7.

[9] Yannick Forster, Dominik Kirst & Gert Smolka (2019): *On synthetic undecidability in coq, with an application to the entscheidungsproblem*. In Assia Mahboubi & Magnus O. Myreen, editors: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, ACM, pp. 38–51, doi:10.1145/3293880.3294091.

[10] Yannick Forster, Dominik Kirst & Dominik Wehr (2020): *Completeness Theorems for First-Order Logic Analysed in Constructive Type Theory*. In Sergei N. Artëmov & Anil Nerode, editors: *Logical Foundations of Computer Science - International Symposium, LFCS 2020, Deerfield Beach, FL, USA, January 4-7, 2020, Proceedings*, *Lecture Notes in Computer Science* 11972, Springer, pp. 47–74, doi:10.1007/978-3-030-36755-8_4.

[11] Peter Freyd (1990): *Recursive types reduced to inductive types*. In: *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 498–507, doi:10.1109/LICS.1990.113772.

[12] M. Hofmann & T. Streicher (1994): *The groupoid model refutes uniqueness of identity proofs*. In: *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pp. 208–212, doi:10.1109/LICS.1994.316071.

[13] Gyesik Lee, Bruno C. d. S. Oliveira, Sungkeun Cho & Kwangkeun Yi (2012): *GMeta: A Generic Formal Metatheory Framework for First-Order Representations*. In Helmut Seidl, editor: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *Lecture Notes in Computer Science* 7211, Springer, pp. 436–455, doi:10.1007/978-3-642-28869-2_22.

[14] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, *Lecture Notes in Computer Science* 9195, Springer, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[15] Rob Nederpelt & Herman Geuvers (2014): *Type theory and formal proof: an introduction*. Cambridge University Press, doi:10.1017/CBO9781139567725.

[16] Christine Paulin-Mohring (2015): *Introduction to the Calculus of Inductive Constructions*. In Bruno Woltzenlogel Paleo & David Delahaye, editors: *All about Proofs, Proofs for All*, *Studies in Logic (Mathematical logic and foundations)* 55, College Publications. Available at `https://hal.inria.fr/hal-01094195`.

[17] Emmanuel Polonowski (2013): *Automatically Generated Infrastructure for De Bruijn Syntaxes*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, *Lecture Notes in Computer Science* 7998, Springer, pp. 402–417, doi:10.1007/978-3-642-39634-2_29.

[18] Steven Schäfer, Gert Smolka & Tobias Tebbi (2015): *Completeness and Decidability of de Bruijn Substitution Algebra in Coq*. In Xavier Leroy & Alwen Tiu, editors: *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 67–73, doi:10.1145/2676724.2693163.

[19] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*. In Christian Urban & Xingyuan Zhang, editors: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, *Lecture Notes in Computer Science* 9236, Springer, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[20] Kathrin Stark, Steven Schäfer & Jonas Kaiser (2019): *Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions*. In Assia Mahboubi & Magnus O. Myreen, editors: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, ACM, pp. 166–180, doi:10.1145/3293880.3294101.

[21] Christian Sternagel & René Thiemann (2015): *Deriving class instances for datatypes*. *Arch. Formal Proofs* 2015. Available at `https://www.isa-afp.org/entries/Deriving.shtml`.

SMLtoCoq: Automated Generation of Coq Specifications and Proof Obligations from SML Programs with Contracts

Laila El-Beheiry Giselle Reis Ammar Karkour
Carnegie Mellon University, Qatar
loe@andrew.cmu.edu, giselle@cmu.edu, akarkour@andrew.cmu.edu

Formally reasoning about functional programs is supposed to be straightforward and elegant, however, it is not typically done as a matter of course. Reasoning in a proof assistant requires "reimplementing" the code in those tools, which is far from trivial. SMLtoCoq provides an automatic translation of SML programs and function contracts into Coq. Programs are translated into Coq specifications, and function contracts into theorems, which can then be formally proved. Using the Equations plugin and other well established Coq libraries, SMLtoCoq is able to translate SML programs without side-effects containing partial functions, structures, functors, records, among others. Additionally, we provide a Coq version of many parts of SML's basis library, so that calls to these libraries are kept almost *as is*.

1 Introduction

Programming language implementations are built for programming, so the aim is to provide useful libraries and constructs to make writing *code* as easy as possible. Proof assistants, on the other hand, are built for reasoning and as such the aim is to make writing *proofs* as easy as possible. As much as functional programs look similar to (relational or functional) specifications, if one wants to prove properties about an implemented program, it is necessary to "reimplement" it in the language of a proof assistant. This requires familiarity with both the programming language and the proof assistant, since not all programs are supported by reasoning tools (e.g. non-terminating programs are typically forbidden), and not all libraries are available in both tools.

In this work we present SMLtoCoq: a tool that *automatically translates SML code into Coq specifications*. Moreover, we extend SML with *function contracts which are directly translated into Coq theorems*. By using this tool, programs can be written using all conveniences of a programming language, and their contracts (functions' pre and post conditions) can be proved using the full power of a proof assistant. Our target audience are programmers fluent in functional programming, but with little expertise in proof assistants. By having programs automatically translated into Coq code that looks as similar as possible to SML, programmers can learn quickly how program specifications look like, thus lowering the entry barrier for using the proof assistant.

Even though both tools use a functional language, representing SML programs in Coq is complicated due to their various differences. The first immediate challenge is that SML programs may be partial and have side-effects, while Coq programs/specifications need to be pure and total. Another issue is that Coq requires recursive definitions to be structurally decreasing so that termination is guaranteed. SML programs can certainly diverge, and even if they terminate, it might be because of a more complicated argument than a straightforward structural induction. Coq does have extensions – such as the `Program` command – that are used to define non-structurally-decreasing, terminating recursive functions. However, a proof of termination must be provided. On top of these more fundamental problems, we have

E. Pimentel and E. Tassi (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 71–87, doi:10.4204/EPTCS.337.6

encountered many small mismatches between the two languages, such as how records are represented, and what type-checking can infer.

By using a number of Coq features and information available in SML's abstract syntax tree (AST), SMLtoCoq is able to translate an extensive fragment of *pure* SML (i.e. without side effects), including partial functions, records, and functors, among others. The resulting Coq specification looks very similar to the original code, so someone proving properties can easily map parts of specifications back to their program counterpart. Using the Equations library, we translate partial, mutually recursive, and recursive functions out of the box. Non-terminating functions are not accepted by Coq, but their translations type check.

Our contributions are:

- We extend HaMLet's implementation of SML to parse and type-check function contracts, written as function annotations. These are included in the SML's AST and translated into theorems.
- We design and implement a tool, SMLtoCoq, that is able to translate pure SML programs and contracts into Coq specifications and theorems completely automatically. Moreover, the translated code looks very similar to the original code.
- We implement in Coq the SML libraries[1]: `INTEGER`, `REAL`, `STRING`, `CHAR`, `Bool`, `Option`, `List`, `ListPair`, and `IEEEReal`.
- We provide many examples of translated code, including a case study where we translate non-trivial SML code and prove properties on the Coq output. This aligns with the intended workflow for the tool: translate SML code, then prove properties in Coq.

SMLtoCoq can be found at: `https://github.com/meta-logic/sml-to-coq/`

2 Infrastructure

In its core, SMLtoCoq implements a translation of SML's abstract syntax tree (AST) into Gallina's (Coq's specification language) AST – which is subsequently used to generate Gallina code. SML was chosen for being a language with an incredibly formal and precise definition, which helped in understanding precisely what fragments of the language were covered by our translation. Coq was chosen for being an established and powerful proof assistant, with many libraries and plugins available. In particular, it provides the Equations library which was crucial for efficiently automating the translation and generating correct code that looks close to SML.

SML's AST is obtained from HaMLet[2]. SML's implementation in HaMLet can be separated into three phases: *parsing*, *elaboration*, and *evaluation*. If a program's syntax is correct, *parsing* will succeed returning an AST with minimal annotation at each node, containing only their position in the source code. Other well-formedness conditions (e.g. non-exhaustive or redundant matches) and type-checking (i.e. static semantics) are computed during *elaboration*, which populates the annotations in the AST with more information. *Evaluation* will simply evaluate the program (i.e. dynamic semantics).

We use the AST after the elaboration phase, when annotations contain useful information such as inferred types and exhaustiveness of matches. Such information is crucial for the generated Coq code. We call evaluation to make sure the program executes correctly and terminates (i.e. no exceptions raised

[1] Some of these libraries are very pervasive and we needed their implementation to test the translation. This is why they were not translated using SMLtoCoq. See Section 2.3.

[2] `https://people.mpi-sws.org/~rossberg/hamlet/`

or infinite loops entered) before starting the translation, but the evaluation result is not used. SMLtoCoq is implemented in SML.

2.1 HaMLet

HaMLet is an SML implementation whose goal is to be an accurate implementation of the language definition [6] and a platform for experimentation. We chose this implementation for three main reasons:

1. It is faithful to SML's definition, and all deviations and design choices are thoroughly documented.
2. The resulting objects from each compilation step are easily accessible. Particularly, the AST after the elaboration phase could be obtained with a couple of function calls.
3. Due to the detailed documentation, HaMLet could be easily modified as per the translation needs.

HaMLet was extended with function contracts, written as code annotations in the following syntax:

```
(!! f input ==> output;
    REQUIRES: exp1;
    ENSURES:  exp2;      !!)
```

This contract must be placed immediately before `f`'s declaration. The first line includes the function name `f` followed by the variable bindings representing its `input`. These can be curried, uncurried, typed, or untyped, following the syntax of function parameters in SML. This is followed by `output`, which is one named variable, typed or not, representing the function's output. The variable names in `input` and `output` are in the scope of the contract only, and should not be confused with variables inside the function. `REQUIRES` and `ENSURES` are new keywords used for indicating the pre and post condition of the function, respectively. They are followed by SML boolean expressions `exp1` and `exp2`. The type of these expressions is enforced by the type checker. The variables used in `input` and `output` can be used in `exp1` and `exp2`. Variables in the function declaration are not available in the contract, but `exp1` and `exp2` can use functions or variables defined previously in the code.

In addition to changing HaMLet's lexer and parser, SML's AST was modified to account for functions with contracts. The node representing function declaration was augmented to hold the variable bindings and expressions from contracts. If the function has no contracts, these fields are empty.

Besides the implementation of function contracts, HaMLet needed to be modified so that the translation would be as faithful as possible. During parsing, constructs called *derived forms* are transformed into semantically equivalent code using other constructs. This makes the core language smaller. For example, `if-then-else` expressions are transformed into `case` expressions (which, in turn, are transformed into anonymous functions). As a result, the AST computed after parsing would not have nodes for `if-then-else`, and its translation would result in a different (although semantically equivalent) Coq specification. We have changed HaMLet to skip the transformation of derived forms, and added constructors to the AST to account for them. The constructors added were:

- Expressions
  - `()` and `(`e_1`, ..., `e_n`)`
  - `[`e_1`, ..., `e_n`]`
  - `case e of m`
  - `if` e_1 `then` e_2 `else` e_3
  - e_1 `andalso` e_2 and e_1 `orelse` e_2
  - e_1 `op` e_2 (infix expressions)
- Patterns: `()`, `(`p_1`, ..., `p_n`)`, `[`p_1`, ..., `p_n`]`, and p_1 `op` p_2 (infix patterns)
- Tuple types: t_1 `*` ... `*` t_n

- Function declarations (originally transformed into `val rec`): `fun id pats = e`
- Functor instantiation with inline specification (e.g. `FuncID (structure <strdesc>)`)
- Type definitions in signatures (e.g. `type t = s`, which was expanded to an inline signature)

SML's AST is annotated during elaboration phase. The annotation of the new constructs can happen in one of two ways: (1) its equivalent form is constructed, elaborated, and we use the resulting annotation interpreted in the context of the derived form; or (2) a new elaboration case is implemented for the construct. We have used a combination of both approaches which incurred as few changes as possible in the elaboration phase. The annotations produced by both approaches are equivalent since they are added during the elaboration phase which only considers the static semantics up to which the derived form and its equivalent form are equivalent.

2.2 Coq

Coq's core language for writing specifications is called Gallina. To be able to generate Gallina's AST, we have implemented it as a datatype in our system. The implementation follows closely the grammar of Gallina's specification[3], with a few small changes described in what follows.

Extra term and pattern constructors were added for string, real, char, tuple, list, unit, and infix. SML expressions of those types are directly translated to Gallina using these constructors, and they can be directly mapped to Coq code either using common libraries or notations (e.g. `Datatypes`, `List`), or Coq libraries that we have implemented to match SML libraries (e.g. string, real, char). New term constructors for product types, and boolean conjunction and disjunction, were implemented for the same reasons as above. To be able to generate theorems, Gallina's AST includes the `Prop` operators for conjunction, disjunction, equality, and quantification. Finally, `match` terms need to have a field indicating exhaustiveness.

Some syntactical elements of Gallina were left out since there is no corresponding SML code which generates them. For example, type coercion (`:>`) and "or" patterns (p_1 `|` $p_2 \Rightarrow$ `t`).

Equations Equations [8] is a Coq plugin which allows a richer form of pattern-matching and arbitrarily complex recursion schemes in the setting of dependent type theory. Using this plugin, SMLtoCoq can generate concise and elegant code that is visually similar to SML. The output is considerably improved when compared to one generated in pure Gallina, and the derived proof principles can be used to reason about the high-level code. Among the main advantages of using Equations, is the ability to define partial functions using dependent types. Domain restrictions identified on the original SML function can be translated to a dependent type, which is added as one of the function's arguments. The resulting code has minimal overhead compared to its SML counterpart, not needing extra default values or ad-hoc constructs to handle unmatched cases. In addition to that, Equations provides a simple interface to handle non-trivial recursion where proofs of termination must be provided by the user.

2.3 Libraries

SML's basis library is included in most of its implementations and contains some of the more popular datatypes and functions. Since most SML programs will make use of some part of the basis library, we have implemented Coq equivalents to: `INTEGER`, `REAL`, `CHAR`, `Bool`, `STRING` (and partially `StringCvt`), `Option`, `List`, `ListPair`, and `IEEEReal`. They have the same interface as their SML

[3] `https://coq.inria.fr/distrib/current/refman/language/core/index.html`

counterpart, so function calls to these libraries can be translated almost "as is" to Coq. Most of the implementations build on existing Coq libraries, such as `List`, `ZArith`, `Ascii`, `Bool`, `Floats`, `String` etc.

Several of SML's basis library functions raise **exceptions** for a given set of inputs. However, Gallina is pure and does not have side effects. In particular, it does not support exceptions. In order to handle these cases, we return an **Axiom** instead of raising an exception. For example, the function `List.hd` in SML's basis library raises the exception `Empty` if an empty list is passed to it. The equivalent implementation of `List.hd` in our libraries return the axiom `EmptyException`, defined as:

```
Axiom  EmptyException : forall{a}, a.
```

Note that this means it is not possible to *catch* the exception, and renders Coq inconsistent. We are currently investigating possible solutions for exceptions, and side-effects in general (see Section 5).

A natural question to ask is why we have not used SMLtoCoq itself to translate SML's libraries. The reason is purely of a practical matter: function translation was being developed at the same time libraries were being implemented in Coq, and one development informed the other. Moreover, a lot of SML code relies on these libraries, so we needed equivalent ones in Coq to test our translation.

3 Translation

The translation from an SML's AST $\mathcal{S}$ into a Gallina's AST $\mathcal{G}$ is defined inductively on $\mathcal{S}$. Depending on the type of construct being translated, we rely on (local or global) auxiliary contexts. We describe in this section the relevant parts of the translation, including examples for each of them. The code in all figures in this section is exactly the one used and generated by SMLtoCoq, except for some line breaks and spaces removed to fit the pages. All the Coq code was tested with the following header, which imports the libraries discussed in Section 2.3, Equations, and sets generalization of variables by default.

```
Require Import intSml.       Require Import listSml.
Require Import realSml.      Require Import stringSml.
Require Import charSml.      Require Import boolSml.
Require Import optionSml.    Require Import listPairSml.
Require Import notationsSml.

From Equations Require Import Equations.
Generalizable All Variables.
```

More involved examples can be found in the `examples` folder in the repository[4]. In particular, we would like to highlight the `tree_proof.v` file which contains the translated code for functions on trees, and proofs of non-trivial theorems about these functions.

3.1 Overloaded operators

Some comparison and arithmetic operators in SML are overloaded for multiple types. For example, the equality check works for strings and integers: `val b = "a" = "a" andalso 5 = 3`.

Boolean equality checks for string and integers in Coq, denoted by `=?`, are defined in `string_scope` and `Z_scope`, respectively. However, the last opened scope shadows the first, and trying to use both at the same time fails. For example:

```
Require Import ZArith. Open Scope Z_scope.
Require Import String. Open Scope string_scope.
```

[4] https://github.com/meta-logic/sml-to-coq/tree/sml-to-coq-with-hamlet/examples

```
Require Import Bool.

Fail Definition b := ("a" =? "a") && (5 =? 3).
```

fails because Coq expects 5 to be of type string.

Operation overload is solved using typeclasses. We have defined several typeclasses for different sets of operators and instantiated them with the types supported. We also defined notations for the operators to be the same as SML whenever possible. For example, the typeclass below is instantiated for strings and integers (among others):

```
Class eqInfixes A : Type := {
  eqb : A → A → bool;
  neq : A → A → bool
}.
Infix "="  := eqb (at level 70).
```

3.2 Records

A record is a set of named fields typed by record types, for example:

```
{name = "Bob", age = 42}: {name: string, age: int}
```

In SML, records can occur as expressions, patterns, or types. When matching a record pattern, the user can specify the names of relevant fields and omit the remaining using ellipsis, as long as SML's type checker can infer the full record type. For example:

```
fun getAge (r: {name: string, age: int }) =
  case r of {age = x, ...} => x;
```

Gallina supports record types, however these must be declared using `Record`. So the `getAge` function above could be:

```
Record rec := { name : string; age : Z }.
Definition getAge (r : rec) := match r with
  {| name := _; age := x |} ⇒ x.
```

There are three important points that must be taken into consideration when translating records:

1. Any record expression, type, or pattern in an SML declaration might require a `Record` declaration preceding the translation, and record types must be replaced by the `Record` identifier.
2. The `Record` declaration automatically generates projection functions for each of the record's fields. As a result, field names cannot be reused in the code.
3. There is no Gallina equivalent to SML's ellipsis when pattern matching records. So the translation must make all record fields explicit in patterns.

To make sure all necessary records are declared in the translation, we make use of a *record context* associated with Gallina's AST. This context is split into a local and a global part. The global record context $\mathcal{R}_g$ contains all record types that already have a declaration in the AST. The local record context $\mathcal{R}_l$ is used in the translation of SML's declarations, and starts empty. As the declaration is deconstructed, record expressions, patterns, or types may be encountered. If there exists a record type in $\mathcal{R}_g$ or $\mathcal{R}_l$ such that the fields are the same, and their types are more general, then the translation proceeds as usual. If not, the type is stored in $\mathcal{R}_l$ using a fresh name. Once the declaration translation is done, the types from $\mathcal{R}_l$ are translated into `Record`s occurring before the declaration. The content of $\mathcal{R}_l$ is then added to $\mathcal{R}_g$.

```
type r = { name : string, age : int }

fun isBob ({name = "Bob",...}: r) = true
  | isBob {...} = false
```

```
Record rid1 := { rid1_name : string; rid1_age : Z }.
Definition r := rid1.

Equations isBob (x1: r): bool :=
  isBob {| rid1_age := _; rid1_name := "Bob" |} := true;
  isBob {| rid1_age := _; rid1_name := _ |} := false.
```

Figure 1: Translation for records

To avoid name clashing, we modify the record fields' names by prefixing it with the fresh name used for the record type. Each time a record type, expression, or pattern is found, either its type is found in the record context, or a new type is created. In both situations we are able to tell what this prefix is, and rename the fields accordingly.

Ellipsis on record patterns are resolved by looking into the annotations after SML's elaboration phase. Since the record type must be able to be inferred, this information can be extracted after elaboration, and the pattern can be unfolded with all fields.

The result of translating a record type and a function on this type is shown in Figure 1.

3.3 Polymorphic Types

The treatments of polymorphic values in SML and Coq are different. For example, in SML `val L = []` declares an empty list `L` of type `'a list`, where `'a` is a type variable. This value can be safely used with instantiated lists: `L = [3]` is a well-typed boolean expression (which evaluates to `false`).

In contrast, a "polymorphic" empty list can be declared in Coq in (at least) two different ways:

```
Definition L1 := @nil.
Definition L2 {A : Type} := [] : list A.
```

The types of the terms `L1` and `L2` *as is* (i.e. without annotations) are sligthly different:

```
L1 : forall A : Type, list A
L2 : list ?A where ?A : [ |- Type]
```

The definition of `L1` looks more similar to what is written in SML. However, if we want to use `L1` in other terms with instantiated lists (such as `L = [3]`), then we need to write: `(L1 _) = [3]`[5]. To avoid adding the type parameter explicitly, we can use `L2`, which is implicitly interpreted as `L2 _` by Coq. Indeed, the (type-)check `L2 = [3]` succeeds.

As a result, type variables are made explicit when translating polymorphic SML value declarations so that these values can be used *as is* in the rest of the program, like the example of `L2`. This is done using a type variable context $\mathcal{T}$. The type variable context is always empty at the beginning of a declaration's translation and, as the declaration is traversed, "unknown" types are added to the context. An unknown type becomes "known" when it is added to $\mathcal{T}$. That is, when the translator encounters an expression `e`

[5]Here `_` represents the type variable `A` whose type is inferred by Coq.

with unknown type α, it adds α to $\mathcal{T}$. If a later expression `e'` has type α, the translator treats this as a known type, not changing the type variable context. At the end of the translation, α is added as an implicit argument of the resulting Coq definition, and the translation of the expression `e` is annotated with the type α. For example, `val L = []` is translated to:

```
Definition L {_'13405 : Type} := ([] : @list _'13405).
```

where `_'13405` is the name of the type variable determined by HaMLet.

Note that this is only needed for value declarations. Functions on polymorphic types do not need explicit type parameters since they can be automatically generalized using `Generalizable All Variables`.

3.4 Non-exhaustive Matches

A very common practice when programming in SML is to use patterns for values to deconstruct expressions. For example: `val x::l = [1,2,3]` would result on value `x` being bound to `1` and `l` bound to `[2,3]`. SML's interpreter will issue a `Warning: binding not exhaustive`, but accepts the code. The warning makes sense since it is usually not possible to tell before runtime if the expression on the right will match the pattern (for example, when it is the result of a function application). Non-exhaustiveness is indicated by a flag in the declaration's annotation after elaboration.

Such declarations cannot be directly translated into Gallina because `Definition`s cannot be patterns, only identifiers. Patterns are accepted in `let ' pat := term` expressions, but `term` can *only* resolve to `pat` (i.e. its type has only one constructor). The translation of non-exhaustive declarations is made exhaustive by adding a default case resulting in a `patternFailure` axiom: (the same strategy is used in [9]): `Local Axiom patternFailure: forall {a}, a.` We should note here that the default case will never be reached in the translated code, as this would mean there was a `Bind` exception raised when evaluating the SML code. As mentioned before, the SML code is evaluated before starting the translation, and if it terminates abnormally (with an exception, for example), SMLtoCoq terminates too.

In addition to that, top level declarations need to be split into as many definitions as there are variables being bound. This is done by recursively traversing the pattern and collecting the variables. For each of them, a new Gallina definition is created. For example, `val x::l = [1,2,3]` becomes:

```
Definition x := match [1; 2; 3] with (x :: l) ⇒ x
                                   | _ ⇒ patternFailure
                end.
Definition l := match [1; 2; 3] with (x :: l) ⇒ l
                                   | _ ⇒ patternFailure
                end.
```

Note that the structure of the match term is the same, apart from the variable returned on the non-default case. If the declaration is inside a `let` block, it is translated into multiple nested `let` blocks.

3.5 Functions

Unless a function is total and structurally decreasing at every recursive call, it cannot be translated into `Fixpoint` (or `Definition`, in case it is not recursive) directly. Most of the problems we encountered in function translation could be solved using the Equations plugin, which provides a powerful tool for defining terminating functions via pattern-matching on dependent types. Equations turned out to be more flexible and easier to use than Coq's built-in `Program` command.

```
(!! posAdd(x, y) ==> b;
    REQUIRES: x > 0 andalso y > 0;
    ENSURES: b > x andalso b > y;    !!)
fun posAdd(x, y) = x + y;
```

```
Equations posAdd (x1: (Z * Z)%type): Z :=
  posAdd (x, y) := (x + y).

Theorem posAdd_THM: forall x y b, posAdd(x, y)=b ∧ ((x > 0) && (y > 0)) = true
                                  → ((b > x) && (b > y)) = true.
Admitted.
```

Figure 2: Translation for function contracts

3.5.1 Pattern matching

Programming in SML typically makes extensive use of pattern-matching, most common among which is pattern-matching on function inputs, for example:

```
fun length [] = 0
  | length (x :: l) = 1 + length l
```

Gallina, however, does not allow pattern-matching on function parameters which means that the above function would – in the best case – be translated to the following Coq code:

```
Fixpoint length {A : Type} ( id : list A ) :=
match id with
| [] ⇒ 0
| x :: l ⇒ 1 + length l
end.
```

While this looks acceptable, the translation is complicated as the number of (curried) parameters increases since `match` only deals with one term at a time. Equations allows the definition of functions by pattern matching on the arguments without the need for an intermediary `match` expression. This enables SMLtoCoq to produce code that looks much more similar to the corresponding SML code. For example, the `length` function defined above translates to the following in Coq with Equations:

```
Equations length `(x1: @list _'14188): Z :=
  length [] := 0;
  length (x :: l) := (1 + (length l)).
```

The main limitation associated with using Equations is that the function's input and output types have to be explicit. This does not pose much of a threat since we have type information from HaMLet's elaboration, and having them explicit does not affect the semantics of the code.

3.5.2 Contracts

Deductive verification is a common way to do formal verification, which consists of generating mathematical proof obligations from the code's specifications, then discharging these obligations using proof assistants such as Coq or automated theorem provers. Generating correct proof obligations is a crucial step in this process. To aid in this task, SMLtoCoq automatically translates function contracts (added to HaMLet as explained in Section 2.1) into Coq theorems. A contract of the form:

```
datatype 'a evenList = ENil
                     | ECons of 'a * 'a oddList
and 'a oddList = OCons of 'a * 'a evenList

fun lengthE (ENil: 'a evenList): int = 0
  | lengthE (ECons (_, l)) = lengthO l
and lengthO (OCons (_, l)) = lengthE l
```

```
Inductive evenList  {_a : Type} : Type :=
  | ENil
  | ECons : (_a * @oddList _a)%type → @evenList _a
with oddList  {_a : Type} : Type :=
  | OCons : (_a * @evenList _a)%type → @oddList _a.

Equations lengthE `(x1: @evenList _a): Z :=
  lengthE ENil := 0;
  lengthE (ECons (_, l)) := (lengthO l)
with lengthO `(x1: @oddList _a): Z :=
  lengthO (OCons (_, l)) := (lengthE l).
```

Figure 3: Translation for mutually recursive type and function

```
(!! f input ==> output;
    REQUIRES: precond;
    ENSURES:  postcond;  !!)
```

is translated into:

```
Theorem f_Theorem: forall vars, (f input = output ∧ precond = true)
                                → postcond = true.
```

The theorem's `precond` and `postcond` are the translations of SML boolean expressions `precond` and `postcond`, respectively. As such, they have type `bool` and not `Prop`, hence the need to use `= true` in the theorem statement. The quantified `vars` are the set of variables used in `input` and `output`. This theorem is placed after the function definition in the Gallina code followed by `Admitted`, and the proof is left to the user. An example of the resulting translation of a function with pre and post conditions is shown in Figure 2.

3.5.3 Mutual recursion

Figure 3 shows the translation of a mutually recursive type and function. SML's `and` construct maps nicely to Coq's `with`, which can also be used for `Equations`. One interesting thing to note about this example is how the polymorphic types `evenList` and `oddList` need to be annotated in Coq. First of all, they take a type variable as an implicit type due to the treatment of polymorphism explained in Section 3.3. As a result, when this type is used and a type variable is passed explicitly to it, it must be preceded by `@`. Another thing noticeable in the translation is the presence of `%type` annotating tuple types. Depending on the context, Coq cannot distinguish whether `*` is the tuple type or `nat` multiplication, so the annotation indicates to Coq that this is indeed a type.

3.5.4 Partial functions

One of the powerful features of SMLtoCoq is its ability to handle partial functions. While it is not possible to define partial functions in Coq as is, restricting the translation to total code would be a big loss, not only because partial functions are pervasive in programming, but also: (1) many times partial functions are defined with guarantees that the function will not be called on non-valid arguments and (2) Coq's rich type system accepts preconditions on functions as part of the function inputs. To address this issue, we exploit Coq's powerful support for dependent types to constrain the function domain. That is, we generate preconditions for partial functions and add them as function inputs, which Equations can then use to accept functions that handle the subset of inputs that satisfies the generated preconditions.

A typical example is the `head` function that returns the head of the list `fun hd (x::l) = x`, which is translated to:

```
Equations hd {A} (x1: list A) {H: ∃ y1 y2, x1 = y1::y2}: A :=
  hd (x :: l) := x;
  hd _ := _.
```

Note that the implicit parameter `H` ensures the function is only called on non-empty lists, and is thus total.

In simple cases, Equations can automatically derive a contradiction between the second case and the precondition and all obligations will be discharged. In more complicated cases, Equations would not be able to derive the contradiction, and an obligation remains for the user to solve[6].

The preconditions can be generated by simply requiring that the input parameters match one of the function's cases, for example: `x = case1 ∨ x = case2 ∨ ... ∨ x = casen`. But this can lead to unnecessarily complicated preconditions in the case of multiple input arguments and/or a function with multiple branches. Instead, we use the AST's knowledge of exhaustive patterns to generate preconditions that are considerably smaller than what would be produced by this naive procedure. Our algorithm works by identifying *generic* patterns, which match any input. For example, a single identifier or a constructor for a datatype with one constructor would be generic patterns. When preconditions are generated, generic patterns are eliminated because they are not imposing restrictions on the input. Consider this example:

```
fun hd_sum ((a,b)::l) ((a',b')::l') init = init + a + b + a' + b'
  | hd_sum ((a,b)::l) l'             init = init + a + b
  | hd_sum l          ((a',b')::l') init = init + a' + b'
```

The naive search would produce the following proposition:

```
(∃ a b l, x1 = (a, b)::l ∧ ∃ a' b' l', x2 = (a', b')::l' ∧ ∃ init, x3 = init) ∨
(∃ a b l, x1 = (a, b)::l ∧ ∃ l',       x2 = l'           ∧ ∃ init, x3 = init) ∨
(∃ l,     x1 = l         ∧ ∃ a' b' l', x2 = (a', b')::l' ∧ ∃ init, x3 = init)
```

Our procedure, however, produces:

```
(∃ y1 l,  x1 = y1 :: l ∧ ∃ y2 l', x2 = y2 :: l') ∨
(∃ y1 l,  x1 = y1 :: l) ∨
(∃ y2 l', x2 = y2 :: l')
```

Note that this can be further simplified to `(∃ y1 l, x1 = y1 :: l) ∨ (∃ y2 l', x2 = y2 :: l')`. This requires the implementation of a simplification algorithm for formulas, which we leave for future work. The actual translation of the `hd_sum` function is:

```
Equations hd_sum (x1: @list (Z * Z)%type) (x2: @list (Z * Z)%type)
  {H: exists y1 y2, eq (x1) (y1 :: y2) ∧ exists y1 y2, eq (x2) (y1 :: y2) ∨
      exists y1 y2, eq (x1) (y1 :: y2) ∨ exists y1 y2, eq (x2) (y1 :: y2)}: Z :=
```

[6] Coq accepts unsolved obligations at first, and a user can optionally "admit obligations" and resolve them when needed.

```
signature PAIR =
sig
  type t1
  type t2
  type t = t1 * t2
  val default : unit -> t
end

structure IntString : PAIR =
struct
  type t1 = int
  type t2 = string
  type t = t1 * t2
  fun default () = (0, "")
end

functor Example (Pair : PAIR) =
struct
  val (a, b) = Pair.default ()
end

structure S = Example (IntString)
```

```
Module Type PAIR.
  Parameter t1 : Type.
  Parameter t2 : Type.
  Definition t := (t1 * t2)%type.
  Parameter default : unit → t.
End PAIR.

Module IntString <: PAIR.
  Definition t1 := Z.
  Definition t2 := string.
  Definition t := (t1 * t2)%type.
  Equations default (x1: unit%type):
    (Z * string)%type :=
    default tt := (0, "").
End IntString.

Module Example ( Pair : PAIR ).
  Definition a :=
    match (Pair.default tt) with
    (a, b) ⇒ a end.
  Definition b :=
    match (Pair.default tt) with
    (a, b) ⇒ b end.
End Example.

Module S := !Example IntString.
```

Figure 4: Translation for the module system

```
hd_sum ((a, b) :: l) ((a', b') :: l') := (((a + b) + a') + b');
hd_sum ((a, b) :: l) l' := (a + b);
hd_sum l ((a', b') :: l') := (a' + b');
hd_sum _ _ := _.
```

3.6 Structures, Signatures, and Functors

An SML program can be divided into structures – with each structure comprising a collection of components, (i.e. datatypes, types, and values). A structure can ascribe to a signature, which acts like an interface. In addition to structures and signatures, SML also provides *functors*, which are essentially structures parametrizes by other structures [4]. Gallina has also a module system that provides similar mechanisms for structuring programs. While Gallina's modules and module types conveniently match SML's module language, there is one major syntactical limitation: inline structures and signatures.

Structure and signature expressions in SML can be either top-level or inline. Top-level structure and signature expressions in SML have a single format, where `S` is the component's name:

```
structure S   = structure_exp
signature S   = signature_exp
```

Inline structure expressions occur as functors' parameters:

```
structure S = functor F (structure_exp)
```

and inline signature expressions can occur in any of the following ways:

```
infix F
fun op F (x, y) = x*x + y
val f = op F
val x = 5 F 2
val y = op F (2, 3)
```

```
Equations F (x1: (Z * Z)%type): Z :=
  F (x, y) := ((x * x) + y).
Definition opF := F.
Notation "x 'F' y" := (F (x, y))
         (left associativity, at level 29).
Definition x := (5 F 2).
Definition y := (opF (2, 3)).
```

Figure 5: Translation for infix functions

```
structure_exp : signature_exp
structure_exp :> signature_exp
include signature_exp
structure S : signature_exp
functor F (structure S : signature_exp) = structure_exp
```

Note that inline structures and signatures are unnamed.

In Gallina, inline modules and module types are not allowed; they must be replaced by identifiers. For example, the following attempt of instantiating a `ListOrdered` module directly in the parameter of the `Dict` module: `Module D := Dict(ListOrdered(IntOrdered))` fails. Therefore, we distinguish between the translation of top-level expressions and inline expressions. The translation of inline expressions uses a structure/signature context Σ. This context is initially empty and gets populated with ASTs for inline signature and structure declarations as they are discovered in (possibly nested) inline expressions. Each declaration in this context is assigned a new name, and this name is used for instantiating the Coq module. An example of how modules are translated is shown in Figure 4.

3.7 Infix Functions

SML allows the declaration of infix functions via the `infix` and `fun op` constructs. HaMLet keeps track of infix functions in the *infix environment*, which is returned after parsing, together with the AST. As a result, the `infix` declaration is not part of the AST. When an infix function is used as a prefix, it is preceded by `op`, and this information is available in the AST. Infix functions can be declared in Coq using `Notation` or `Infix`. For our translation we need to use `Notation` because `Infix` assumes the function to be curried, while in SML infix functions are always uncurried.

Once a function declaration of the shape `fun op f` is found in SML's AST, SMLtoCoq checks if `f` in the infix environment, in which case it was declared as an infix function[7]. If `f` is infix, it creates two additional Gallina sentences to be placed after the function definition. First, it defines `opf` as `f` to be used when `op f` is used in the SML code. Secondly, it defines the `Notation "x 'f' y"` so that `f` can be used in infix form from this point onward. See an example in Figure 5. Similar to records, SMLtoCoq uses a context to keep track of which functions have infix notations in the Gallina AST.

3.8 Typed patterns

SML supports types in patterns at any level. The same does not hold for Gallina. For example:

```
Definition f := fun '((x, y) : nat * nat) ⇒ x + y.
Fail Definition f := fun '((x: nat, y: nat)) ⇒ x + y.
Fail Definition f x := match x with | x : nat ⇒ 1 | _ ⇒ 0 end.
```

[7]Note that SML allows non-infix functions to be declared using `fun op`.

Typing the pattern in the first definition is accepted, however, types "inside" the pattern as in the second definition are rejected. Also, typing patterns in `match` expressions is not accepted at all.

If we want to retain the same explicit types as in the SML code, it is possible to extract types nested in patterns to the top level. However, this may include other types that were not explicit. Also, we would need to identify precisely when top-level patterns are not supported, and find an alternative to make the types explicit. For `match` expressions, for example, one can type the expression being matched as opposed to the pattern. Due to these conflicts, our design choice was to ignore types in patterns. Most of the times this is not problematic as Coq is able to infer the correct types. Moreover, function types are already explicit since `Equations` requires it.

4 Related Work

Closest to our approach are hs-to-coq [9], which translates Haskell code into Coq specifications, and coq-of-ocaml[8], which translates OCaml code into Coq specifications. Even if these projects look (superficially) the same, an important difference is that our main goal is to lower the entry barrier into interactive theorem proving (in particular, Coq) for those that are familiar with functional programming (in particular, in SML). As such, we aim for a translation that looks as similar as possible to the SML code, which is obtained using the Equations plugin. We note that Equations is not used in hs-to-coq or coq-of-ocaml, and getting similarly looking code does not seem to be a priority in those projects, which are more focused on the formal verification of large codebases. Another distinguishing feature is the implementation of contracts for functions, which is not available in Haskell or OCaml.

As mentioned, SMLtoCoq uses the Equations plugin, while both hs-to-coq and coq-of-ocaml translate functions to `Definition`s or `Fixpoint`s. As such, they look quite different (and less elegant) than their Haskell or OCaml counterparts. For example, definitions of mutually recursive functions in hs-to-coq require the bodies of the functions to be repeated for each definition, a problem that is solved using mutually defined `Equations` (Section 3.5.3). We find that the Equations plugin helps in obtaining more aesthetically pleasing functions. Haskell functions are defined via cases, like in SML, but this is not supported in Gallina (as explained in Section 3.5.1). Therefore, hs-to-coq must create intermediate names for the arguments to be used in `match` expressions. SMLtoCoq avoids this, again via the Equations plugin. When it comes to partial functions, both hs-to-coq and coq-of-ocaml add a default case on `match` expressions, and use an `Axiom` as its return value. In contrast, SMLtoCoq generates the domain restriction as a dependent type to be used in the `Equations` and avoids the need for a (yet another) `Axiom` (see Section 3.5.4). We find this is a particularly elegant solution, as it reduces the amount of inconsistent axioms that need to be used.

Other smaller differences include the treatment of built-in types and records. While hs-to-coq translate types such as `Int` into their own implementation `GHC.Types.Int`, SMLtoCoq tries to leverage Coq's types as much as possible, translating `int` into Coq's `Z`. Records in hs-to-coq are translated into `Inductive` types with associated projection functions. SMLtoCoq uses Coq's built-in `Records`, having no need to declare projection functions explicitly (see Section 3.2). The treatment of these constructs in coq-of-ocaml is similar to ours. It is worth noting that coq-of-ocaml curries all constructors, while SMLtoCoq and hs-to-coq retain the user defined datatypes as faithful as possible to the original definition.

CFML [1] is a tool for verifying Caml programs based on the so-called *characteristic formulae*. These formulas are derived automatically from the program (without the need for annotations), and describe its behaviour. The resulting formula can be proved using Coq. CFML's newest version uses

[8] `https://clarus.github.io/coq-of-ocaml/`

separation logic to reason about OCaml code [2]. Differently from SMLtoCoq, CFML "translates" functions into specification lemmas in Coq, in the style of Hoare triples.

The Why tool [3] encompasses WhyML, and ML-like language with support for annotations, and the verifier Why3. Why3 leverages several automated and interactive theorem provers to discharge proof obligations coming from WhyML code as automatically as possible. Even if the language resembles our use of contracts in SML, the goal is not the same. SMLtoCoq uses solely Coq for verification, and does not aim at automation.

SMLtoCoq translates SML programs into Coq specifications for reasoning. Going in the other direction, Coq has an extraction mechanism which exports specifications to OCaml, Haskell, or Scheme. Similarly, F* [10] is an OCaml-like functional language which allows the programmer to state and prove lemmas about the code. After verification, programs can be extracted in OCaml, F#, C, WASM, or ASM. To use those tools the programmer needs to have expertise with proof assistants in advance. SMLtoCoq assumes a greater fluency in programming and less in theorem proving, thus enabling the user to write programs in their comfort zone, and later experiment with proving properties in a proof assistant. We believe this direction helps beginners in Coq understanding how program specifications would look like.

5 Conclusion & Future Work

We have described SMLtoCoq, a tool for automatically generating Coq specifications from SML programs. To the best of our knowledge, this is the first tool of its kind. SMLtoCoq is able to handle a considerable fragment of SML, including constructs that are not trivially translated into Gallina, such as partial functions, structures, and records. Additionally, we have implemented contracts for functions and their translation into Coq theorems. We have also ported a big part of SML's basis library into Coq, so that the code can be translated with the minimum amount of modifications. Using the resulting translation the user is able to prove properties about their code using Coq.

We plan to improve and extend SMLtoCoq in several ways.

Simplification of automatically generated pre-conditions As mentioned in Section 3.5.4, the automatically generated preconditions for functions can be further simplified by using logical equivalences. Since the result is always a proposition in disjunctive normal form, we plan to improve our procedure by converting that to conjunctive normal form and applying SAT heuristics for simplifying propositions.

Functions inside let blocks One of the drawbacks of using the Equations library is that `Equations` is a top-level declaration. In SML, functions can be declared nested inside let blocks, for example:

```
fun f x = let fun g y = y + 1 in g x end
val n = let fun h x = x + 1 in h 7 end
```

The definitions of `f` and `g` could be declared mutually using Equations' `where` construct. However, if the nesting depth is bigger than 2, the translation would flatten the structure, which could turn out to be problematic if some functions have the same name. The definition of `h`, on the other hand, cannot be translated into `Equations`. In this case, we need to define a new translation using Coq's native `let (fix)`.

Non-trivial recursion One of the fundamental differences between SML and Coq is that Coq only accepts terminating functions. Among those, functions that are non-trivially terminating must be accompanied by a proof of termination to be accepted. At the moment, SMLtoCoq translates all functions correctly, but those that need termination proofs cannot be compiled in Coq.

For example, the `div_two` function:

```
Equations div_two (n : nat) : nat :=
div_two 0 := 0;
div_two 1 := 0;
div_two n := 1 + div_two (n / 2) .
```

is terminating, but not trivially since the recursive call is not on the predecessor of `n`, but on `n / 2`. To make Coq accept this function, we can annotate it with the well-founded inductive measure to use for the recursive calls. In this case it is the simple `lt` function:

```
Equations div_two (n : nat) : nat by wf n lt :=
div_two 0 := 0;
div_two 1 := 0;
div_two n := 1 + div_two (n / 2) .
```

This generates the proof obligation $n/2 < n$ for $n \geq 2$ that needs to be proved by the user.

Using the `by` annotation, we can also have Coq accept non-terminating functions, but at the cost of an inconsistent axiom and admitted obligations.

```
Local Axiom indMeasure: forall {a}, a → nat.

Equations loop (x1: Z): Z by wf (indMeasure x1) _ :=
  loop x := (loop1 ((x + 1))).
Admit Obligations.
```

We are investigating ways to determine termination information for SML functions, ideally analogous to Coq's termination check. It is unlikely we can automatically figure out the correct inductive measure to use in the annotation (if there is any), but we can at least have functions that compile, and leave it up to the user to remove the use of inconsistent axioms and prove the obligations when possible.

Side-effects Two language features that we have largely (and reasonably) ignored were exceptions and reference cells. These operations involve side effects and, naturally, have no easy correspondence in Coq. Fortunately, dealing with side-effects in pure functional settings is a well-studied problem [7, 11, 5] and there are different solutions, including Coq libraries, that we could adapt to translate effectful SML code.

Correctness We would like to formally prove that our translation for SML into Gallina is correct, which would ultimately guarantee that the reasoning about the code in Coq translates to its SML source. To that end, we started to formalize our translation as a derivation system. Our goal is to show a simulation theorem between the SML source and its translation, using both languages' evaluation semantics. We will start with a small, purely functional, core of both languages, and extend from there.

References

[1] Arthur Charguéraud (2010): *Program Verification through Characteristic Formulae*. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, Association for Computing Machinery, New York, NY, USA, p. 321–332, doi:10.1145/1863543.1863590.

[2] Arthur Charguéraud (2020): *Separation logic for sequential programs (functional pearl)*. *Proc. ACM Program. Lang.* 4(ICFP), pp. 116:1–116:34, doi:10.1145/3408998.

[3] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 — Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: *Proceedings of the 22nd European Symposium on Programming*, Lecture Notes in Computer Science, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[4] Robert Harper (2011): *Programming in Standard ML*. Licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0.

[5] Thomas Letan & Yann Régis-Gianas (2020): *FreeSpec: Specifying, Verifying, and Executing Impure Computations in Coq*. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, Association for Computing Machinery, p. 32–46, doi:10.1145/3372885.3373812.

[6] Robin Milner, Mads Tofte & David Macqueen (1997): *The Definition of Standard ML*. MIT Press, doi:10.7551/mitpress/2319.001.0001.

[7] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau & Lars Birkedal (2008): *Ynot: Dependent Types for Imperative Programs*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, Association for Computing Machinery, p. 229–240, doi:10.1145/1411204.1411237.

[8] Matthieu Sozeau, Cyprien Mangin, Pierre-Marie Pédrot, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Robin Green, Maxime Dénès, Hugo Herbelin, Guillaume Claret, Siddharth, Enrico Tassi, Anton Trunov, Joachim Breitner, Antonio Nikishaev, SimonBoulier, Søren Nørbæk, Vincent Laporte & Yves Bertot (2020): *mattam82/Coq-Equations: Equations 1.2.3 for Coq 8.11*. doi:10.5281/zenodo.3967149.

[9] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah & Stephanie Weirich (2018): *Total Haskell is reasonable Coq*. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2018*, doi:10.1145/3167092.

[10] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué & Santiago Zanella-Béguelin (2016): *Dependent Types and Multi-Monadic Effects in F**. In: *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, pp. 256–270, doi:10.1145/2837614.2837655.

[11] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce & Steve Zdancewic (2020): *Interaction trees: representing recursive and impure programs in Coq*. *Proceedings of the ACM on Programming Languages* 4(POPL), p. 1–32, doi:10.1145/3371119.

Systematic Translation of Formalizations of Type Theory from Intrinsic to Extrinsic Style

Florian Rabe Navid Roux
University Erlangen-Nuremberg
Department of Computer Science
University Erlangen-Nuremberg
Erlangen, Germany
florian.rabe@fau.de navid.roux@fau.de

Type theories can be formalized using the intrinsically (hard) or the extrinsically (soft) typed style. In large libraries of type theoretical features, often both styles are present, which can lead to code duplication and integration issues.

We define an operator that systematically translates a hard-typed into the corresponding soft-typed formulation. Even though this translation is known in principle, a number of subtleties make it more difficult than naively expected. Importantly, our translation preserves modularity, i.e., it maps structured sets of hard-typed features to correspondingly structured soft-typed ones.

We implement our operator in the MMT system and apply it to a library of type-theoretical features.

1 Introduction

Motivation and Related Work Soft type theory goes back to Curry's work [4], where typing is a meta-language (ML) predicate $\texttt{of} : \texttt{term} \to \texttt{tp} \to \texttt{prop}$ between object-language (OL) terms and types. This is also called extrinsic typing. This leads to a deep embedding of typing where the OL judgment $t : A$ corresponds to the existence of a typing proof, i.e., an ML term witnessing $\texttt{of}\, t\, A$.

Hard type theory goes back to Church's work [2], where typing is a function from terms to types. If that function is sufficiently simple, formalizations in a dependently-typed ML like LF [6] may be able to capture it directly in the framework's type system. Concretely, such a representation uses a shallow embedding of typing where an object-language typing judgment $t : A$ corresponds to the meta-language (ML) typing judgment $t : \texttt{tm}\, A$. Only well-formed terms can be encoded at, and the OL type of a term can be read off of its ML type. This is also called intrinsic typing. If ML type-checking is decidable, that means OL must be as well.

Soft typing is more expressive and flexible than hard typing. But

- introducing an OL variable x of type a requires two ML variables of $x : \texttt{term}, x^* : \texttt{of}\, x\, a$,
- correspondingly, substituting a term for a variable requires the term and a proof of its typing judgment,
- type-checking is reduced to ML non-emptiness-checking, which is usually undecidable.

In the LATIN project [3] going back to ideas developed in the Logosphere project [11], we built a highly modular library of formalizations of logics and type theories. The goal was to create a library of Little Logics (in the style of Little Theories [5]), each formalizing one language feature such as product types, which can be combined to form concrete systems. This allows the reuse, translation, and combination of formalizations in the style of [7, 8]. It helps the meta-theoretical analysis as each modular

E. Pimentel and E. Tassi (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 88–103, doi:10.4204/EPTCS.337.7

construction is itself a meta-theorem, e.g., reusing the formalization of a language feature implies that two languages share that feature, and translations between languages can allow moving theorems across formal systems [9, 10].

Due to the incomparable advantages of soft and hard encodings, we had to formalize each feature in both styles. This led not only to a duplication of code but also caused significant maintenance problems. In particular, it is difficult to ensure coherent encoding styles (e.g., naming conventions, order of arguments, notations, etc.) in such a way that the two sets of encodings are related systematically.

For every pair of a hard- and a soft-typed encoding of the same language feature, there is a type erasure translation from the former to the latter with an associated type preservation property. Such translations have been investigated in various forms, see e.g., [1] for a systematic study in the form of realizability theories. We will cast the type preservation as a logical relation proof as formalized in an LF-based logical framework in [16]. Notably, given the hard-typed encoding, it is possible to derive the soft-typed one, the erasure translation, and the preservation proof automatically. We call this derivation *softening*. Systematic softening not only greatly reduces the encoding effort but simplifies maintenance and produces more elegant code.

Contribution and Overview We define an operator `Soften` in the logical framework LF [6]. Despite being conceptually straightforward, softening is a rather complex process, and an ad-hoc implementation, while possible, would be error-prone and hard to maintain. Therefore, we employ a systematic approach for deriving the softening operator that constructs the logical relation proof along with the softened theory. A particular subtlety was to ensure the generated code to still be human-readable. That required softening to consider pragmatic aspects like notations and choice of implicit arguments.

Our work is carried out under the LATIN2 header, which aims at a complete reimplementation of the LATIN library. While LATIN worked with modular Twelf [15], LATIN2 uses the MMT/LF incarnation of LF [13]. In addition to an implementation of LF and a module system, MMT provides a framework for diagram operators [14], which supports the meta-theory and implementation of operators that systematically derive formalizations from one another. It also makes it easy to annotate declarations, which we will use to guide the softening operator in a few places.

Importantly, these diagram operators are functorial in the category of LF theories and theory morphisms. That enables scaling them up to entire libraries in a way that preserves modularity. That is important to derive human-readable formalizations.

Sect. 2 introduces MMT/LF. Sect. 3 shows the key definition of the softening operator, and Sect. 4 establishes meta-theoretical properties that allow lifting it to libraries. Sect. 5 shortly sketches our implementation in the MMT system.

2 The MMT Framework and Basic Formalizations

MMT [13] is a framework for designing and implementing logical frameworks. To simplify, we only use the implementation of LF that comes with MMT's standard library, and restrict the grammar to the main features of MMT/LF: We assume the reader is familiar with LF (see e.g., [6]) and only recap the notions of theories and morphisms that MMT adds on top.

Δ	$::= \cdot$	diagrams
	$\mid \Delta, \texttt{theory}\, T = \{\Theta\}$	theory definition
	$\mid \Delta, \texttt{morph}\, m : S \to T = \{\vartheta\}$	morphism definition
Θ	$::= \cdot$	declarations in a theory
	$\mid \Theta, c : A[= t]$	typed, optionally defined constants
	$\mid \Theta, \texttt{include}\, S$	include of a theory
ϑ	$::= \cdot \mid \vartheta, c = t \mid \vartheta, \texttt{include}\, m$	declarations in a morphism
Γ	$::= \cdot \mid \Gamma, x : A$	contexts
t, A, f	$::= c \mid x \mid \texttt{type} \mid \texttt{kind} \mid \lambda_{x:A}\, t \mid \Pi_{x:A}\, B \mid f\, t$	LF expressions

Theories An MMT/LF theory is ultimately a list of **constant** declarations $c : A[= t]$ where the definiens t is optional. A constant declaration may refer to any previously declared constant. LF provides the primitives of a dependently typed λ-calculus, namely universes `type` and `kind`, function types $\Pi_{x:A}\, B$, abstraction $\lambda_{x:A}\, t$ and application $f\, t$. In a constant declaration $c : A$, we must have $A : \texttt{type}$ or $A : \texttt{kind}$, and in a variable binding $x : A$, we must have $A : \texttt{type}$. As usual, MMT/LF allows writing $A \to B$ for $\Pi_{x:A}\, B$ and omitting inferable brackets, arguments, and types. If we need to be precise about **typing**, we write $\Gamma \vdash_T t : A$ for the typing judgment between two expressions that may use all constants from theory T and all variables from context Γ.

A theory T may **include** a previously defined theory S, which makes all constants of S available in T as if they were declared in T.

Example 1. We give theories formalizing hard- and soft-typed type theories. The left shows the common theory `Proofs` that formalizes proofs in standard LF fashion using the judgments-as-types principle: $\texttt{ded}\, P$ is the type of proofs of the proposition $P : \texttt{prop}$, i.e., $\texttt{ded}\, P$ is non-empty iff p is provable. `HTyped` formalizes hard typing, also called intrinsic or Church typing, where typing is a function from terms to types, i.e., every term has a unique type that can be inferred from it. That enables the representation of object language terms $t : a$ as LF terms $t : \texttt{tm}\, a$. And `STyped` formalizes soft typing, also called extrinsic or Curry typing, where typing is a relation between terms and types, i.e., a term may have multiple or no types. That corresponds to a representation of an object language term $t : a$ in LF as an untyped term $t : \texttt{term}$ for which a proof of $\texttt{ded}\, \texttt{of}\, t\, a$ exists.

```
theory Proofs =
  prop : type
  ded  : prop → type
```

```
theory HTyped =
  include Proofs
  tp : type
  tm : tp → type
```

```
theory STyped =
  include Proofs
  tp   : type
  term : type
  of   : term → tp → prop
```

Morphisms A morphism $m : S \to T$ represents a compositional translation of all S-syntax to T-syntax. We spell out the definition and key property:

Definition 1. A morphism $m : S \to T$ is a mapping of S-constants to T-expressions such that for all S-constants $c : A$ we have $\vdash_T m(c) : \overline{m}(A)$ where $\overline{m}$ maps S-syntax to T-syntax as defined in Fig. 1. In the sequel, we write m for $\overline{m}$.

constants of S

$\overline{m}(c) = m(c)$

other expressions

$\overline{m}(x) = x$

$\overline{m}(\texttt{type}) = \texttt{type}$

$\overline{m}(\Pi_{x:A} B) = \Pi_{x:\overline{m}(A)} \overline{m}(B)$

$\overline{m}(\lambda_{x:A} t) = \lambda_{x:\overline{m}(A)} \overline{m}(t)$

$\overline{m}(f\,t) = \overline{m}(f)\,\overline{m}(t)$

contexts

$\overline{m}(\cdot) = \cdot$

$\overline{m}(\Gamma, x:A) = \overline{m}(\Gamma), x:\overline{m}(A)$

theories that include S

$\overline{m}(E = \{\ldots, D_i, \ldots\}) = E^m = \{\ldots, \overline{m}(D_i), \ldots\}$

$\overline{m}(\texttt{include}\, S) = \texttt{include}\, T$

$\overline{m}(c : A[= t]) = c : \overline{m}(A)[= \overline{m}(t)]$

$\overline{m}(\texttt{include}\, E) = \texttt{include}\, E^m$

constants of a theory including S

$\overline{m}(c) = c$

where E^m generates a fresh name for the translated theory

Figure 1: Map induced by a Morphism

Theorem 1. *For a morphism $m : S \to T$ and a theory E that includes S, if $\Gamma \vdash_E t : A$, then $m(\Gamma) \vdash_{E^m} m(t) : m(A)$. In particular for $E = S$, we have $m(\Gamma) \vdash_T m(t) : m(A)$.*

In terms of category theory, a morphism m induces a **pushout functor** $\mathcal{P}(m)$ from the category of theories including S to the category of theories including T. As a functor, m extends to diagrams, i.e., any diagram of theories E including S and morphisms between them is mapped to a corresponding diagram of theories E^m including T. Moreover, for each E, m extends to a morphism $E \to E^m$ that maps every S-constant according to m and every other constant to itself. Each of these morphisms maps E-contexts/expressions to E^m and that mapping preserves all judgments. These morphisms form a natural transformation, and we speak of a **natural functor**.

$\texttt{morph TE} : \texttt{HTyped} \to \texttt{STyped} =$
  $\texttt{include Proofs}$
  $\texttt{tp} = \texttt{tp}$
  $\texttt{tm} = \lambda_{a:\texttt{tp}}\, \texttt{term}$

$\texttt{theory HProd} =$
  $\texttt{include HTyped}$
  $\texttt{prod} : \texttt{tp} \to \texttt{tp} \to \texttt{tp}$
  $\texttt{pair} : \Pi_{a,b}\, \texttt{tm}\, a \to \texttt{tm}\, b \to \texttt{tm prod}\, a\, b$
  $\texttt{projL} : \Pi_{a,b}\, \texttt{tm prod}\, a\, b \to \texttt{tm}\, a$
  $\texttt{projR} : \Pi_{a,b}\, \texttt{tm prod}\, a\, b \to \texttt{tm}\, b$

$\texttt{theory HProd}^{\texttt{TE}} =$
  $\texttt{include STyped}$
  $\texttt{prod} : \texttt{tp} \to \texttt{tp} \to \texttt{tp}$
  $\texttt{pair} : \Pi_{a,b}\, \texttt{term} \to \texttt{term} \to \texttt{term}$
  $\texttt{projL} : \Pi_{a,b}\, \texttt{term} \to \texttt{term}$
  $\texttt{projR} : \Pi_{a,b}\, \texttt{term} \to \texttt{term}$

$\texttt{morph TE}_{\texttt{HProd}} : \texttt{HProd} \to \texttt{HProd}^{\texttt{TE}} =$
  $\texttt{include TE}$
  $\texttt{prod} = \texttt{prod}$
  $\texttt{pair} = \texttt{pair}$
  $\texttt{projL} = \texttt{projL}$
  $\texttt{projR} = \texttt{projR}$

Figure 2: Pushout along the Type Erasure Morphism

Example 2 (related to Fig. 2). The type erasure translation $\texttt{TE} : \texttt{HTyped} \to \texttt{STyped}$ maps types $a : \texttt{tp}$ to types $\texttt{TE}(a) : \texttt{tp}$, which we formalize by $\texttt{tp} = \texttt{tp}$. And it maps typed terms $t : \texttt{tm}\, a$ to untyped

terms $\mathtt{TE}(t) : \mathtt{term}$, which we formalize by $\mathtt{tm} = \lambda_{a:\mathtt{tp}}\,\mathtt{term}$ and thus $\mathtt{TE}(\mathtt{tm}\,a) = \mathtt{term}$. We also use `include Proofs` to include the identity morphism on `Proofs`, i.e., all constants of `Proofs` are mapped to themselves.

Applying this morphism, i.e., the pushout functor $\mathcal{P}(\mathtt{TE})$, to the theory `HProd` of hard-typed simple products yields the theory $\mathtt{HProd}^{\mathtt{TE}}$, which arises by replacing every occurrence of $\mathtt{tm}A$ with `term`. `TE` also extends to the morphism $\mathtt{TE}_{\mathtt{HProd}}$, which translates all expressions of `HProd` to expressions of $\mathtt{HProd}^{\mathtt{TE}}$. This translations preserves LF-typing, e.g., if $\vdash_{\mathtt{HTyped}} t : \mathtt{tm}\,\mathtt{prod}\,a\,b$, then $\vdash_{\mathtt{HTyped}^{\mathtt{TE}}} \mathtt{TE}_{\mathtt{HProd}}(t) : \mathtt{term}$.

However, $\mathtt{HProd}^{\mathtt{TE}}$ is not the desired formalization of soft-typed products (e.g., because it lacks constants relating types and terms), and we develop a more suitable functor in the next section.

3 The Softening Operator

3.1 Basic Overview

`Soften` translates theories based on `HTyped` to theories based on `STyped`. The key idea is that whenever we have an expression $t : \mathtt{tm}\,a$ in `HTyped`, then in `STyped` we need to synthesize two things: an expression $\mathtt{TE}(t) : \mathtt{term}$ and an expression $t^* : \mathtt{of}\,\mathtt{TE}(t)\,\mathtt{TE}(a)$ acting as a witness of type preservation. And whenever we have an expression $a : \mathtt{tp}$, we need to synthesize one thing only, namely $\mathtt{TE}(a) : \mathtt{tp}$. (Note that for simple types such as product and function types, we have $\mathtt{TE}(a) = a$. We discuss dependent function types in the next section.) Both intuitions extend homomorphically to all concepts of LF such as function types and contexts.

As an example, consider the constants $\mathtt{pair}, \mathtt{projL}, \mathtt{projR}$ in `HProd` in Fig. 3. For each of them we synthesize a type-erased constant of the same name and a starred typing witness in `STyped`. Note that the type parameters a and b are removed in their corresponding type-erased constant in `SProd`. We have been unable to find a systematic way to determine when arguments need to be removed and discuss this problem in Sec. 3.3. For the arguments in the starred constants such as $\mathtt{pair}^*$, we synthesize two parameters $x : \mathtt{term}$ and $x^* : \mathtt{ded}\,\mathtt{of}\,x\,a$ (whose name can often be omitted).

$$
\begin{array}{l}
\mathtt{theory}\ \mathtt{HProd} = \\
\quad \mathtt{include}\ \mathtt{HTyped} \\
\quad \mathtt{prod} \ : \mathtt{tp} \to \mathtt{tp} \to \mathtt{tp} \\
\quad \mathtt{pair} \ : \Pi_{a,b}\,\mathtt{tm}\,a \to \mathtt{tm}\,b \to \mathtt{tm}\,\mathtt{prod}\,a\,b \\
\quad \mathtt{projL} : \Pi_{a,b}\,\mathtt{tm}\,\mathtt{prod}\,a\,b \to \mathtt{tm}\,a \\
\quad \mathtt{projR} : \Pi_{a,b}\,\mathtt{tm}\,\mathtt{prod}\,a\,b \to \mathtt{tm}\,b
\end{array}
$$

$$
\begin{array}{l}
\mathtt{theory}\ \mathtt{SProd} = \\
\quad \mathtt{include}\ \mathtt{STyped} \\
\quad \mathtt{prod} \ : \mathtt{tp} \to \mathtt{tp} \to \mathtt{tp} \\
\quad \mathtt{pair} \ : \mathtt{term} \to \mathtt{term} \to \mathtt{term} \\
\quad \mathtt{pair}^* \ : \Pi_{a,b}\,\Pi_x\,\mathtt{ded}\,\mathtt{of}\,x\,a \to \Pi_y\,\mathtt{ded}\,\mathtt{of}\,y\,b \\
\qquad\qquad \to \mathtt{ded}\,\mathtt{of}\,(\mathtt{pair}\,x\,y)\,(\mathtt{prod}\,a\,b) \\
\quad \mathtt{projL} \ : \mathtt{term} \to \mathtt{term} \\
\quad \mathtt{projL}^* : \Pi_{a,b}\,\Pi_x\,\mathtt{ded}\,\mathtt{of}\,x\,(\mathtt{prod}\,a\,b) \\
\qquad\qquad \to \mathtt{ded}\,\mathtt{of}\,(\mathtt{projL}\,x)\,a \\
\quad \mathtt{projR} \ : \mathtt{term} \to \mathtt{term} \\
\quad \mathtt{projR}^* : \Pi_{a,b}\,\Pi_x\,\mathtt{ded}\,\mathtt{of}\,x\,(\mathtt{prod}\,a\,b) \\
\qquad\qquad \to \mathtt{ded}\,\mathtt{of}\,(\mathtt{projR}\,x)\,b
\end{array}
$$

Figure 3: Hard and Soft Product Types

3.2 Logical Relations

We capture the type preservation proof using logical relations. The meta-theory for using logical relations to represent type preservation was already sketched in [16], but we have to make a substantial generalization to *partial* logical relations and extend those to natural functors. Besides allowing the representation of partial translations, this has an important practical advantage: the translations in [16] must introduce unit argument types in places where no particular property about a term is proved. While semantically irrelevant, softening must remove these in order to produce the softened theories actually expected by humans, thus potentially violating the correctness of the translation. Partiality allows constructing the softened theories in a way that these artefacts are not introduced in the first place.

Because logical relations can be very difficult to wrap one's head around, we focus on the special case needed for softening although we have designed and implemented it for the much more general setting of [16]. Moreover, we advise readers to maintain the following intuitions while perusing the formal treatment below:

- The morphism $m : S \to T$ is the type erasure translation $\mathtt{TE} : \mathtt{HTyped} \to \mathtt{STyped}$.
- The logical relation r is a mapping $\mathtt{TP}$ from $\mathtt{HTyped}$-syntax to $\mathtt{STyped}$-syntax that maps
  - types $A : \mathtt{type}$ to unary predicates $\mathtt{TP}(A) : \mathtt{TE}(A) \to \mathtt{type}$ about $\mathtt{TE}$-translated terms of type A
  - terms $t : A : \mathtt{type}$ to proofs $\mathtt{TP}(t) : \mathtt{TP}(A)\,\mathtt{TE}(t)$ of the predicate associated with A
- Even more concretely,
  - $\mathtt{TP}(\mathtt{prop}), \mathtt{TP}(\mathtt{ded}), \mathtt{TP}(\mathtt{tp})$ are all undefined because we need not prove anything about terms at those types
  - $\mathtt{TP}(\mathtt{tm}) = \lambda_{a:\mathtt{tp}}\,\lambda_{x:\mathtt{term}}\,\mathtt{of}\,x\,a$ and thus $\mathtt{TP}(\mathtt{tm}\,a) = \lambda_{x:\mathtt{term}}\,\mathtt{of}\,x\,a$, i.e., $\mathtt{TP}$ maps every $t : \mathtt{tm}\,a$ to its typing proof $\mathtt{TP}(t) : \mathtt{of}\,\mathtt{TE}(t)\,a$.

Moreover, it may help readers to compare Def. 1 and 2 as well as Thm. 1 and 2.

Definition 2. A partial **logical relation** on a morphism $m : S \to T$ is a partial mapping r of S-constants to T-expressions such that for every S-constant $c : A$, if $r(c)$ is defined, then so is $\overline{r}(A)$ and $\vdash_T r(c) : \overline{r}(A)\,m(c)$. r is called **term-total** if it is defined for a typed constant if it is for the type. The partial mapping $\overline{r}$ of S-syntax to T-syntax is defined in Fig. 4. In the sequel, we write r for $\overline{r}$.

The key idea of the map $\overline{r}$ is to attempt to construct $\overline{r}$ in the same way as in [16] for total r. Whenever $\overline{r}$ is applied to an argument for which it is not defined, the expression is simply removed: if the type of a bound variable would be undefined, the whole binding is removed; if an argument of a function application would be undefined, the function is applied to one fewer argument. The next theorem states that these removals fit together in the sense that $\overline{r}$ still satisfies the main property of logical relations whenever it is defined:

Theorem 2. *For a partial logical relation r on a morphism $m : S \to T$, we have*

- *if $\Gamma \vdash_S t : A$ and r is defined for t, then r is defined for A and $r(\Gamma) \vdash_T r(t) : r(A)\,m(t)$*
- *if r is term-total, it is defined for a typed term if it is for its type*

Proof. The inductive definition is the same as in [16] except for the possibility of undefinedness. Thus, whenever the results are defined, the typing properties follow from the theorems there.

First, it is straightforward to see that r is total on contexts and substitutions because the case distinctions explicitly avoid recursing into arguments for which r is undefined.

Second, we show by induction on derivations of $\Gamma \vdash_S t : A$ that if $A : \mathtt{type}$ then r is defined for t iff it is defined for A.

- constant $c : A$: True by assumption.

$$
\begin{aligned}
\overline{r}(c) &= r(c) \\
\overline{r}(x) &= \begin{cases} x^* & \text{if } x^* \text{ was declared when traversing into the binder of } x \\ \text{undefined} & \text{otherwise} \end{cases} \\
\overline{r}(\texttt{type}) &= \lambda_{a:\texttt{type}}\, a \to \texttt{type} \\
\overline{r}(\Pi_{x:A} B) &= \lambda_{f:m(\Pi_{x:A} B)}\, \Pi_{\overline{r}(x:A)}\, \overline{r}(B)\,(f\,x) \\
\overline{r}(\lambda_{x:A}\, t) &= \lambda_{\overline{r}(x:A)}\, \overline{r}(t) \\
\overline{r}(f\,t) &= \begin{cases} \overline{r}(f)\, m(t)\, \overline{r}(t) & \text{if } \overline{r}(t) \text{ defined} \\ \overline{r}(f)\, m(t) & \text{otherwise} \end{cases} \\
\overline{r}(\cdot) &= \cdot \\
\overline{r}(\Gamma, x:A) &= \overline{r}(\Gamma), \begin{cases} x : m(A), x^* : \overline{r}(A)\, x & \text{if } \overline{r}(A) \text{ defined} \\ x : m(A) & \text{otherwise} \end{cases}
\end{aligned}
$$

$\overline{r}(-)$ is undefined whenever an expression on the right-hand side is.

Figure 4: Map induced by a Logical Relation

- variable $x : A$: The case for $\Gamma, x : A$ introduces the variable x^* into the target context if $r(A)$ is defined. The case for x picks up on that and (un)defines r at x accordingly.
- λ-abstraction $\lambda_{x:A}\, t : \Pi_{x:A} B$: r is always defined for $x : A$. By induction hypothesis, it is defined for t if it is for B.
- t cannot be a Π-abstraction
- application $f\,t : B(t)$ for some $f : \Pi_{x:A} B(x)$: By definition, r is defined for $f\,t$ if it is defined for f. By induction hypothesis the latter holds iff r is defined for $\Pi_{x:A} B(x)$, which by definition holds iff it is defined for $B(x)$. It remains to show that r is defined for $B(t)$ iff it is defined for $B(x)$ in the context extended with $x : A$. By induction hypothesis, r is defined for t iff it is defined for x. Therefore, and because the definition of r is compositional, substituting t for x cannot affect whether r is defined for an expression.

Finally, if $\Gamma \vdash_E t : A$ for $A : \texttt{kind}$, we need to show that r is defined for A if it is for t. That is trivial: inspecting the definition shows that r is always defined for kinds anyway. □

We now capture how to synthesize `SProd` from `HProd` via logical relations. First we define the type erasure morphism `TE` : `HTyped` $\to$ `STyped` and the type preservation property as a logical relation `TP` on `TE`. Then we define and apply the following functor on `HProd`, which can be thought of as an analog of pushout along a logical relation:

Definition 3. Consider a morphism $m : S \to T$ and a term-total logical relation r on m. Then the functor $\mathcal{LR}(m, r)$ from S-extensions to T-extensions maps theories E as follows:

1. We compute the pushout $E^m := \mathcal{P}(m)(E)$.
2. E^m has the same shape as E and there is a morphism $m_E : E \to E^m$. For each, we create an initially empty logical relation r_E on m_E.
3. For each declaration $c : A[= t]$ in E for which $r_E(A)$ is defined, we add
   (a) the constant declaration $c^* : r_E(A)\, m_E(c)[= r_E(t)]$ to E^m
   (b) the case $r(c) = c^*$ to r_E.

Concretely we get:

$$\begin{array}{l}
\texttt{theory}\ \mathcal{LR}(\texttt{TE},\texttt{TP})(\texttt{HProd}) = \\
\quad \texttt{include}\ \texttt{STyped} \\
\quad \texttt{prod} : \texttt{tp} \to \texttt{tp} \to \texttt{tp} \\
\quad \texttt{pair} : \Pi_{a,b}\, \texttt{term}\, a \to \texttt{term}\, b \to \texttt{term}\, (\texttt{prod}\, a\, b) \\
\quad \texttt{pair}^* : \Pi_{a,b}\, \Pi_x\, \texttt{ded of}\, x\, a \to \Pi_y\, \texttt{ded of}\, y\, b \\
\qquad \to \texttt{ded of}\, (\texttt{pair}\, a\, b\, x\, y)\, (\texttt{prod}\, a\, b) \\
\quad \texttt{projL} : \Pi_{a,b}\, \texttt{term}\, a \to \texttt{term}\, (\texttt{prod}\, a\, b) \\
\quad \texttt{projL}^* : \Pi_{a,b}\, \Pi_x\, \texttt{ded of}\, x\, (\texttt{prod}\, a\, b) \to \texttt{ded of}\, (\texttt{projL}\, a\, b\, x)\, a \\
\quad \texttt{projR} : \Pi_{a,b}\, \texttt{term}\, b \to \texttt{term}\, (\texttt{prod}\, a\, b) \\
\quad \texttt{projR}^* : \Pi_{a,b}\, \Pi_x\, \texttt{ded of}\, x\, (\texttt{prod}\, a\, b) \to \texttt{ded of}\, (\texttt{projR}\, a\, b\, x)\, b
\end{array}$$

Here we see that `pair`, `projL`, `projR` all take undesired (and unused) type arguments. In the sequel, we will suitably extend the naive definition given above.

Fig. 5 gives some additional examples of hard-typed features. Here we also include hard-typed equality `HEqual` to formulate the reduction rules for function types. Fig. 6 shows the corresponding soft-typed variants that we intend to obtain. Note that these examples already foreshadow that `Soften` can be extended to theories containing includes in a straightforward way. We will define that formally in Sect. 4.

3.3 Removal of Unnecessary Parameters

In Sect. 3.2 we developed a translation from `HTyped` to `STyped` that maps every constant $c : A$ to a translated constant $c : m(A)$ and a witness $c^* : r(A)\, c$, where we chose $m = \texttt{TE}$ to be our type erasure morphism and $r = \texttt{TP}$ our logical relation capturing type preservation. This translation almost produced the desired formalization `SProd` except that some translated constants featured undesired type parameters. Pre- or post-composing our translation with one that removes selected type parameters is straightforward and presented in the following. The major problem is identifying these parameters in the first place. For example, in the library in Fig. 5 above we can distinguish the following cases:

- removal required, e.g., $\texttt{pair} : \Pi_{a,b}\, \texttt{tm}\, a \to \texttt{tm}\, b \to \texttt{tm}\, \texttt{prod}\, a\, b$ should go to $\texttt{pair} : \texttt{term} \to \texttt{term} \to \texttt{term}$
- removal optional depending on the intended result, e.g., $\texttt{eq} : \Pi_a\, \texttt{tm}\, a \to \texttt{tm}\, a \to \texttt{prop}$ can go to $\texttt{eq} : \texttt{term} \to \texttt{term} \to \texttt{prop}$ or to $\texttt{eq} : \Pi_a\, \texttt{term} \to \texttt{term} \to \texttt{prop}$; analogously $\texttt{lam} : \Pi_{a,b}\, (\texttt{tm}\, a \to \texttt{tm}\, b) \to \texttt{tm}\, \texttt{fun}\, a\, b$ can go to $\texttt{lam} : (\texttt{term} \to \texttt{term}) \to \texttt{term}$ or to $\texttt{lam} : \Pi_a\, (\texttt{term} \to \texttt{term}) \to \texttt{term}$
- removal forbidden, e.g., $\texttt{dfun} : \Pi_{a:\texttt{tp}}\, (\texttt{tm}\, a \to \texttt{tp}) \to \texttt{tp}$ should go to $\texttt{dfun} : \Pi_a\, (\texttt{term} \to \texttt{tp}) \to \texttt{tp}$

Definition 4 (Unused Positions)**.** Consider a constant $c : A$ in a theory S in a diagram D. After suitably normalizing, A must start with a (possibly empty) sequence of n Π-bindings, and any definition of c (direct or morphism) must start with the same variable sequence λ-bound. We write $c^1, \ldots, c^n$ for these variable bindings. Each occurrence of c in an expression in D is (after suitably η-expanding if needed) applied to exactly n terms, and we also write c^i for those argument positions.

We call a set P of argument positions of D-constants **unused** if for every $c^i \in P$, the i-th bound variable of the type or any definition of c occurs at most as a subexpression of argument positions that are themselves in P.

We write $D \setminus P$ for the diagram that arises from P by removing for every $c^i \in P$

- the i-th variable binding in the type and all definitions of c, e.g., $c : \Pi_{x_1:A_1}\, \Pi_{x_2:A_2}\, B$ becomes $c : \Pi_{x_1:A_1}\, B$ if $i = 2$,

$\texttt{theory HEqual} =$
$\quad \texttt{include HTyped}$
$\quad \texttt{@keep(eq}^1\texttt{)}$
$\quad \texttt{eq} \quad : \Pi_a \, \texttt{tm}\, a \to \texttt{tm}\, a \to \texttt{prop}$
$\quad \texttt{refl} \; : \Pi_{a,x} \, \texttt{ded eq}\, a\, x\, x$
$\quad \texttt{eqsub} : \Pi_{a,x,y} \, \texttt{ded eq}\, a\, x\, y \to$
$\qquad \Pi_{F:\texttt{tm}\, a \to \texttt{prop}} \, \texttt{ded}\, F\, x \to \texttt{ded}\, F\, y$

$\texttt{theory HFun} =$
$\quad \texttt{include HEqual}$
$\quad \texttt{fun} : \texttt{tp} \to \texttt{tp} \to \texttt{tp}$
$\quad \texttt{@keep(lam}^1\texttt{)}$
$\quad \texttt{lam} : \Pi_{a,b} \, (\texttt{tm}\, a \to \texttt{tm}\, b) \to \texttt{tm fun}\, a\, b$
$\quad \texttt{app} : \Pi_{a,b} \, \texttt{tm fun}\, a\, b \to \texttt{tm}\, a \to \texttt{tm}\, b$

$\texttt{theory HBeta} =$
$\quad \texttt{include HFun}$
$\quad \texttt{beta} : \Pi_{a,b} \, \Pi_{F:\texttt{tm}\, a \to \texttt{tm}\, b} \, \Pi_x$
$\qquad \texttt{ded eq}\, b\, (\texttt{app}\, a\, b\, (\texttt{lam}\, a\, b\, F)\, x)\, (F\, x)$

$\texttt{theory HEta} =$
$\quad \texttt{include HFun}$
$\quad \texttt{eta} : \Pi_{a,b} \, \Pi_{f:\texttt{tm fun}\, a\, b}$
$\qquad \texttt{ded eq}\, (\texttt{fun}\, a\, b)\, f\, (\texttt{lam}\, a\, b\, \lambda_x \, \texttt{app}\, f\, x)$

$\texttt{theory HDepFun} =$
$\quad \texttt{include HEqual}$
$\quad \texttt{@keep(dfun}^1\texttt{)}$
$\quad \texttt{dfun} : \Pi_a \, (\texttt{tm}\, a \to \texttt{tp}) \to \texttt{tp}$
$\quad \texttt{@keep(dlam}^1\texttt{)}$
$\quad \texttt{dlam} : \Pi_a \, \Pi_{b:\texttt{tm}\, a \to \texttt{tp}} \, (\Pi_{x:\texttt{tm}\, a} \, \texttt{tm}\, b\, x)$
$\qquad \to \texttt{tm dfun}\, a\, b$
$\quad \texttt{dapp} : \Pi_{a,b} \, \texttt{tm dfun}\, a\, b \to \Pi_{x:\texttt{tm}\, a} \, \texttt{tm}\, b\, x$

$\texttt{theory HExten} =$
$\quad \texttt{include HFun}$
$\quad \texttt{exten} : \Pi_{a,b} \, \Pi_{f,g:\texttt{tm fun}\, a\, b}$
$\qquad (\Pi_x \, \texttt{ded eq}\, b\, (\texttt{app}\, a\, b\, f\, x)$
$\qquad (\texttt{app}\, a\, b\, g\, x)) \to \texttt{ded eq}\, (\texttt{fun}\, a\, b)\, f\, g$

$\texttt{theory HDepBeta} =$
$\quad \texttt{include HDepFun}$
$\quad \texttt{dbeta} : \Pi_{a,b} \, \Pi_{F:\Pi_{x:\texttt{tm}\, a} \, \texttt{tm}\, b\, x} \, \Pi_x$
$\qquad \texttt{ded eq}\, (b\, x)\, (\texttt{dapp}\, a\, b\, (\texttt{dlam}\, a\, b\, F)\, x)\, (F\, x)$

Figure 5: Theories for Function Types with Annotations for Needed Arguments

- the i-argument of any application of c, e.g., $c\, t_1\, t_2$ becomes $c\, t_1$ if $i = 2$.

Lemma 3 (Removing Unused Positions)**.** *Consider a well-typed diagram D and a set P of argument positions unused in D. Then $D \setminus P$ is also well-typed.*

Proof. Technically, this is proved by induction on the typing derivation of D. But it is easy to see: by construction, (i) the variables bindings in P do not occur in $D \setminus P$ so that all types and definitions stay well-typed, and (ii) the type, definitions, and uses of all constants are changed consistently so that they stay well-typed. The only subtlety is that we need to apply LF's η-equality to expand not fully applied uses of a constant. □

Note that, in the presence of include declarations or morphisms, the decision whether an argument position may be removed is not local: we must consider the entire diagram to check for all occurrences. If a theory T includes the theory S and uses a constant c declared in S, then an argument position c^i may be unused in S but used in T. Thus, the functor that removes argument positions may have to be undefined on T.

Implementing the operation $D \setminus P$ is straightforward. However, much to our surprise and frustration, automatically choosing an appropriate set P turned out to be difficult:

$\texttt{theory SEqual} =$
$\texttt{include STyped}$
$\texttt{eq} : \Pi_a \texttt{term} \to \texttt{term} \to \texttt{prop}$
$\texttt{refl}^* : \Pi_{a,x} \texttt{ded of} x a \to \texttt{ded eq} a x x$
$\texttt{eqsub}^* : \Pi_a \Pi_{x,x^*:\texttt{ded of} x a} \Pi_{y,y^*:\texttt{ded of} y a}$
$\texttt{ded eq} a x y \to$
$\Pi_{F:\texttt{term}\to\texttt{prop}} \texttt{ded} F x \to \texttt{ded} F y$

$\texttt{theory SFun} =$
$\texttt{include SEqual}$
$\texttt{fun} : \texttt{tp} \to \texttt{tp} \to \texttt{tp}$
$\texttt{lam} : \Pi_a (\texttt{term} \to \texttt{term}) \to \texttt{term}$
$\texttt{lam}^* : \Pi_{a,b} \Pi_{F:\texttt{term}\to\texttt{term}}$
$(\Pi_x \texttt{ded of} x a \to \texttt{ded of} (F x) b)$
$\to \texttt{ded of} (\texttt{lam} a F) (\texttt{fun} a b)$
$\texttt{app} : \texttt{term} \to \texttt{term} \to \texttt{term}$
$\texttt{app}^* : \Pi_{a,b} \Pi_f \texttt{ded of} f (\texttt{fun} a b) \to$
$\Pi_x \texttt{ded of} x a \to \texttt{ded of} (\texttt{app} f x) b$

$\texttt{theory SDepFun} =$
$\texttt{include SEqual}$
$\texttt{dfun} : \Pi_a (\texttt{term} \to \texttt{tp}) \to \texttt{tp}$
$\texttt{dlam} : \Pi_a (\texttt{term} \to \texttt{term}) \to \texttt{term}$
$\texttt{dlam}^* : \Pi_a \Pi_{b:\texttt{term}\to\texttt{tp}} \Pi_{F:\texttt{term}\to\texttt{term}}$
$(\Pi_x \texttt{ded of} x a \to \texttt{ded of} (F x) (b x))$
$\to \texttt{ded of} (\texttt{dlam} a b F) (\texttt{dfun} a b)$
$\texttt{dapp} : \texttt{term} \to \texttt{term} \to \texttt{term}$
$\texttt{dapp}^* : \Pi_{a,b} \Pi_f \texttt{ded of} f (\texttt{dfun} a b) \to$
$\Pi_x \texttt{ded of} x a \to \texttt{ded of} (\texttt{dapp} f x) (b x)$

$\texttt{theory SBeta} =$
$\texttt{include SFun}$
$\texttt{beta}^* : \Pi_{a,b} \Pi_{F:\texttt{term}\to\texttt{term}}$
$(\Pi_x \texttt{ded of} x a \to \texttt{ded of} (F x) b)$
$\to \Pi_x \texttt{ded of} x a \to$
$\texttt{ded eq} b (\texttt{app} (\texttt{lam} a F) x) (F x)$

$\texttt{theory SExten} =$
$\texttt{include SFun}$
$\texttt{exten}^* : \Pi_{a,b} \Pi_{f:\texttt{term}} \texttt{ded of} f (\texttt{fun} a b) \to$
$\Pi_{g:\texttt{term}} \texttt{ded of} g (\texttt{fun} a b) \to$
$(\Pi_x \texttt{ded of} x a \to \texttt{ded eq} b (\texttt{app} f x) (\texttt{app} g x))$
$\to \texttt{ded eq} (\texttt{fun} a b) f g$

$\texttt{theory SEta} =$
$\texttt{include SFun}$
$\texttt{eta}^* : \Pi_{a,b} \Pi_{f:\texttt{term}} \texttt{ded of} f (\texttt{fun} a b)$
$\texttt{ded eq} (\texttt{fun} a b) f (\texttt{lam} a \lambda_x \texttt{app} f x)$

$\texttt{theory SDepBeta} =$
$\texttt{include SDepFun}$
$\texttt{dbeta}^* : \Pi_{a,b} \Pi_{F:\texttt{term}\to\texttt{term}}$
$(\Pi_x \texttt{ded of} x a \to \texttt{ded of} (F x) (b x)) \to$
$\Pi_x \texttt{ded of} x a \to$
$\texttt{ded eq} (b x) (\texttt{dapp} (\texttt{dlam} a F) x) (F x)$

Figure 6: Result of Softening the Theories from Fig. 5

Example 3. The undesired argument positions in $\texttt{TE}^{\texttt{HProd}}$ are exactly the named variables in HProd that do not occur in their scopes in $\texttt{TE}^{\texttt{HProd}}$ anymore. This includes the positions $\texttt{pair}^1$ and $\texttt{pair}^2$, and removing them yields the desired declaration of pair in SProd.

However, that does not hold for HDepFun. Here the argument $\texttt{dfun}^1$ is named in HDepFun and unused in the declaration $\texttt{dfun} : \Pi_{a:\texttt{tp}} (\texttt{term} \to \texttt{tp}) \to \texttt{tp}$ that occurs in $\texttt{TE}^{\texttt{HDepFun}}$. However, that is in fact the desired formalization of the soft-typed dependent function type. Removing $\texttt{dfun}^1$ would yield the undesired $\texttt{dfun} : (\texttt{term} \to \texttt{tp}) \to \texttt{tp}$. While we do not mention MMT's implicit arguments in this paper, note also that $\texttt{dfun}^1$ is an *implicit* argument in HDepFun that must become *explicit* in SDepFun.

This is trickier than it sounds because some argument positions may only be removable if they are

removed at the same time; so a fixpoint iteration might be necessary. Moreover, picking the largest possible P is entirely wrong as it would remove all argument positions. At the very least, we should only remove *named* argument positions, i.e., those that are bound by a named variable (as opposed to the anonymous variables introduced by parsing e.g., $\texttt{prod} : \texttt{tp} \to \texttt{tp} \to \texttt{tp}$). A smarter choice is to remove all named argument positions that become redundant through pushout, e.g., that are named and used in $\texttt{HProd}$ but unused in $\texttt{TE}(\texttt{HProd})$. (Note that the pushout $\mathcal{P}(m)(D)$ has at least the argument positions that D has. It may have more if m maps an atomic type to a function type.) That is the right choice almost all the time but not always.

After several failed attempts, we have been unable to find a good heuristic for choosing P. For now, we remove all named variables that never occur in their scope anymore, and we allow users to annotate constants like $\texttt{@keep}(\texttt{dfun}^1)$ where the system should deviate from that heuristic (see Fig. 5). We anticipate finding better solutions after collecting more data in the future. In the sequel, we write $\mathcal{P}^-(m)(D) := \mathcal{P}(m)(D) \setminus P_D$ where P_D is any fixed heuristic. $\texttt{HProd}^{\mathcal{P}^-(\texttt{TE})}$ yields the theory $\texttt{SProd}$ except that it still lacks the *-ed constants. The following lemma shows that we can now obtain the morphism $e : \texttt{HProd} \to \texttt{SProd}$ from above as $\mathcal{P}^-(\texttt{TE})_{\texttt{HProd}}$:

Lemma 4 (Removing Arguments Preserves Naturality)**.** *Consider a natural functor O and a functor $O'(D) := O(D) \setminus P_D$ for some heuristic P. Then O' is natural as well.*

Proof. O being natural yields morphisms $O_E : E \to E^O$ from D-theories to $O(D)$ theories. $O(D')$ has the same shape as $O(D)$, and to show that $O'(D)$ is natural, we reuse essentially the same morphisms from D-theories to $O'(D)$-theories. We only have to η-expand the right-hand sides of all assignments in the morphisms O_E and remove the same argument positions in P_D as well. □

It is straightforward to extend Def. 2 to all theories extending S in the same way as pushout extends a morphism. That would yield an include- and definition-preserving natural functor. However, we omit that here because that functor would work with $\mathcal{P}(m)$ whereas we want to use $\mathcal{P}^-(m)$. Instead, we make a small adjustment similar how we obtained $\mathcal{P}^-(m)$ from $\mathcal{P}(m)$:

Definition 5. Consider a morphism $m : S \to T$ and a term-total logical relation r on m. Then the functor $\mathcal{LR}(m,r)$ maps a theory E as follows:

1. We compute $E^m = \mathcal{P}^-(m)(E)$.
2. Due to Lem. 4, E^m has the same shape as E, and there is a morphism $m_E : E \to E^m$. We create an initially empty logical relation r_E on m_E.
3. For each E-declaration $c : A[= t]$ for which $r_E(A)$ is defined, we add
   (a) the constant declaration $c^* : r_E(A)\, c[= r_E(t)]$ to E^m
   (b) the case $r(c) = c^*$ to r_E.

Theorem 5. *In the situation of Def. 5, the operator $\mathcal{LR}(m,r)$ is a natural functor. And every r_E is a term-total logical relation on m_E.*

Proof. We already know that $\mathcal{P}^-(-)$ has the desired properties. Moreover, adding well-typed declarations to $\mathcal{P}^-(m)$ does not affect the naturality (because adding declaration to the codomain never affects the well-typedness of a morphism). So for the first claim, we only have to prove that our additions are well-typed.

We prove that and the fact that r_E is a logical relation jointly by induction on the derivation of the well-typedness of D: We appeal to Thm. 2 to show that the added constant declarations are well-typed. And the cases $r(c) = c^*$ satisfy the typing requirements of logical relations by construction. □

Now the functor $\mathcal{LR}(\texttt{TE},\texttt{TP})$ generates for every hard-typed feature F
- the corresponding soft-typed feature F'
- the type-erasure translation $\texttt{TE}_F : F \to F'$ as a compositional/homomorphic mapping,
- the type preservation proof $\texttt{TP}_F$ for the type erasure as a logical relation on $\texttt{TE}_F$.

In particular, we have $\texttt{SProd} = \mathcal{LR}(\texttt{TE},\texttt{TP})(\texttt{HProd})$.

3.4 Translating Proof Rules Correctly

We omitted the reduction rules in our introductory example HProd. This was because $\mathcal{LR}(\texttt{TE},\texttt{TP})$ is still not the right operator. To see what goes wrong, assume we leave $\texttt{TP}(\texttt{ded})$ undefined, and consider the type of the beta rule from HBeta:

HBeta	$\Pi_{a,b}\,\Pi_{F:\texttt{tm}\,a\to\texttt{tm}\,b}\,\Pi_x\,\texttt{ded}\,\texttt{eq}\,b\,(\texttt{app}\,a\,b\,(\texttt{lam}\,a\,b\,F)\,x)\,(F\,x)$
$\texttt{HBeta}^{\mathcal{LR}(\texttt{TE},\texttt{TP})}$ (generated)	$\Pi_{a,b}\,\Pi_{F:\texttt{term}\to\texttt{term}}\,\Pi_x$ $\texttt{ded}\,\texttt{eq}\,b\,(\texttt{app}\,(\texttt{lam}\,a\,F)\,x)\,(F\,x)$
SBeta (needed)	$\Pi_{a,b}\,\Pi_{F:\texttt{term}\to\texttt{term}}\,\Pi_{F^*:\Pi_a\,\texttt{ded of}\,x\,a\to\texttt{ded of}\,(F\,x)\,b}\,\Pi_x\,\Pi_{x^*:\texttt{ded of}\,x\,a}$ $\texttt{ded}\,\texttt{eq}\,b\,(\texttt{app}\,(\texttt{lam}\,a\,F)\,x)\,(F\,x)$

The rule generated by $\mathcal{LR}(\texttt{TE},\texttt{TP})(\texttt{HProd})$ is well-typed but not sound. In general, the softening operator must insert *-ed assumptions for all variables akin to how Def. 5 inserts them for constants. But it must only do so for proof rules and not for, e.g., fun, lam, and app.

We can achieve that by generalizing to partial logical relations on *partial* morphisms. Intuitively, we define $\mathcal{PLR}(m,r)$ for partial m and r in the same way as $\mathcal{LR}(m,r)$, again dropping all variable and constant declarations for whose type the translation is partial.

First we refine TE and TP (from Fig. 2 and Page 93, respectively) as follows:
- We leave $\texttt{TE}(\texttt{ded})$ undefined, i.e., our morphisms do not translate proofs. That is to be expected because we know that TE cannot be extended to a morphism that also translates proofs [12].
- We put $\texttt{TP}(\texttt{ded}) = \lambda_{p:\texttt{prop}}\,\texttt{ded}\,p$ and thus $\texttt{TP}(\texttt{ded}\,P) = \texttt{ded}\,\texttt{TE}(P)$ for all P. This trick has the effect that $\texttt{beta}^*$ is generated as well and has the needed type (whereas the generation of beta is suppressed).

Then we finally define $\texttt{Soften} = \mathcal{PLR}(\texttt{TE},\texttt{TP})$. For every proof rule c over HTyped, it
- drops the declaration of c,
- generates the declaration of c^*, which now has the needed type.

Soften is still include- and definition-preserving but is no longer natural. We conjecture that it is lax-natural and captures proof translations as lax morphisms in the sense of [12].

4 Translating Libraries

In the examples so far we have applied Soften on theories extending HTyped one at a time. We now extend it to a translation on whole structured diagrams of theories and morphisms, mapping whole libraries of hard-typed features at once.

Before spelling out the definition, we show an exemplary library of hard-typed features in Fig. 7. Here, we extend the collection of theories shown so far, the most notable extensions being several morphisms and the theory HQuot formalizing hard-typed quotient types. Here, the morphism $\texttt{HProd} \to \texttt{HDepProd}$ realizes simple product types as a special case of dependent product types. Analogously, all of the morphisms $\texttt{H}\{\texttt{Fun},\texttt{Beta},\texttt{Eta},\texttt{Exten}\} \to \texttt{HDep}\{\texttt{Fun},\texttt{Beta},\texttt{Eta},\texttt{Exten}\}$ realize the simply-typed

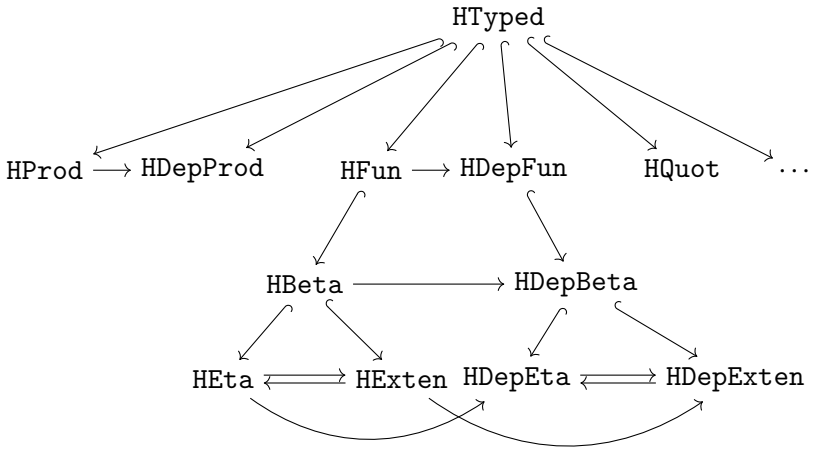


Figure 7: Diagram of hard-typed features

feature as a special case of the corresponding dependently-typed feature. And the anti-parallel morphism pairs $\texttt{HEta} \rightleftharpoons \texttt{HExten}$ and $\texttt{HDepEta} \rightleftharpoons \texttt{HDepExten}$ capture that η and extensionality are equivalent in the presence of β reduction. Our goal is to soften this library in a way that preserves the modular structure.

Definition 6 (`Soften` on Diagrams)**.** On the category of theories and partial morphisms, we define `Soften` as the partial functor translating diagrams D over `HTyped` to diagrams D' over `STyped` as follows:

- every theory/morphism with name X in D yields a theory/morphism $X^{\texttt{Soften}}$ in D'
- every `include` `HTyped` is replaced by `include` `STyped`; and every `include` X by `include` $X^{\texttt{Soften}}$
- every declaration $c : A\,[= t]$ in a theory S yields those on the left below, and every assignment $c := t$ in a morphism yields those on the right below (whenever the involved translations are defined)

$$\begin{array}{lll} c & : m_S(A)\,[= m_S(t)] & \qquad c \quad := m_S(t) \\ c^* & : \texttt{TP}^S(A)\,c\,[= \texttt{TP}^S(t)] & \qquad c^* := \texttt{TP}^S(t) \end{array}$$

where $m_S : S \to \mathcal{P}^-(\texttt{TE}^S)(S)$ is the morphism from Lem. 4 and $\texttt{TE}^S$ and $\texttt{TP}^S$ are given below.

We define `TE` as the partial morphism and `TP` as the partial logical relation on `TE` by

$$\begin{array}{ll} \texttt{TE}(\texttt{prop}) & = \texttt{prop} \\ \texttt{TE}(\texttt{tp}) & = \texttt{tp} \\ \texttt{TE}(\texttt{tm}) & = \lambda_{a:\texttt{tp}}\,\texttt{term} \\ \texttt{TP}(\texttt{ded}) & = \lambda_{p:\texttt{prop}}\,\texttt{ded}\,p \\ \texttt{TP}(\texttt{tm}) & = \lambda_{a:\texttt{tp}}\,\lambda_{x:\texttt{term}}\,\texttt{of}\,x\,a \end{array}$$

Then we additionally build the following components of D':

- every theory X yields a partial morphism $\texttt{TE}^T : T \to T^{\texttt{Soften}}$ and a partial logical relation $\texttt{TP}^T$ over $\texttt{TE}^T$

- every `include HTyped` in a theory S is replaced by `include TE` in $\texttt{TE}^S$, and $\texttt{TP}^S$ is made to extend TP; and every other `include` T in theories S yields `include` $\texttt{TE}^S$, and the definition of $\texttt{TP}^S$ extends $\texttt{TP}^T$
- every declaration $c : A\,[= t]$ in a theory S yields $c := c$ in $\texttt{TE}^S$ and $\texttt{TP}^S(c) := c^*$ (whichever are defined)

Theorem 6 (Structure Preservation)**.** *Consider the category of LF theories and partial type- and substitution-preserving expression translations as morphisms. Then* `Soften` *is functorial and preserves the structure of includes and definitions. It is natural via the morphisms* $\texttt{TE}^S$ *and the relation* $\texttt{TP}^S$.

Proof. This holds by construction. □

This finally yields the intended soft-typed formulation of $\texttt{SFun} := \texttt{HFun}^{\texttt{Soften}}$ and $\texttt{SDepFun} := \texttt{HDepFun}^{\texttt{Soften}}$. As an example, we give the morphism $\texttt{HSFtoDF} : \texttt{HFun} \to \texttt{HDepFun}$ and its translation below.

$\texttt{morph}\ \texttt{HSFtoDF} : \texttt{HFun} \to \texttt{HDepFun} =$
$\quad \texttt{include}\ \texttt{HEqual}$
$\quad \texttt{fun} = \lambda_{a,b}\, \texttt{dfun}\, a\, \lambda_x\, b$
$\quad \texttt{lam} = \lambda_{a,b,F}\, \texttt{dlam}\, a\, (\lambda_x\, b)\, F$
$\quad \texttt{app} = \lambda_{a,b,f,x}\, \texttt{dapp}\, a\, (\lambda_x\, b)\, f\, x$

$\texttt{morph}\ \texttt{HSFtoDF}^{\texttt{Soften}} : \texttt{HFun}^{\texttt{Soften}} \to \texttt{HDepFun}^{\texttt{Soften}} =$
$\quad \texttt{include}\ \texttt{HEqual}^{\texttt{Soften}}$
$\quad \texttt{fun} = \lambda_{a,b}\, \texttt{dfun}\, a\, \lambda_x\, b$
$\quad \texttt{lam} = \lambda_{a,F}\, \texttt{dlam}\, a\, F$
$\quad \texttt{lam}^* = \lambda_{a,b,F,F^*}\, \texttt{dlam}^*\, a\, (\lambda_x\, b)\, F\, F^*$
$\quad \texttt{app} = \lambda_{f,x}\, \texttt{dapp}\, f\, x$
$\quad \texttt{app}^* = \lambda_{a,b,f,f^*,x,x^*}\, \texttt{dapp}\, a\, (\lambda_x\, b)\, f\, f^*\, x\, x^*$

If we generalize the meta-theory of [12] to partial morphisms/relations and work in a variant of LF that adds product types, we could pair up $\texttt{TE}^S$ and $\texttt{TP}^S$ into a single expression translation that maps every term to the pair of its type erasure and its type preservation proof.

5 Implementation

The formalizations developed in this paper including the act of softening are available online as part of the LATIN2 library.[1] Our implementation adds a component to MMT that applies logical relation-based translations to entire diagrams of theories and morphisms. Then softening arises as one special case of that construction. While all translations are implemented in the underlying programming language of MMT and thus part of the trusted code base, our general and systematic approach minimizes the amount of new code needed for any given instance such as softening. That makes it much easier to review their correctness. In any case, the generated diagrams can easily be double-checked by the original logical framework.

We are still experimenting with how to trigger these translations. It is non-obvious if softening should be triggered by a new kind of declaration in the logical framework, a library-level script that lives outsides the logical framework, or a feature of the implementation that transparently builds the softened theory whenever the user refers to it. As a prototype, we have chosen the first of these approaches.

In our implementation it proved advantageous to not have separate syntax for morphisms and logical relations. Indeed, both are maps of names to expressions that extend to compositional translations of expressions to expressions, differing only in the inductive extension. Instead, we found a way to represent

[1] https://gl.mathhub.info/MMT/LATIN2/-/tree/devel/source/casestudies/2021-softening

every logical relation as a morphism, thus obviating the need to introduce additional syntax for relations. The trick is to implement a special include-preserving functor that generates a theory I that specifies exactly the typing requirements for the cases in a logical relation r, and then to represent r as a morphism out of I. We can even use this trick to represent multiple translations at once in a single morphism. For example, in our implementation we jointly represent `TE` and `TP` from Def. 6 as a single morphism `TypePres`, which in implementation-near syntax reads as follows.

```
view TypePres : HTyped_comptrans -> STyped =
  prop/TE = prop
  tp/TE   = tp
  tm/TE   = [x] term

  tm/TP   = [A,t] ⊢ t : A
  ded/TP  = ded
```

Here, `HTyped_comptrans` refers to a suitable kind interface theory for combined translations on `HTyped`.

Softening now emerges as the composition of multiple operators in our implementation, which for the sake of conciseness were combined in one big operator in this paper. Assume we wanted to soften a library of hard-typed features given as a diagram `HLibrary` (e.g., the one from Fig. 7). First, we compute the diagram `HLibrary_comptrans` of corresponding interface theories. This diagram has the same shape as `HLibrary` and is a tree rooted in `HTyped_comptrans`. Then, we compute the pushout of the resulting diagram along `TypePres`. The steps so far are effectively equivalent to applying the operator from Def. 3 accounted for with correct translation of proof rules. Finally, it remains to apply the operator that drops unnecessary parameters (according to the heuristic outlined in Sec. 3.3). Below, we show how the last two steps look like in our.

```
diagram SLibrary :=
  DROP_PARAMS STyped (PUSHOUT TypePres HLibrary_comptrans)
```

6 Conclusion

We have given a translation of hard-typed (intrinsic) to soft-typed (extrinsic) formalizations of type theory. Even though the existence of such translation is known, it had previously proved difficult to derive it from meta-theoretic principles in such a way that it can be studied and implemented easily. Our key insight was that the associated type preservation proof can be cast as a logical relation, which allowed us to derive the translation from the requirement that the logical relation proof succeeds.

Our translation preserves modularity, which makes it suitable for translating modular libraries of formalizations of various type theories. That enhances the quality and coverage of the library while reducing the maintenance effort. Our implementation will serve as a key component in scaling up our modular formalizations of type theories in our LATIN2 library.

We expect our methodology of functors on diagrams of LF theories to extend to other important translations such as adding polymorphism or universes.

References

[1] J. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In M. Hofmann, editor, *Foundations of Software Science and Computational Structures*, pages 108–122, 2011. doi:10.1007/978-3-642-19805-2_8.

[2] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940. doi:10.2307/2266170.

[3] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011. doi:10.1007/978-3-642-22673-1_24.

[4] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958. doi:10.1016/S0049-237X(08)72041-X.

[5] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992. doi:10.1007/3-540-55602-8_192.

[6] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

[7] R. Harper, D. Sannella, and A. Tarlecki. Structured theory presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994. doi:10.1016/0168-0072(94)90009-4.

[8] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program*, 67(1–2):114–145, 2006. doi:10.1016/j.jlap.2005.09.005.

[9] P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In R. Boulton and P. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001. doi:10.1007/3-540-44755-5_23.

[10] F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000. doi:10.1006/inco.1999.2832.

[11] F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. https://kwarc.info/projects/logosphere/.

[12] F. Rabe. Lax Theory Morphisms. *ACM Transactions on Computational Logic*, 17(1), 2015. doi:10.1145/2818644.

[13] F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017. doi:10.1093/logcom/exu079.

[14] F. Rabe and N. Roux. Structure-Preserving Diagram Operators. In M. Roggenbach, editor, *Recent Trends in Algebraic Development Techniques*, pages 142–163. Springer, 2020. doi:10.1007/978-3-030-73785-6_8.

[15] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009. doi:10.1145/1577824.1577831.

[16] F. Rabe and K. Sojakova. Logical Relations for a Logical Framework. *ACM Transactions on Computational Logic*, 14(4):1–34, 2013. doi:10.1145/2536740.2536741.

Adelfa: A System for Reasoning about LF Specifications

Mary Southern Gopalan Nadathur
University of Minnesota
Minneapolis Minnesota, USA
south163@umn.edu ngopalan@umn.edu

We present a system called Adelfa that provides mechanized support for reasoning about specifications developed in the Edinburgh Logical Framework or LF. Underlying Adelfa is a new logic named $\mathscr{L}_{LF}$. Typing judgements in LF are represented by atomic formulas in $\mathscr{L}_{LF}$ and quantification is permitted over contexts and terms that appear in such formulas. Contexts, which constitute type assignments to uniquely named variables that are modelled using the technical device of *nominal constants*, are characterized in $\mathscr{L}_{LF}$ by *context schemas* that describe their inductive structure. We present these formulas and an associated semantics before sketching a proof system for constructing arguments that are sound with respect to the semantics. We then outline the realization of this proof system in Adelfa and illustrate its use through a few example proof developments. We conclude the paper by relating Adelfa to existing systems for reasoning about LF specifications.

1 Introduction

This paper describes a proof assistant called Adelfa that supports reasoning about specifications written in the Edinburgh Logical Framework, or LF [8]. Adelfa is based on a logic called $\mathscr{L}_{LF}$ whose atomic formulas represent typing judgements in LF, and quantification is permitted over both contexts and terms which appear in these judgements. Term quantification is qualified by simple types that identify the functional structure of terms while forgetting dependencies. Context quantification is similarly qualified by schemas that capture the way contexts evolve during typing derivations in LF. The logic is complemented by a sequent-calculus based proof system that supports the construction of arguments of validity for relevant formulas. In addition to interpreting the logical symbols, the rules of the proof system embody an understanding of LF derivability; particular rules encode a case analysis style reasoning on LF judgements, the ability to reason inductively on the heights of such derivations, and an understanding of LF metatheorems. Adelfa is a tactics style theorem prover that implements this proof system: the development of a proof proceeds by invoking one of a set of sound tactic commands towards partially solving one of the proof obligations comprising the current proof state.

The rest of the paper is structured as follows. In Section 2 we briefly describe the underlying logic. Section 3 then outlines the associated proof system. In Section 4 we describe Adelfa and illustrate its use through a few examples. Section 5 concludes the paper by contrasting our work with other approaches that have been explored for reasoning about LF specifications.

2 A Logic for Articulating Properties of LF Specifications

We limit ourselves here to a brief overview of $\mathscr{L}_{LF}$, leaving its detailed presentation to other work [13, 19]. As already noted, the atomic formulas in $\mathscr{L}_{LF}$ represent typing judgements in LF, a calculus with which we assume the reader to be already familiar. We begin by summarizing aspects of LF that are pertinent to discussions in this paper. We then present the formulas of $\mathscr{L}_{LF}$ and illustrate their use in stating properties about LF derivability.

E. Pimentel and E. Tassi (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2021)
EPTCS 337, 2021, pp. 104–120, doi:10.4204/EPTCS.337.8

2.1 LF Basics and Hereditary Substitution

The variant of LF that is used in $\mathscr{L}_{LF}$ is that presented in [9]. The only expressions permitted in this version, which is referred to as *canonical LF*, are those that are in β-normal form. Moreover, the typing rules ensure that all well-formed expressions are also in η-long form.

$$
\begin{array}{ll}
ty : Type & tm : Type \\
unit : ty & app : tm \to tm \to tm \\
arr : ty \to ty \to ty & abs : ty \to (tm \to tm) \to tm
\end{array}
$$

$$of : tm \to ty \to Type$$

$$of\_app : \Pi M_1{:}tm.\, \Pi M_2{:}tm.\, \Pi T_1{:}ty.\, \Pi T_2{:}ty.\; of\ M_1\ (arr\ T_1\ T_2) \to of\ M_2\ T_1 \to of\ (app\ M_1\ M_2)\ T_2$$

$$of\_abs : \Pi T_1{:}ty.\, \Pi T_2{:}ty.\, \Pi R{:}tm \to tm.\; (\Pi x{:}tm.\, of\ x\ T_1 \to of\ (R\ x)\ T_2) \to of\ (abs\ T_1\ (\lambda x.\, R\ x))\ (arr\ T_1\ T_2)$$

Figure 1: An LF signature for typing in the simply typed λ-calculus

Object systems are encoded in LF by describing a signature which provides a means for representing the relevant constructs in the system as well as the relations between them. For example, if our focus is on typing judgements concerning the simply typed λ-calculus (STLC), then we might use the signature shown in Figure 1. This specification begins by identifying the LF types *ty* and *tm* and constructors for these types that serve to build LF representations of STLC types and terms; note that object language abstractions are represented using LF abstractions, following the idea of higher-order abstract syntax. Dependencies in types are then exploited to encode the typing relation between STLC terms and types. Specifically, the signature defines the type-level constant *of* towards this end: if *T* and *Ty* are LF terms of type *tm* and *ty*, respectively, then $(of\ T\ Ty)$ is intended to denote that the STLC expressions represented by *T* and *Ty* stand in this relation. The term-level constants *of_app* and *of_abs* that are identified as constructors for this type serve to encode the usual typing rules for applications and abstractions. In the use of such a specification, a central focus is on LF typing judgements of the form $\Gamma \vdash_\Sigma M \Leftarrow A$ where Σ represents a well-formed signature, *A* represents a well-formed type and Γ represents a well-formed context that assigns types to variables that may appear in *M* and *A*. Such a judgement is an assertion that *M* is typeable at the type *A* relative to the signature Σ and a context Γ that assigns particular LF types to the variables that appear in *M* and *A*. Such judgements are meant to translate into meaningful statements about the object systems encoded by the signature. For example, if Σ is the signature presented in Figure 1 and *A* is a type of the form $(of\ T\ Ty)$, then the judgement corresponds to the assertion that the STLC term represented by *T* is typeable at the type represented by *Ty* and that *M* encodes an STLC derivation for this judgement.

The typing rules in LF require the consideration of substitutions into LF expressions. In canonical LF, substitution must build in normalization, a requirement that is realized in [9] via the operation of *hereditary substitution*. We have generalized this operation to permit multiple simultaneous substitutions.Formally, a substitution is given by a finite set of triples of the form $\{\langle x_1, M_1, \alpha_1\rangle, \ldots, \langle x_n, M_n, \alpha_n\rangle\}$ in which, for $1 \leq i \leq n$, x_i is a variable, M_i is a term and α_i is a simple type constructed from the sole base type *o*. We refer to the simple types that index substitutions as *arity types*. The attempt to apply such a substitution to an LF expression always terminates. Moreover, it must yield a unique result whenever the expression being substituted into and the substituents satisfy the functional structure determined by

the arity types. We do not discuss this constraint in more detail here, noting only that the constraint is satisfied in all the uses we make of hereditary substitution in this paper. We write $E[\![\theta]\!]$ to denote the result of applying the substitution θ to the expression E in this situation.

2.2 Formulas over LF Typing Judgements and their Meaning

The logic $\mathscr{L}_{LF}$ is parameterized by an LF signature Σ. The basic building blocks for formulas in this context are typing judgements of the form $\Gamma \vdash_\Sigma M \Leftarrow A$ that are written as $\{G \vdash M : A\}$. However, the syntax of the expressions in the logic differs somewhat from that of the LF expressions. To begin with, we permit term variables that are bound by quantifiers to appear in types and terms. These variables have a different logical character from the variables that are bound in a context. More specifically, these variables may be instantiated by terms in the domains of the quantifiers, whereas the variables bound by declarations in an LF context represent fixed entities that are also distinct from all other similar entities within the typing judgement. To accurately capture the role of the variables bound in an LF context, we represent them using *nominal constants* [7, 21]; these are entities that are represented in this paper by the symbol n possibly with subscripts and that behave like constants except that they can be permuted with other such entities in an atomic formula without changing logical content. To support this treatment, we also allow nominal constants to appear in expressions corresponding to types and terms in the logic. Finally, we allow for contexts to be variables so as to permit quantification over them. More specifically the syntax for G, which constitutes a *context expression* in $\{G \vdash M : A\}$, is given by the following rule:

$$G \quad ::= \quad \Gamma \mid \cdot \mid G, n : A.$$

The symbol Γ here denotes the category of variables that range over contexts.

In typical reasoning scenarios, instantiations for context variables must be constrained in order to be able to articulate contentful properties. Such constraints are described via a special kind of typing for context variables. The formal realization of this idea, which is inspired by the notion of *regular worlds* in Twelf [16, 18], is based on declarations given by the following syntax rules.

$$\begin{array}{rrcl} \textbf{Block Declarations} & \Delta & ::= & \cdot \mid \Delta, y : A \\ \textbf{Block Schema} & \mathscr{B} & ::= & \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta \\ \textbf{Context Schema} & \mathscr{C} & ::= & \cdot \mid \mathscr{C}, \mathscr{B} \end{array}$$

According to this definition, a *context schema* is a collection of block schemas. The instances of a block schema $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}(y_1 : B_1, \ldots, y_m : B_m)$ are sequences $n_1 : C_1, \ldots, n_m : C_m$ obtained by choosing particular terms for the schematic variables $x_1, \ldots, x_n$ and particular nominal constants for the $y_1, \ldots, y_m$. A context satisfies a context schema if the context comprises a sequence of instances for the block schemas defining it. Context schemas are the types that are associated with context variables and they play the obvious role in limiting their domains.

The formulas of $\mathscr{L}_{LF}$ are given by the following syntax rule:

$$F ::= \{G \vdash M : A\} \mid \forall x : \alpha.F \mid \exists x : \alpha.F \mid \Pi\Gamma : \mathscr{C}.F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \supset F_2 \mid \top \mid \bot$$

The symbol Π represents universal quantification pertaining to contexts; note that such quantification is qualified by a context schema. The symbol x represents a term variable, i.e. the logic permits universal and existential quantification over LF terms. The symbol α that annotates such variables represents an arity type. For a formula to be well-formed, the terms appearing in it must be well-typed and in canonical form relative to an arity typing that is determined as follows: for the quantified variables this is given by

their annotations; and for a constant in the signature or a nominal constant in the context expression that is assigned the LF type A it is the *erased form* of A, written as $(A)^-$, that is obtained by replacing the atomic types in A by o and otherwise retaining the functional structure. This well-typing requirement is intended to eliminate structural issues in the consideration of validity of a formula, thereby allowing the focus to be on the more contentful relational aspects that are manifest in dependencies in typing.

A closed formula, i.e. a formula in which no unbound context or term variable appears, that is of the form $\{G \vdash M : A\}$ is deemed to be true exactly when G is a well-formed LF context, A is a well formed LF type relative to G and the typing judgement $G \vdash_\Sigma M \Leftarrow A$ is derivable. This semantics is extended to closed formulas involving the logical connectives and constants by using their usual understanding. Finally, the quantifiers are accorded a substitution semantics. The formula $\Pi\Gamma : \mathscr{C}.F$ holds just in the case that $F[\{G/\Gamma\}]$ holds for every context G that satisfies the context schema $\mathscr{C}$, where $E[\sigma]$ denotes the result of a standard replacement substitution applied to the expression E, being careful, of course, to carry out any renaming of bound variables that is necessary to avoid inadvertent capture. The formula $\forall x : \alpha.F$ holds exactly when $F[\![\{\langle x, t, \alpha\rangle\}]\!]$ holds for every closed term t that has the arity type α. Finally, the formula $\exists x : \alpha.F$ is true if there is some closed term t with arity type α such that $F[\![\{\langle x, t, \alpha\rangle\}]\!]$ is true.

2.3 An Illustration of the Logic

The property of uniqueness of type assignment for the STLC can be expressed via the formula

$$\Pi\Gamma : c.\forall E : o.\forall T_1 : o.\forall T_2 : o.\forall D_1 : o.\forall D_2 : o.\{\Gamma \vdash T_1 : ty\} \supset \{\Gamma \vdash T_2 : ty\} \supset \\ \{\Gamma \vdash D_1 : of\ E\ T_1\} \supset \{\Gamma \vdash D_2 : of\ E\ T_2\} \supset \exists D_3 : o.\{\cdot \vdash D_3 : eq\ T_1\ T_2\}$$

where c represents the context schema $\{T : o\}(x : tm, y : of\ x\ T)$ and the signature parameterizing the logic is that in Figure 1 augmented with the declarations $eq : ty \to ty \to Type$ and $refl : \Pi T : ty.eq\ T\ T$. A special case of this formula is that when the context variable is instantiated with the empty context. However, the typing rule for abstractions will require us to consider non-empty contexts in the analysis. Note, though, that the signature ensures that the extended contexts that need to be considered will always satisfy the constraint expressed by c.

We can argue for the validity of the formula above in two steps. We show first the validity of

$$\Pi\Gamma : c.\forall T_1 : o.\forall T_2 : o.\forall D : o.\{\Gamma \vdash D : eq\ T_1\ T_2\} \supset \{\cdot \vdash D : eq\ T_1\ T_2\}.$$

This formula, which is essentially a *strengthening* property for eq, can be argued to be valid by observing that bindings in a well-formed LF context satisfying the context schema c have no role to play in constructing a term of the (well-formed) LF type $(eq\ T_1\ T_2)$ for any terms T_1 and T_2. We then combine this observation with the validity of the formula

$$\Pi\Gamma : c.\forall E : o.\forall T_1 : o.\forall T_2 : o.\forall D_1 : o.\forall D_2 : o.\{\Gamma \vdash T_1 : ty\} \supset \{\Gamma \vdash T_2 : ty\} \supset \\ \{\Gamma \vdash D_1 : of\ E\ T_1\} \supset \{\Gamma \vdash D_2 : of\ E\ T_2\} \supset \exists D_3 : o.\{\Gamma \vdash D_3 : eq\ T_1\ T_2\}.$$

To establish the validity of the last formula, we must show that, for a closed context expression G that instantiates the schema c and for closed expressions E, T_1, T_2, D_1, and D_2, if $\{G \vdash D_1 : of\ E\ T_1\}$ and $\{G \vdash D_2 : of\ E\ T_2\}$ are both valid, then T_1 and T_2 must be identical. The argument proceeds by induction on the height of the derivation of $G \vdash_\Sigma D_1 \Leftarrow of\ E\ T_1$ that the assumption implies must exist. An analysis of this derivation shows that there are three cases to consider, corresponding to whether the head symbol of D_1 is *of_app*, *of_abs*, or a nominal constant from G. For the last of these, the argument is that the validity of $\{G \vdash D_1 : of\ E\ T_1\}$ implies that G is a well-formed context and thus that the typing is unique. The other two cases invoke the induction hypothesis; in the case of *of_abs* we must consider a shorter derivation in which the context has been extended, but in a way that conforms to the definition of the context schema c.

3 A Proof System for the Logic

We now describe a sequent calculus that supports arguments of validity of the kind outlined in Section 2.3. We aim only to present the spirit of this calculus, leaving its detailed consideration again to other work [13, 19]. We note also that the main goal for the calculus is to provide a means for sound and effective reasoning rather than to be complete with respect to the semantics described for $\mathscr{L}_{LF}$. We begin by explaining the structure of sequents before proceeding to a discussion of the inference rules.

3.1 The Structure of Sequents

A sequent, written as $\mathbb{N};\Psi;\Xi;\Omega \longrightarrow F$, is a judgement that relates a finite subset $\mathbb{N}$ of nominal constants with associated arity types, a finite set Ψ of term variables also with associated arity types, a finite set Ξ of context variables with types of the kind described below, a finite set Ω of *assumption formulas* and a *conclusion* or *goal* formula F; here, $\mathbb{N}$ is the *support set* of the sequent, Ψ is its *eigenvariables context* and Ξ is its *context variables context*. The formulas in $\Omega \cup \{F\}$ must be formed out of the symbols in $\mathbb{N}$, Ψ, Ξ and the (implicit) signature Σ, and they must be well-formed with respect to these collections in the sense explained in Section 2.2. The members of Ξ have the form $\Gamma \uparrow \mathbb{N}_\Gamma : \mathscr{C}[G_1, \ldots, G_n]$, where $\mathbb{N}_\Gamma$ is a collection of nominal constants, $\mathscr{C}$ is a context schema, and $G_1, \ldots, G_n$ is a listing of instances of block schemas from $\mathscr{C}$ in which the types assigned to nominal constants are well-formed with respect to Σ, $\mathbb{N} \setminus \mathbb{N}_\Gamma$, and Ξ. This "typing" of the variable Γ is intended to limit its range to closed contexts obtained by interspersing instances of block schemas from $\mathscr{C}$ in which nominal constants from $\mathbb{N}_\Gamma$ do not appear between instances of $G_1, \ldots, G_n$ obtained by substituting terms formed from Σ and nominal constants not appearing in the support set of the sequent for the variables in Ψ; the substitutions for the variables in Ψ must respect arity typing and the LF types in the resulting context must be well-typed in an arity sense.

The basic notion of meaning for sequents is one that pertains to closed sequents, i.e. ones of the form $\mathbb{N};\emptyset;\emptyset;\Omega \longrightarrow F$. Such a sequent is *valid* if F is valid or one of the assumption formulas in Ω is not valid. A sequent of the general form $\mathbb{N};\Psi;\Xi;\Omega \longrightarrow F$ is then considered valid if all of its instances obtained by substituting closed terms not containing the nominal constants in $\mathbb{N}$ and respecting arity typing constraints for the variables in Ψ and replacing the variables in Ξ with closed contexts respecting their types in the manner described above are valid. The goal of showing that a formula F whose nominal constants are contained in the set $\mathbb{N}$ is valid now reduces to showing the validity of the sequent $\mathbb{N};\emptyset;\emptyset;\emptyset \longrightarrow F$.

3.2 The Rules for Deriving Sequents

The sequent calculus comprises two kinds of rules: those that pertain to the logical symbols and structural aspects of sequents and those that encode the interpretation of atomic formulas as assertions of derivability in LF. We discuss the rules under these categories below, focusing mainly on the latter kind of rules. For paucity of space, we do not present the rules explicitly but rather discuss their intuitive content. We also note that all the rules that we describe have been shown to be sound [13].

3.2.1 Structural and Logical Rules

The calculus includes the usual contraction and weakening rules pertaining to assumption formulas. Also included are rules for adding and removing entries from the support set and the eigenvariables and context variables contexts when these additions do not impact the overall well-formedness of sequents.

Finally, the cut rule, which facilitates the use of well-formed formulas as lemmas, is also present in the collection.

The most basic logical rule is that of an axiom. The main deviation from the usual form for this is the incorporation of the invariance of LF derivability under permutations of the names of context variables; this is realized in our proof system via a notion of equivalence of formulas under permutations of nominal constants. The rules for the connectives and quantifiers take the expected form. The only significant point to note is that eigenvariables that are introduced for (essential) universal quantifiers must be raised over the support set of the sequent to correctly reflect dependencies given our interpretation of sequents [11].

3.2.2 The Treatment of Atomic Formulas

The calculus builds in the understanding of formulas of the form $\{G \vdash M : A\}$ via LF derivability. If A is a type of the form $\Pi x{:}A_1.\, A_2$, then M must have the form $\lambda x.\, M'$ and the atomic formula can be replaced by one of the form $\{G, n : A_1 \vdash M' : A_2\}$ in the sequent; here, n must be a nominal constant not already in the support set and if G contains a context variable then its type annotation must be changed to prohibit the occurrence of n in its instantiations. If A is an atomic type and $\{G \vdash M : A\}$ is the goal formula, then the corresponding rule allows a step to be taken in the validation of the typing judgement; specifically, if M is the term $(h\ M_1\ \ldots\ M_n)$ where h is a constant or a nominal constant to which Σ or G assigns the LF type $\Pi x_1{:}A_1. \ldots \Pi x_n{:}A_n.\, A'$ and A is identical to $A'[\![\{\langle x_1, M_1(A_1)^-\rangle, \ldots, \langle x_n, M_n, (A_n)^-\rangle\}]\!]$, then the rule leads to the consideration of the derivation of sequents in which the goal formula is changed to $\{G \vdash M_i[\![\{\langle x_1, M_1(A_1)^-\rangle, \ldots, \langle x_{i-1}, M_{i-1}, (A_{i-1})^-\rangle\}]\!]\}$ for $1 \leq i \leq n$. Note that if G begins with a context variable Γ, then the assignments in the blocks in the "type" of Γ are considered to be assignments in G.

The most contentful part of the treatment of atomic formulas is when the formula $\{G \vdash M : A\}$ in which A is an atomic type appears as an assumption formula in the sequent. The example in Section 2.3 demonstrates the *case analysis* style of reasoning that we would want to capture in the proof rule; we must identify all the possibilities for the valid closed instances of this formula and analyze the validity of the sequent based on these instances. The difficulty, however, is that there may be far too many closed instances to consider explicitly. This issue can be refined into two specific problems that must be addressed. First, the context G might begin with a context variable and we must then identify a realistic way to consider all the instantiations of that variable that yield an actual, closed context. Second, we must describe a manageable approach to considering all possible instantiations for the term variables that may appear in $\{G \vdash M : A\}$.

The first problem is solved in the enunciation of the rule through an *incremental elaboration* of a context variable that is driven by the atomic formula under scrutiny. Suppose that G begins with the context variable Γ corresponding to which there is the declaration $\Gamma{\uparrow}\mathbb{N}_\Gamma : \mathscr{C}[G_1, \ldots, G_k]$ in the context variables context. We would at the outset need to consider all the constants in Σ and all the nominal constants identified explicitly in G, which includes the ones declared in $G_1, \ldots, G_k$, as potential heads for M in the formula $\{G \vdash M : A\}$. Additionally, this head may come from a part of Γ that has not yet been made explicit. To account for this, the rule considers all the possible instances for the block declarations constituting $\mathscr{C}$ and all possible locations for such blocks in the sequence $G_1, \ldots, G_k$. We note that the number of such instances that have to be examined is finite because it suffices to consider exactly one representative for any nominal constant that does not appear in the support set of the sequent; this observation follows from the invariance of LF typing judgements under permutations of names for the variables bound in the context.

The second problem is addressed by first describing a notion of unification that will ensure that all closed instances will be considered and then identifying the idea of a *covering set of unifiers* that enables

us to avoid an exhaustive consideration. To elaborate a little on this approach, suppose that the (nominal) constant h with LF type $\Pi x_1{:}A_1.\ \ldots \Pi x_n{:}A_n.A'$ has been identified as the candidate head for M. Further, for $1 \leq i \leq n$, let t_i be terms representing fresh variables raised over the support set of the sequent.[1] Then, based on the notion of unification described, for each substitution θ that unifies $(h\ t_1\ \ldots\ t_n)$ and M on the one hand and A and $A'[\![\{\langle x_1, t_1(A_1)^-\rangle, \ldots, \langle x_n, t_n, (A_n)^-\rangle\}]\!]$ on the other, it suffices to consider the derivability of the sequent that results from replacing $\{G \vdash M : A\}$ in the original sequent with the set of formulas

$$\{\{G \vdash t_i : A_i[\![\{\langle x_1, t_1(A_1)^-\rangle, \ldots, \langle x_{i-1}, t_{i-1}, (A_{i-1})^-\rangle\}]\!]\} \mid 1 \leq i \leq n\}$$

and then applying the substitution θ. However, this will still result in a a large number of cases since the collection of unifiers must include all relevant closed instances for the analysis to be sound. The notion of a covering set provides a means for limiting attention to a small subset of unifiers while still preserving soundness.

3.2.3 Induction over the Heights of LF Derivations

The example in Subsection 2.3 also illustrates the role of induction over the heights of LF typing derivations in informal reasoning. This kind of induction is realized in our sequent calculus by using an annotation based scheme inspired by Abella [4, 6]. Specifically, we add to the syntax two additional forms of atomic formulas: $\{G \vdash M : A\}^{@^i}$ and $\{G \vdash M : A\}^{*^i}$. These represent, respectively, a formula that has an LF derivation of some given height and another formula of strictly smaller height; the latter formula is obtained typically by an unfolding step embodied in the use of a case analysis rule. The index i on the annotation symbol is used to identify distinct pairs of @ and $*$ annotations. The induction proof rule then has the form

$$\frac{\begin{array}{c}\mathbb{N};\Psi;\Xi;\Omega, \mathscr{Q}_1.(F_1 \supset \ldots \supset \mathscr{Q}_{k-1}.(F_{k-1} \supset \mathscr{Q}_k.(\{G \vdash M : A\}^{*^i} \supset \ldots \supset F_n))) \longrightarrow \\ \mathscr{Q}_1.(F_1 \supset \ldots \supset \mathscr{Q}_{k-1}.(F_{k-1} \supset \mathscr{Q}_k.(\{G \vdash M : A\}^{@^i} \supset \ldots \supset F_n)))\end{array}}{\mathbb{N};\Psi;\Xi;\Omega \longrightarrow \mathscr{Q}_1.(F_1 \supset \ldots \supset \mathscr{Q}_{k-1}.(F_{k-1} \supset \mathscr{Q}_k.(\{G \vdash M : A\} \supset \ldots \supset F_n)))}\ \textit{induction}$$

where $\mathscr{Q}_i$ represent a sequence of context quantifiers or universal term quantifiers and the annotations $@^i$ and $*^i$ must not already appear in the conclusion sequent. The premise of this proof rule can be viewed as providing a proof schema for constructing an argument of validity for any particular height m, and so by an inductive argument we can conclude that the formula will be valid regardless of the derivation height. This idea is made precise in a proof of soundness for the rule [13, 19].

For this proof rule to be useful in reasoning, we will need a form of case analysis which permits us to move from a formula annotated with @ to one annotated by $*$ when reduced. Such a mechanism is built into the proof system and is used in the examples in the next section.

3.2.4 Rules Encoding Metatheorems Concerning LF Derivability

Typing judgements in LF admit several metatheorems that are useful in reasoning about specifications: if $\Gamma \vdash_\Sigma M \Leftarrow A$ has a derivation then so does $\Gamma, x : A' \vdash_\Sigma M \Leftarrow A$ for a fresh variable x and any well-formed type A' (weakening); if $\Gamma, x : A' \vdash_\Sigma M \Leftarrow A$ has a derivation and x does not appear in M or A then so also does $\Gamma \vdash_\Sigma M \Leftarrow A$ (strengthening); if $\Gamma_1, x_1 : A_1, x_2 : A_2, \Gamma_2 \vdash_\Sigma M \Leftarrow A$ has a derivation and

[1] If $n_1, \ldots, n_\ell$ is a listing of $\mathbb{N}$ then, for $1 \leq i \leq n$, t_i is the term $(z_i\ n_1\ \cdots\ n_\ell)$ where z_i is a fresh variable of suitable arity type.

x_1 does not appear in A_2 then $\Gamma_1, x_2 : A_1, x_1 : A_1, \Gamma_2 \vdash_\Sigma M \Leftarrow A$ must have a derivation (permutation); and if $\Gamma_1 \vdash_\Sigma M' \Leftarrow A'$ and $\Gamma_1, x : A', \Gamma_2 \vdash_\Sigma M \Leftarrow A$ have derivations then there must be a derivation for $\Gamma_1, \Gamma_2[\![\{\langle x, M', (A')^-\rangle\}]\!] \vdash_\Sigma M[\![\{\langle x, M', (A')^-\rangle\}]\!] \Leftarrow A[\![\{\langle x, M', (A')^-\rangle\}]\!]$ (substitution). Moreover, in the first three cases, the derivations are structurally similar, e.g. they have the same heights. These metatheorems are built into the sequent calculus via (sound) axioms. For example, (one version of) the strengthening metatheorem is encoded in the axiom

$$\frac{n \text{ does not appear in } M, A, \text{ or the explicit bindings in } G}{\mathbb{N}; \Psi; \Xi; \Omega \longrightarrow \{G, n : B \vdash M : A\} \supset \{G \vdash M : A\}} \; \textit{LF-str}$$

These axioms can be combined with the cut rule to encode the informal reasoning process.

4 The Adelfa System and its Use in Reasoning

In this section we expose the Adelfa system that provides support for mechanizing arguments of validity for formulas in $\mathscr{L}_{LF}$ using the proof system outlined in Section 3. The first subsection below provides an overview of Adelfa. We then consider a few example reasoning tasks to indicate how the system is intended to be used and also to give a sense for its capabilities.

4.1 Overview of Adelfa

The structure of Adelfa is inspired by the proof assistant Abella [4, 6]. Each development in Adelfa is parameterized by an LF signature which is identified by an initial declaration. The interaction then proceeds to a mode in which context schemas can be identified and theorems can be posited. When a theorem has been presented, the interaction enters a *proof mode* in which the user directs the construction of a proof for a suitable sequent using a repertoire of tactics. These tactics encode the (sound) application of a combination of rules from the sequent calculus to produce a new proof state represented by a collection of sequents for which proofs need to be constructed. We note in this context the presence of a *search* tactic that attempts to construct a proof for a sequent with an atomic goal formula through the repeated use of LF typing rules. If a tactic yields more than one proof obligation then these are ordered in a predetermined way and all existing obligations are maintained in a stack, to be eventually treated in order for the proof to be completed.

The syntax of LF terms and types in Adelfa follows that of fully explicit Twelf. The type $\Pi x{:}A.\, B$ is represented by $\{x : A\}\ B$ and the abstraction term $\lambda x.\, M$ is represented by $[x]\ M$. A context schema definition is introduced by the keyword *Schema* and it associates an identifier with the schema that is to be used in its place in the subsequent development. The block schemas defining a context schema are separated by ;, the schematic variable declarations of each block schema are presented within braces, and the block itself is surrounded by parenthesis. A theorem is introduced by the keyword *Theorem*, and the declaration associates an identifier with the formula to be proved. Once a theorem has been successfully proved this identifier is available for use as a lemma (via the cut rule) in later proofs.

The following table identifies the formula syntax of Adelfa.

Formula	Adelfa Syntax	Formula	Adelfa Syntax
$\forall x : \alpha.F$	forall $x : \alpha.\ F$	$F_1 \vee F_2$	$F_1 \setminus/ F_2$
$\exists x : \alpha.F$	exists $x : \alpha.\ F$	$F_1 \supset F_2$	F_1 => F_2
$\Pi G : \mathscr{C}.F$	ctx $G : \mathscr{C}.\ F$	$\top$	true
$F_1 \wedge F_2$	$F_1 /\backslash F_2$	$\bot$	false

A key issue in the Adelfa implementation is the realization of case analysis. In Section 3, we have discussed the basis for this rule in a special form of unification and also the role of the idea of covering sets of unifiers in its practical realization. We have shown that, in the situations where it is applicable, the notion of higher-order pattern unification [10] can be adapted to provide such covering sets of solutions [19]. Adelfa employs this approach, using the higher-order pattern unification algorithm described by Nadathur and Linnell [12] in its implementation. The benefit of following this course is that the covering set of solutions comprises a single substitution.

4.2 Example Developments in Adelfa

We begin by discussing the formalization of the proof of type uniqueness for the STLC in Adelfa, thereby demonstrating the use of both induction and case analysis in reasoning. We then show the usefulness of LF metatheorems by considering the proof of cut admissibility for a simple sequent calculus. Finally, we demonstrate the flexibility of the logic through a simple example which is of a form that cannot be directly represented as a function type; we will see in the next section that the ability to reason directly about such formulas distinguishes Adelfa from the Twelf family of systems.

4.2.1 Type Uniqueness for the STLC

The informal argument in Section 2.3 had identified a context schema that is relevant to this example. This schema would be presented to Adelfa via the following declaration:

```
Schema c := {T:o}(x:tm,y:of x T).
```

The proof of the actual type uniqueness theorem had relied on a strengthening lemma concerning the equality relation between encodings of types in the STLC. We elide the proof of this lemma and its use in the formalization, focusing instead on the proof of the formula

$$\Pi\Gamma : c.\forall E : o.\forall T_1 : o.\forall T_2 : o.\forall D_1 : o.\forall D_2 : o.\{\Gamma \vdash T_1 : ty\} \supset \{\Gamma \vdash T_2 : ty\} \supset \{\Gamma \vdash D_1 : of\ E\ T_1\} \supset \{\Gamma \vdash D_2 : of\ E\ T_2\} \supset \exists D_3 : o.\{\Gamma \vdash D_3 : eq\ T_1\ T_2\};$$

the full development of this example and the others in this paper are available with the Adelfa source from the Adelfa web page [1].

The proof development process starts with the presentation of the above formula as a theorem. This will result in Adelfa displaying the following proof state to the user:

```
Vars:
Nominals:
Contexts:
==================================
ctx G:c, forall E:o T1:o T2:o D1:o D2:o,
  {G |- T1 : ty} => {G |- T2 : ty} => {G |- D1 : of E T1} =>
  {G |- D2 : of E T2} => exists d3:o, {G |- D3 : eq T1 T2}
```

This proof state corresponds transparently to the sequent whose proof will establish the validity of the formula; the components of the sequent to the left of the arrow appear above the double line and the goal formula appears below the double line. To make it easy to reference particular formulas during reasoning, a name is associated with each assumption formula in the sequent. In the general case, where there can be more than one proof obligation, the remaining obligations will be indicated by a listing of just their goal formulas below the obligation currently in focus.

The informal argument used a induction on the height of the LF derivation represented by the third atomic formula that appears in the goal formula shown. To initiate this process in Adelfa, we would invoke an induction tactic command, indicating the atomic formula that is to be the focus of the induction. This causes the goal formula with the third atomic formula annotated with a * to be added to the assumption formulas and the goal formula to be changed to one in which the third formula has the annotation @. At this stage, we may invoke a tactic command to apply a sequence of right introduction rules, followd by another command to invoke case analysis on the assumption formula {G |- D1 : of E T1}@. Doing so results in the following proof state:

```
Vars: a1:o -> o -> o, R:o -> o, T3:o, T4:o, D2:o, T2:o
Nominals: n2:o, n1:o, n:o
Contexts: G{n2, n1, n}:c[]
IH: ctx G:c, forall E:o T1:o T2:o D1:o D2:o,
      {G |- T1 : ty} => {G |- T2 : ty} => {G |- D1 : of E T1}* =>
      {G |- D2 : of E T2} => exists D3:o, {G |- D3 : eq T1 T2}
H1:{G |- arr T3 T4 : ty}
H2:{G |- T2 : ty}
H4:{G |- D2 : of (abs T3 ([x]R x)) T2}
H5:{G |- T3 : ty}*
H6:{G |- T4 : ty}*
H7:{G, n:tm |- R n : tm}*
H8:{G, n1:tm, n2:of n1 T3 |- a1 n1 n2 : of (R n1) T4}*

==================================
exists D3, {G |- D3 : eq (arr T3 T4) T2}

Subgoal 2: exists D3:o, {G |- D3 : eq T1 T2}
Subgoal 3: exists D3:o, {G |- D3 : eq (T1 n n1) (T2 n n1)}
```

As is to be anticipated, case analysis results in the consideration of three different possibilities: that where the head of D1 is, respectively, the constant *of_abs*, the constant *of_app*, or a nominal constant from the context G. The first of these cases is shown in elaborated form, the other two become additional proof obligations. The analysis based on the first case replaces the original assumption formula with new ones according to the type associated with *of_abs*. The annotation on these formulas is changed from @ to *, to represent the fact that the derivations of the LF judgements associated with them must be of smaller height. Finally, those of these formulas that represent typing judgements for (LF) abstractions are analyzed further, resulting in the introduction of the new nominal constants n, n1, and n2 into the corresponding assumption formulas, and an annotation of the context variable G that indicates that only those instantiations in which these nominal constants do not appear must be considered for it.

At this point, a case analysis based on the formula identified by H4 will identify only a single case, as the term structure (abs T3 ([x]R x)) uniquely identifies a single structure for D2, that where the head is *of_abs*. We may now use the weakening metatheorem for LF to change the context for the assumption formula identified by H6 and the ones resulting from the case analysis of the formula identified by H4 to (G, n1:tm, n2:of n1 T3). Noting that this context must satisfy the context schema ctx if G satisfies it, we have the ingredients in place to utilize the induction hypothesis, i.e. the assumption formula identified by IH, to be able to add the formula

```
{G, n1:tm, n2:of n1 T3 |- D1 n5 n4 n3 n2 n1 n : eq T4 T5}
```

to the collection of assumption formulas; note that the conclusion formula for the sequent would have also become

```
exists D3, {G |- D3 : eq (arr T3 T4) (arr T3 T5)}
```

as a result of the case analysis. Analyzing this assumption formula will produce a single case in which `D1` is bound to `(refl T5)` and `T4` is replaced by `T5` throughout the sequent. This branch of the proof can then be completed by instantiating the existential quantifier in the goal formula with the term `(refl (arr T3 T5))` and using the assumption formulas that indicate that `T3` and `T5` must have the type *ty* to conclude that `(arr T3 T5)` must also be similarly typed.

The second subgoal, the one corresponding to the application case, will now be presented in elaborated form. We will elide a discussion of this case because it does not illustrate any capabilities beyond those already exhibited in the consideration of the abstraction case. Instead we will jump to the final subgoal which is manifest in the following proof state:

```
Vars: D2:o -> o -> o, T2:o -> o -> o, T1:o -> o -> o
Nominals: n1:o, n:o
Contexts: G{}:c[(n:tm, n1:of n (T1 n n1))]
IH: ctx G:c, forall E:o T1:o T2:o D1:o D2:o,
      {G |- T1 : ty} => {G |- T2 : ty} => {G |- D1 : of E T1}* =>
      {G |- D2 : of E T2} => exists D3:o, {G |- D3 : eq T1 T2}
H1:{G |- T1 n n1 : ty}
H2:{G |- T2 n n1 : ty}
H4:{G |- D2 n n1 : of n (T2 n n1)}

==================================
exists D3, {G |- D3 : eq (T1 n n1) (T2 n n1)}
```

As we can see in this state, a new block has been elaborated in the context variable type of `G`. The nominal constants that are introduced by this elaboration may be used in the terms that instantiate the eigenvariables `D2`, `T2`, and `T1` in the original sequent, a fact that is realizes here by raising these variables over the nominal constants. The key to the proof for this case is that case analysis based on the assumption formula `H4` will only identify a single relevant case: that when `D2` is `n1` and `T2` is identified with `T1`. This is because `n` represents a unique nominal constant that cannot be matched with any other term or nominal constant, and a name can have at most one binding in `G`.

4.2.2 Cut Admissibility for the Intuitionistic Sequent Calculus

The example we consider now is that of proving the admissibility of cut for an intuitionistic sequent calculus. In this proof, there is a need to consider the weakening of the antecedents of sequents. We propose an encoding of sequents below under which the weakening rule applied to sequents can be modelled by a weakening applied to LF typing judgements. Thus, this example illustrates the use of LF metatheorems encoded in Adelfa in directly realizing reasoning steps in informal proofs.

We will actually consider only a fragment of the intuitionistic sequent calculus for propositional logic in this illustration. In particular, this fragment includes only the logical constant $\top$ and the connectives for conjunction and implication. An LF signature that encodes this fragment appears below.

$$\begin{array}{ll} \mathit{proptm} : \mathit{Type} & \mathit{top} : \mathit{proptm} \\ \mathit{hyp} : \mathit{proptm} \rightarrow \mathit{Type} & \mathit{imp} : \mathit{proptm} \rightarrow \mathit{proptm} \rightarrow \mathit{proptm} \\ \mathit{conc} : \mathit{proptm} \rightarrow \mathit{Type} & \mathit{and} : \mathit{proptm} \rightarrow \mathit{proptm} \rightarrow \mathit{proptm} \end{array}$$

$init : \Pi A{:}proptm.\, \Pi D{:}hyp\, A.\, conc\, A$

$topR : conc\, top$

$andL : \Pi A{:}proptm.\, \Pi B{:}proptm.\, \Pi C{:}proptm.\, \Pi D1{:}(\Pi x{:}hyp\, A.\, \Pi y{:}hyp\, B.\, conc\, C).$
$\quad \Pi D2{:}hyp\, (and\, A\, B).\, conc\, C$

$andR : \Pi A{:}proptm.\, \Pi B{:}proptm.\, \Pi D_1{:}conc\, A.\, \Pi D_2{:}conc\, B.\, conc\, (and\, A\, B)$

$impL : \Pi A{:}proptm.\, \Pi B{:}proptm.\, \Pi C{:}proptm.\, \Pi D_1{:}conc\, A.\, \Pi D_2{:}(\Pi x{:}hyp\, B.\, conc\, C).$
$\quad \Pi D_3{:}hyp\, (imp\, A\, B).\, conc\, C$

$impR : \Pi A{:}proptm.\, \Pi B{:}proptm.\, \Pi D{:}(\Pi x{:}hyp\, A.\, conc\, B).\, conc\, (imp\, A\, B)$

The propositions of this calculus are encoded as terms of type *proptm*. The two LF type constructors *hyp* and *conc* that take terms of type *proptm* as arguments are used to identify propositions that are hypotheses and conclusions of a sequent respectively. More specifically, a sequent of the form $A_1, \ldots A_n \longrightarrow B$ is represented under this encoding by the LF typing judgement

$$x_1 : hyp\, \overline{A_1}, \ldots, x_n : hyp\, \overline{A_n} \vdash_\Sigma c \Leftarrow conc\, \overline{B},$$

where $\overline{A}$ denotes the encoding of the proposition A.

The contexts relevant to this example can be characterized by the following schema declaration:

```
Schema cutctx := {A : o}(x : hyp A)
```

Relative to the given signature and schema declaration, the admissibility of cut is captured by the following formula:

$$\Pi G : cutctx.\forall a : o.\forall b : o.\forall d_1 : o.\forall d_2 : o \to o.$$
$$\{\cdot \vdash a : proptm\} \supset \{G \vdash d_1 : conc\, a\} \supset \{G \vdash \lambda x.\, d_2\, x : \Pi x{:}hyp\, a.\, conc\, b\} \supset \exists d : o.\{G \vdash d : conc\, b\}$$

Note that in the third antecedent the term variable d_2 is permitted to depend on the cut formula a through the explicit dependency on the bound variable x, reflecting the fact that this proof can use the "cut formula."

The informal proof that directs our development uses a nested induction. The first induction is on the structure of the cut formula, expressed by the formula $\{\cdot \vdash a : proptm\}$. The second induction is on the height of the derivation that uses the cut formula, expressed by $\{G \vdash \lambda x.\, d_2\, x : \Pi x{:}hyp\, a.\, conc\, b\}$. The full proof is based on considering the different possibilities for the last rule in the proof with the cut formula included in the assumption set. In this illustration, we consider only the case where this is the rule that introduces an implication on the right side of the sequent. In the Adelfa proof, the indicated analysis is reflected in the case analysis of the formula $\{G, n : hyp\, a \vdash d_2\, n : conc\, b\}$, which is derived from $\{G \vdash \lambda x.\, d_2\, x : \Pi x{:}hyp\, a.\, conc\, b\}$, and the case to be considered is that where the head of the term $(d_2\, n)$ is chosen to be *impR*.

One of the characteristics of LF typing judgements is that the assignments in typing contexts are ordered so as to reflect possible dependencies. However, in the object system under consideration, the ordering of propositions on the left of the sequent arrow is unimportant. This is evident in our encoding by the fact that there are no means to represent dependencies in the kind of contexts in consideration. Thus, we may use the permutation metatheorem to realize reordering in the context in a proof development that requires this. We will need to employ this idea in order to ensure that the context structure has the form $(G, n : hyp\, a)$ as is needed to invoke the induction hypothesis.

To get to the details of the case under consideration, we note that the cut formula is preserved through the concluding rule in the derivation. The informal argument then uses the induction hypothesis to obtain proofs of the premises of the rule but where the cut formula is left out of the left side of the sequents.

These proofs are then combined to get a proof for the concluding sequent, again with the cut formula left out from the premises, using the same rule to introduce an implication on the right. It is this argument that we want to mimic in the Adelfa development.

The proof state at the start of the case that is the focus of our discussion is depicted below:

```
Vars: d:o -> o -> o, b1:o, b2:o, d1:o, a:o
Nominals: n1:o, n:o
Contexts: G:cutctx[]
IH:ctx G:cutctx. forall a:o, forall b:o, forall d1:o, forall d2:o -> o,
        {a : proptm}* => {G |- d1 : conc a} =>
            {G, n:hyp a |- d2 n : conc b} => exists d:o, {G |- d : conc b}
IH1:ctx G:cutctx. forall a:o, forall b:o, forall d:o1, forall d2:o -> o,
        {a : proptm}@ => {G |- d1 : conc a} =>
            {G, n:hyp a |- d2 n : conc b}** => exists d:o, {G |- d : conc b}
H1:{a : proptm}@
H2:{G |- d1 : conc a}
H3:{G, n:hyp a |- impR b1 b2 ([x] d n x) : conc (imp b1 b2)}@@
H4:{G, n:hyp a |- b1 : proptm}**
H5:{G, n:hyp a |- b2 : proptm}**
H6:{G, n:hyp a, n1:hyp b1 |- d n n1 : conc b2}**
==================================
exists d, {G |- d : conc (imp b1 b2)}
```

For simplicity, we have elided the other subgoals that remain in this display. Our objective in this case is to add to the assumptions the formula `{G, n1:hyp b1 |- D : conc b2}` for some term `D`; from this, we can easily construct a term that we can use the instantiate the existential quantifier in the goal formula to conclude the proof. We would like to use the inductive hypothesis identified by `IH1` towards this end. To be able to do this, we would need to extend the context expression in the assumption formula identified by `H2` with a new binding of type `(hyp b1)`. We can do this using a tactic command that encodes the weakening metatheorem for LF. In actually carrying out this step, we would additionally use the strengthening metatheorem for LF to conclude that the type `(hyp b)` is well-formed in the relevant context because of the assumption identified by `H4` and the fact that `b1` cannot depend on `n`. We must also rearrange the type assignments in the context in the assumption formula `H6` so that the assignment corresponding to the cut formula appears at the end. This can be realized using the tactic command that encodes the permutation metatheorem in LF, a use of which will be valid for the reasons identified in the earlier discussion.

The end result of applying these reasoning steps is a state of the following form:

```
Vars: d:o -> o -> o, b1:o, b2:o, d1:o, a:o
Nominals: n1:o, n:o
Contexts: G:cutctx[]
IH:ctx G:cutctx. forall a:o, forall b:o, forall d1:o, forall d2:o -> o,
        {a : proptm}* => {G |- d1 : conc a} =>
            {G, n:hyp a |- d2 n : conc b} => exists d:o, {G |- d : conc b}
IH1:ctx G:cutctx. forall a:o, forall b:o, forall d1:o, forall d2:o -> o,
        {a : proptm}@ => {G |- d1 : conc a} =>
            {G, n:hyp a |- d2 n : conc b}** => exists d:o, {G |- d : conc b}
```

```
H1:{a : proptm}@
H2:{G |- d1 : conc a}
H3:{G, n:hyp a |- impR b1 b2 ([x] d n x) : conc (imp b1 b2)}@@
H4:{G, n:hyp a |- b1 : proptm}**
H5:{G, n:hyp a |- b2 : proptm}**
H6:{G, n:hyp a, n1:hyp b1 |- d n n1 : conc b2}**
H7:{G |- b1 : proptm}**
H8:{G, n1:hyp b1 |- d1 : conc a}
H9:{G, n1:hyp b1, n:hyp a |- d n n1 : conc b2}**
==================================
exists d, {G |- d : conc (imp b1 b2)}
```

We can now use the induction hypothesis `IH1` to the assumption formulas `H1`, `H8`, and `H9` to introduce a new term variable `d'` and the assumption formula `{G, n1:hyp b1 |- d' n1 : conc b2}`, from which we can complete the proof as previously indicated.

4.2.3 A Disjunctive Property for Natural Numbers

The formulas that we have considered up to this point have had a "function" structure: for every term satisfying certain typing constraints they have posited the existence of a term satisfying another typing constraint. Towards exhibiting the flexibility of the logic underlying Adelfa, we now consider an example of a theorem that *does not* adhere to this structure. This example is based on the following signature that encodes natural numbers and the even and odd properties pertaining to these numbers.

$nat : Type$	$even : nat \to Type$	$odd : nat \to Type$
$z : nat$	$e\text{-}z : even\ z$	$e\text{-}o : \Pi N{:}nat.\ even\ N \to odd\ (s\ N)$
$s : nat \to nat$	$o\text{-}e : \Pi N{:}nat.\ odd\ N \to even\ (s\ N)$	

Given this signature, we can express the property that every natural number is either even or odd:

$$\forall N : o.\{\cdot \vdash N : nat\} \supset (\exists D : o.\{\cdot \vdash D : even\ N\}) \vee (\exists D : o.\{\cdot \vdash D : odd\ N\}).$$

Following the obvious informal proof, this formula can be proved by induction on the height of the derivation of the LF typing judgement $\cdot \vdash_\Sigma N \Leftarrow nat$. The application of the induction hypothesis yields two possible cases and in each case we pick a disjunct in the conclusion that we can show to be true.

5 Conclusion

This paper has described the Adelfa system for reasoning about specifications written in LF. Adelfa is based on the logic $\mathscr{L}_{LF}$ in which the atomic formulas represent typing judgements in LF and where quantification is permitted over terms and over contexts that are characterized by context schemas. We have sketched a proof system for this logic and have demonstrated the effectiveness of Adelfa, which realizes this proof system, through a few examples. We have developed formalizations beyond the ones discussed here, such as type preservation for the STLC and subtyping for $F_{<:}$; the latter is a problem from the PoPLMark collection [3].

While this work is not unique in its objectives, it differs from other developments in how it attempts to achieve them. One prominent approach in this context is that adopted by what we refer to as the "Twelf family." The first exemplar of this approach is the Twelf system [15] that allows properties of interest to be characterized by types and the validity of such properties to be demonstrated by exhibiting

the totality as functions of inhabitants of these types. This approach has achieved much success but it is also limited by the fact that properties that can be reasoned about must be encodable by a function type, i.e. they must be formulas of the $\forall...\exists...$ form. By contrast, the logic underlying Adelfa has a much more flexible structure, allowing us, for example, to represent properties with a disjunctive structure as we have seen in Section 4.2.3; the reader might want to consult the encoding of this property provided in the wiki page of the Twelf project that covers output factoring to understand more fully the content of this observation [2]. Another virtue of our approach is that we are able to exhibit an explicit proof for properties at the end of a development. This drawback is mitigated to an extent in the Twelf context by the presence of the logic M_2^+ [18] that provides a formal counterpart to the approach to reasoning embodied in the Twelf system [22].

The Twelf approach differs in another way from the approach described here: it does not provide an explicit means for quantifying over contexts. It is possible to parameterize a development by a context description, but it is one fixed context that then permeates the development. As an example, it is not possible to express the strengthening lemma pertaining to equality of types in the STLC that we discussed in Section 2.3. The Beluga system [17], which otherwise shares the characteristics of the Twelf system, alleviates this problem by using a richer version of type theory that allows for an explicit treatment of contexts as its basis [14].

An alternative approach, that is more in alignment with what we have described here, is one that uses a translation of LF specifications into predicate logic to then be reasoned about using the Abella system [20]. This approach has several auxiliary benefits deriving from the expressiveness of the logic underlying Abella [7]; for example, it is possible to define relations between contexts, to treat binding notions explicitly in the reasoning process through the ∇-quantifier, and to use inductive (and co-inductive) definitions in the reasoning logic. In future work, we would like to examine how to derive some of these benefits in the context of Adelfa as well. There are, however, also some drawbacks to the translation-based approach. One of the problems is that it is based on a "theorem" about the translation [5] that we know to be false for the version of LF that it pertains to.[2] Another issue is that proof steps that can be taken in Abella are governed by the logic underlying that system and these allow for many more possibilities than are sensible in the LF context. Moreover, it is not immediately clear how the reasoning steps that are natural with LF specifications translate into macro Abella steps relative to the translation. In this respect, one interesting outcome of the work on the logic underlying the Adelfa system might be an understanding of how one might build within Abella (or other related proof assistants) a targeted capability for reasoning about LF specifications. This also raises another interesting possibility that is worthy of attention, that of providing an alternative justification for the proof system for $\mathscr{L}_{LF}$ based on the translation.

Acknowledgements

This paper is based upon work supported by the National Science Foundation under Grant No. CCF-1617771. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[2]The theorem in question states that a (closed) LF typing judgement is derivable if and only if the translated form of the judgement is derivable in the relevant predicate logic. This is true in the forward direction. However, in the version of LF that the theorem is about, the translation loses typing information from terms and, hence, the inverse translation is ambiguous. Thus, it is only a weaker conclusion that can be drawn, that there is *some* LF judgement that is derivable if the formula in predicate logic is derivable; there is no guarantee that this is the same LF judgement that we started out with.

References

[1] *The Adelfa System*. http://adelfa.cs.umn.edu/.

[2] *The Twelf Project*. http://twelf.org/.

[3] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich & Steve Zdancewic (2005): *Mechanized Metatheory for the Masses: The POPLmark Challenge*. In: *Theorem Proving in Higher Order Logics: 18th International Conference*, *LNCS* 3603, Springer, pp. 50–65, doi:10.1007/11541868_4.

[4] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A System for Reasoning about Relational Specifications*. *Journal of Formalized Reasoning* 7(2), doi:10.6092/issn.1972-5787/4650. Available at http://jfr.unibo.it/article/download/4650/4137.

[5] Amy Felty & Dale Miller (1990): *Encoding a Dependent-Type λ-Calculus in a Logic Programming Language*. In Mark Stickel, editor: *Proceedings of the 1990 Conference on Automated Deduction*, *LNAI* 449, Springer, pp. 221–235, doi:10.1007/3-540-52885-7_90.

[6] Andrew Gacek (2009): *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. Ph.D. thesis, University of Minnesota.

[7] Andrew Gacek, Dale Miller & Gopalan Nadathur (2011): *Nominal Abstraction*. *Information and Computation* 209(1), pp. 48–73, doi:10.1016/j.ic.2010.09.004.

[8] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[9] Robert Harper & Daniel R. Licata (2007): *Mechanizing Metatheory in a Logical Framework*. *Journal of Functional Programming* 17(4–5), pp. 613–673, doi:10.1017/S0956796807006430.

[10] Dale Miller (1991): *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*. *J. of Logic and Computation* 1(4), pp. 497–536, doi:10.1093/logcom/1.4.497.

[11] Dale Miller (1992): *Unification Under a Mixed Prefix*. *Journal of Symbolic Computation* 14(4), pp. 321–358, doi:10.1016/0747-7171(92)90011-R.

[12] Gopalan Nadathur & Natalie Linnell (2005): *Practical Higher-Order Pattern Unification With On-the-Fly Raising*. In: *ICLP 2005: 21st International Logic Programming Conference*, *LNCS* 3668, Springer, Sitges, Spain, pp. 371–386, doi:10.1007/11562931_28.

[13] Gopalan Nadathur & Mary Southern (2021): *A Logic for Reasoning About LF Specifications*. Available from http://arxiv.org/abs/2107.00111.

[14] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual Model Type Theory*. *ACM Trans. on Computational Logic* 9(3), pp. 1–49, doi:10.1145/1352582.1352591.

[15] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In H. Ganzinger, editor: *16th Conf. on Automated Deduction (CADE)*, *LNAI* 1632, Springer, Trento, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[16] Frank Pfenning & Carsten Schürmann (2002): *Twelf User's Guide*. Available from http://www.cs.cmu.edu/~twelf/guide-1-4.

[17] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In J. Giesl & R. Hähnle, editors: *Fifth International Joint Conference on Automated Reasoning*, *LNCS* 6173, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[18] Carsten Schürmann (2000): *Automating the Meta Theory of Deductive Systems*. Ph.D. thesis, Carnegie Mellon University. Available at http://www.cs.yale.edu/homes/carsten/papers/S00b.ps.gz.

[19] Mary Southern (2021): *A Framework for Reasoning About LF Specifications*. Ph.D. thesis, University of Minnesota.

[20] Mary Southern & Kaustuv Chaudhuri (2014): *A Two-Level Logic Approach to Reasoning About Typed Specification Languages*. In Venkatesh Raman & S. P. Suresh, editors: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 29, Dagstuhl, Germany, pp. 557–569, doi:10.4230/LIPIcs.FSTTCS.2014.557. Available at http://drops.dagstuhl.de/opus/volltexte/2014/4871.

[21] Alwen Tiu (2006): *A Logic for Reasoning about Generic Judgments*. In A. Momigliano & B. Pientka, editors: *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, *ENTCS* 173, pp. 3–18, doi:10.1016/j.entcs.2007.01.016.

[22] Yuting Wang & Gopalan Nadathur (2013): *Towards Extracting Explicit Proofs from Totality Checking in Twelf*. In: *LFMTP 2013 - Proceedings of the 2013 ACM SIGPLAN Workshop on Logical Frameworks and Meta-Languages*, pp. 55–66, doi:10.1145/2503887.2503893.