VeriML: Typed Computation of Logical Terms inside a Language with Effects

Antonis Stampoulis   Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520-8285
{antonis.stampoulis,zhong.shao}@yale.edu

Abstract

Modern proof assistants such as Coq and Isabelle provide high degrees of expressiveness and assurance because they support formal reasoning in higher-order logic and supply explicit machine-checkable proof objects. Unfortunately, large scale proof development in these proof assistants is still an extremely difficult and time-consuming task. One major weakness of these proof assistants is the lack of a single language where users can develop complex tactics and decision procedures using a rich programming model and in a typeful manner. This limits the scalability of the proof development process, as users avoid developing domain-specific tactics and decision procedures.

In this paper, we present VeriML—a novel language design that couples a type-safe effectful computational language with first-class support for manipulating logical terms such as propositions and proofs. The main idea behind our design is to integrate a rich logical framework—similar to the one supported by Coq—inside a computational language inspired by ML. The language design is such that the added features are orthogonal to the rest of the computational language, and also do not require significant additions to the logic language, so soundness is guaranteed. We have built a prototype implementation of VeriML including both its type-checker and an interpreter. We demonstrate the effectiveness of our design by showing a number of type-safe tactics and decision procedures written in VeriML.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Verification

1. Introduction

In recent years, there has been a growing interest in formal verification of substantial software code bases. Two of the most significant examples of this trend is the verification of a full optimizing compiler for a subset of the C language in the CompCert project [Leroy 2009], as well as the verification of the practical operating system microkernel seL4 [Klein et al. 2009]. Both of these efforts use powerful proof assistants such as Coq [Barras et al. 2010] and Isabelle [Nipkow et al. 2002] which support higher-order logic with explicit proof objects. Other verification projects have opted to use first-order automated theorem provers; one such example is the certified garbage collector by Hawblitzel and Petrank [2009].

Still, the actual process of software verification requires significant effort, as clearly evidenced by the above developments. We believe that a large part of this effort could be reduced, if the underlying verification frameworks had better support for extending their automation facilities. During a large proof development, a number of user-defined datatypes are used; being able to define domain-specific decision procedures (for these) can significantly cut back on the manual proof effort required. In other cases, different program logics might need to be defined to reason about different parts of the software being verified, as is argued by Feng et al. [2008] for the case of operating system kernels. In such cases, developing automated provers tailored to these logics would be very desirable.

We thus believe that in order to be truly extensible, a proof development framework should support the following features:

- Use of a well-established logic with well-understood metatheory, that also provides explicit proof objects. This way, the trusted computing base of the verification process is kept at a minimum. The high assurance offered by developments such as CompCert and seL4 owes largely to this characteristic of the proof assistants they are developed on.
- Being able to programmatically reflect on logical terms (e.g., propositions) so that we can write a large number of procedures (e.g., tactics, decision procedures, and automated provers) tailored to solving different proof obligations. Facilities such as LTac [Delahaye 2000], and their use in developments like Chlipala et al. [2009], demonstrate the benefits of this feature.
- An unrestricted programming model for developing these procedures, that permits the use of features such as non-termination and mutable references. The reason for this is that even simple decision procedures might make essential use of imperative data structures and might have complex termination arguments. One such example are decision procedures for the theory of equality with uninterpreted functions [Bradley and Manna 2007]. By enabling an unrestricted programming model, porting such procedures does not require significant re-engineering.
- At the same time, being able to provide certain static guarantees and rich type information to the programmer. Terms of a formal logic come with rich type information: proof objects have "types" representing the propositions that they prove, and propositions themselves are deemed to be valid according to some typing rules. By retaining this information when programmatically manipulating logical terms, we can specify the behavior of the associated code. For example, we could statically specify that a tactic transforms a propositional goal into

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-60558-794-3/10/09...$10.00

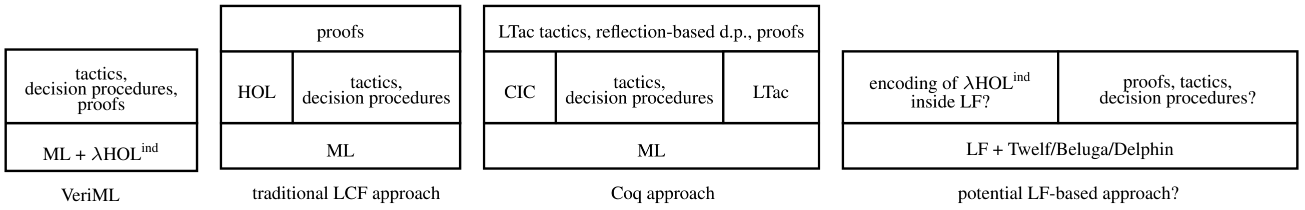
tactics, decision procedures, proofs
ML + λHOL^ind
VeriML
proofs
HOL
tactics, decision procedures
ML
traditional LCF approach
LTac tactics, reflection-based d.p., proofs
CIC
tactics, decision procedures
LTac
ML
Coq approach
encoding of λHOL^ind inside LF?
proofs, tactics, decision procedures?
LF + Twelf/Beluga/Delphin
potential LF-based approach?

Figure 1. Schematic comparison of the structure of related approaches

an equivalent one, by requiring a proof object witnessing this equivalence. Another guarantee we would like is the correct handling of the binding constructs that logical terms include (e.g. quantification in propositions), so that this common source of errors is statically avoided.

A framework that combines these features is not currently available. As a result, existing proof assistants must rely on a mix of languages (with incompatible type systems) to achieve a certain degree of extensibility. In this paper, we present VeriML—a novel language design that aims to support all these features and provide a truly extensible and modular proof development framework. Our paper makes the following new contributions:

- As far as we know, VeriML is the first proof framework that successfully combines a type-safe effectful computational language with first-class support for manipulating rich logical terms such as propositions, proofs, and inductive definitions.
- An important feature of VeriML is the strong separation of roles played by its underlying logic language and computational language. The logic language, $\lambda\text{HOL}^{\text{ind}}$, supports higher-order logic with inductive definitions (as in Coq), so it can both serve as a rich meta logic and be used to define new object logics/languages and reason about their meta theory. All proof objects in VeriML can be represented using $\lambda\text{HOL}^{\text{ind}}$ alone. The computational language is used only for general-purpose programming, including typed manipulation of logical terms.

  This is in sharp contrast to recent work such as Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008] where meta-logical proofs are represented using their computational languages. Maintaining soundness of such proofs when adding imperative features to these languages would be non-trivial, and would put additional burden (e.g. effect annotations) on general-purpose programming.
- We present the complete development of the type system and operational semantics for VeriML, as well as their associated meta-theory . We also show how to adapt contextual modal type theory to work for a rich meta-logic such as $\lambda\text{HOL}^{\text{ind}}$.
- We have built a prototype implementation of VeriML and used it to write a number of type-safe tactics and decision procedures. We use these examples to demonstrate the applicability of our approach and show why it is important to support type-safe handling of binders, general recursion, and imperative features such as arrays and hash tables in VeriML.

The rest of the paper is organized as follows. We first give a high-level overview of VeriML (Sec 2) and then use examples to explain the basic design (Sec 3); we then present the logic language, the computational language, and their meta-theory (Sec 4-5); finally, we describe the implementation and discuss related work.

2. Overview of the language design

We will start off by presenting a high-level overview of our framework design. The first choice we have to make is the formal logic

	static guarantees	non-termination	mutable references	logic with P.P.
Traditional LCF	no	no	yes	(maybe)
LTac	no	yes	no	yes
Reflection-based	yes	no	no	yes
Beluga, Delphin	yes	yes	no	no
VeriML	yes	yes	yes	yes

Table 1. Comparison of different approaches based on features

that we will use to base our framework. We opt to use a higher-order logic with inductive definitions and explicit proof objects. This gives us a high degree of expressivity, enough for software verification as evidenced by the aforementioned large-scale proof developments, while at the same time providing a high level of assurance. Furthermore, we allow a notion of computation inside our logic, by adding support for defining and evaluating total recursive functions. In this way, logical arguments that are based solely on such computation need not be explicitly witnessed in proof objects, significantly reducing their sizes. This notion of computation must of course be terminating, in order to maintain soundness. Because of this characteristic, our logic satisfies what we refer to as the Poincaré Principle (abbreviated as P.P.), following the definition in Barendregt and Geuvers [1999]. Last, we choose to omit certain features like dependent types from our logic, in order to keep its metatheory straightforward. We will see more details about this logic in Section 4. We refer to propositions, inhabitants of inductive types, proof objects, and other terms of this logic as logical terms.

Developing proofs directly inside this logic can be very tedious due to the large amount of detail required; because of this, proof development frameworks provide a set of computational functions (tactics and decision procedures) that produce parts of proofs, so that the proof burden is considerably lessened. The problem that we are interested in is the design of a computational language, so that such functions can be easily and effectively written by the user for the domains they are interested in, leading to a scalable and modular proof development style.

As we have laid out in the introduction, we would like a number of features out of such a computational language: being able to programmatically pattern match on propositions, have a general-purpose programming model available, and provide certain static guarantees to the programmer.

Let us briefly consider how computational approaches in existing proof assistants fare towards those points. A schematic comparison is given in Figure 1, while Table 1 compares existing approaches based on these points. A standard and widely available approach [Slind and Norrish 2008, Harrison 1996, Nipkow et al. 2002, Barras et al. 2010] is to write user-defined tactics and decision procedures inside the implementation language of the proof assistant, which is in most cases a member of the ML family of languages. This gives to the user access to a rich programming model, with non-terminating recursion and imperative data structures. Still, the user has to deal with the implementation details of the framework, and no static guarantees are given whatsoever. All logical terms are essentially identified at the ML type level, leading to an untyped programming style when programming with them.

Another approach is the use of LTac [Delahaye 2000] in the Coq proof assistant: a specialized tactic language that allows pattern matching on propositions, backtracking proof search and general recursion. This language too does not provide any static guarantees, and has occasional issues when dealing with binders and variables. Also, the programming model supported is relatively poor, without support for rich data structures or imperativity.

An interesting approach is the technique of proof-by-reflection [Boutin 1997], where the computational notion inside the logic itself is used in order to create certified decision procedures. While this approach gives very strong static guarantees (total correctness), it does not support non-termination or imperative data structures, limiting the kind of decision procedures we can write. Also, the use of a mix of languages is required in this technique.

In order to combine the benefits of these approaches, we propose a new language design, that couples a general-purpose programming language like ML with first-class support for our logical framework. Furthermore we integrate the type system of the logic inside the type system of the computational language, leading to a dependently typed system. Logical term literals will thus retain the type information that can be statically determined for them. Moreover, a pattern matching construct for logical terms is explicitly added, which is dependently typed too; the type of each branch depends on the specific pattern being matched. We use dependent types only as a way to provide lightweight static guarantees. For example, we can require that a function receives a proposition and returns a proof object for that proposition, ruling out the possibility of returning an "invalid" proof object because of programmer error. Our approach therefore differs from other dependently-typed frameworks like Agda [Norell 2007] or HTT [Nanevski et al. 2006], as we are not necessarily interested in reasoning about the correctness of code written in our computational language. Also, programming in such systems includes an aspect that amounts to proof development, as evidenced e.g. in the Russell framework [Sozeau 2007]. We are interested in how proof development itself can be automated, so our approach is orthogonal to such systems. Our notion of pattern matching on propositions would amount to typecase-like constructs in these languages, which in general are not provided.

Dependently-typed frameworks for computing with LF terms like Beluga [Pientka and Dunfield 2008] and Delphine [Poswolsky and Schürmann 2008], are not ideal for our purposes, because of the lack of imperative features and the fact that encoding a logic like the one we describe inside LF is difficult and has not been demonstrated yet in practice. The reason for this is exactly our logic's support for the Poincaré principle. Still, we draw inspiration from such frameworks.

In order to type-check logical terms, and make sure that binding is handled correctly, we need information about the free variables context that they depend on. Not all logical terms manipulated during evaluation of a program written in our language need to refer to the same context; for example, when pattern matching a quantified proposition like $\forall x : Nat.P$, the variable P might refer to an extra variable compared to the original proposition that was being matched. Therefore, in order to guarantee proper scoping of variables used inside the logical terms, the type system of our computational language tracks the free variable context of the logical terms that are manipulated. This is done by using the main idea of contextual modal type theory [Nanevski et al. 2008, Pientka 2008]. We introduce a notion of contextual logical terms, that is, logical terms that come packaged together with the free variables context they depend on. Our computational language manipulates such terms, instead of normal logical terms, which would also need some external information about their context. A notion of context polymorphism will also need to be introduced, in order to write code that is generic with respect to variable contexts.

3. Programming Examples

In this section we will present a number of programming examples in our computational language in order to demonstrate its use as well as motivate some of our design choices, before presenting the full technical details in later sections. Each example demonstrates one particular feature of the computational language. The examples we will present are successive versions of a tactic that attempts to automatically derive intuitionistic proofs of propositional tautologies (similar to Coq's tauto tactic [Barras et al. 2010]) and a decision procedure for the theory of equality, along with the data structures that they use. Note that we use a somewhat informal style here for presentation purposes. Full details for these examples can be found as part of our implementation at `http://flint.cs.yale.edu/publications/veriml.html`.

3.1 Pattern matching

We will start with the automatic tautology proving tactic which is structured as follows: given a proposition, it will perform pattern matching in order to deconstruct it, and attempt to recursively prove the included subformulas; when this is possible, it will return a proof object of the given proposition. Some preliminary code for such a function follows, which handles only logical conjunction and disjunction, and the True proposition as a base case. Note that we use a monadic do notation in the style of Haskell for the failure monad (computation with the ML option type), and we use the syntax $e_1 || e_2$ where both expressions are of option type in order to choose the second expression when the first one fails.

We use the "holcase $\cdot$ of $\cdots$" construct to perform pattern matching on a logical term. Also, we use the notation $\langle\cdot\rangle$ to denote the lifting of a logical term into a computational language term. Under the hood, this is an existential package which packages a logical term with the unit value. Here we have used it so that our function might return proof objects of the relevant propositions. We have not written out the details of the proof objects themselves to avoid introducing unnecessary technical details at this point, but they are straightforward to arrive at.

```
tauto $P$ = holcase $P$ of
      $P_1 \wedge P_2$    $\mapsto$  do $\mathrm{pf}_1 \leftarrow$ tauto $P_1$;
                       $\mathrm{pf}_2 \leftarrow$ tauto $P_2$;
                       $\langle \cdots$ proof of $P_1 \wedge P_2 \cdots \rangle$
    | $P_1 \vee P_2$    $\mapsto$  (do $\mathrm{pf}_1 \leftarrow$ tauto $P_1$;
                       $\langle \cdots$ proof of $P_1 \vee P_2 \cdots \rangle$) ||
                    (do $\mathrm{pf}_2 \leftarrow$ tauto $P_2$;
                       $\langle \cdots$ proof of $P_1 \vee P_2 \cdots \rangle$)
    | True       $\mapsto$ Some $\langle \cdots$ proof of True $\cdots \rangle$
    | $P'$         $\mapsto$ None
```

We assign a dependent type to this function, requiring that the proof objects returned by the function prove the proposition that is given as an argument. This is done by having the pattern matching be dependently typed too; we will see the details of how this is achieved after we describe our type system. We use the notation $\mathsf{LT}(\cdot)$ to denote lifting a logical term into the level of computational types; thus the function's type will be:

$$\Pi P : \mathsf{Prop}.\mathsf{option}\ \mathsf{LT}(P)$$

3.2 Handling binders and free variables

Next we want to handle universally quantified propositions. Our procedure needs to go below the quantifier, and attempt to prove the body of the proposition; it will succeed if this can be done parametrically with regards to the new variable. In this case, the procedure we have described so far will need to be run recursively on a proposition with a new free variable (the body of the quantifier). To avoid capture of variables, we need to keep track of the free variables used so far in the proposition; we do this by having the function also take the free variables context of the proposition as an argument. Also, we annotate logical terms with the free variables context they refer to. Thus we will be handling *contextual terms* in our computational language, i.e., logical terms packaged with the free variables context they refer to. We use ϕ to range over contexts (their kind is abbreviated as ctx), and the notation $[\phi]\,\mathsf{Prop}$ to denote propositions living in a context ϕ; a similar notation is used for other logical terms. The new type of our function will thus be:

$$\Pi\phi : \mathsf{ctx}.\Pi P : [\phi]\,\mathsf{Prop}.\mathsf{option}\ \mathsf{LT}([\phi]\,P)$$

The new code for the function will look like the following:

```
tauto $\phi$ $P$  =  holcase $P$ of
    $P_1 \wedge P_2$      $\mapsto$   do pf$_1$ $\leftarrow$ tauto $\phi$ $P_1$;
                          pf$_2$ $\leftarrow$ tauto $\phi$ $P_2$;
                          $\langle \cdots$ proof of $P_1 \wedge P_2 \cdots\rangle$
  | $\forall x : A.P$     $\mapsto$   do pf $\leftarrow$ tauto $(\phi, x : A)$ $P$;
                          $\langle \cdots$ proof of $\forall x : A.P \cdots\rangle$
  | $\cdots$
```

Here the proof object pf returned by the recursive call on the body of the quantifier will depend on an extra free variable; this dependence is to be discharged inside the proof object for $\forall x : A.P$.

Let us now consider the case of handling a propositional implication like $P_1 \rightarrow P_2$. In this case we would like to keep information about P_1 being a hypothesis, so as to use it as a fact if it is later encountered inside P_2 (e.g. to prove a tautology like $P \rightarrow P$). The way we can encode this in our language is to have our tauto procedure carry an extra list of hypotheses. The default case of pattern matching can be changed so that this list is searched instead of returning None as we did above. Each element in the hypothesis list should carry a proof object of the hypothesis too, so that we can return it if the hypothesis matches the goal. Thus each element of the hypothesis list must be an existential package, bundling together the proposition and the proof object of each hypothesis:

$$\mathsf{hyplist} = \lambda\phi : \mathsf{ctx}.\mathsf{list}\ (\Sigma H : [\phi]\,\mathsf{Prop}.\mathsf{LT}([\phi]\,H))$$

The type list is the ML list type; we assume that the standard map and fold_left functions are available for it.

For this data structure, we define the function hyplistWeaken which lifts a hypotheses list from one context to an extended one; and the function hyplistFind which given a proposition P and a hypotheses list, goes through the list trying to find whether the proposition P is included, returning the associated proof object if it is. For hyplistWeaken we only give its desired type; we will see its full definition in Section 5, after we have introduced some details about the logic and the computational language.

```
hyplistWeaken : $\Pi\phi$ : ctx.$\Pi A$ : $[\phi]$ Prop.hyplist $\phi \rightarrow$ hyplist $(\phi, x : A)$

  hypMatch : $\Pi\phi$ : ctx.$\Pi P$ : $[\phi]$ Prop.$(\Sigma P'$ : $[\phi]$ Prop.LT$([\phi]\,P'))$ $\rightarrow$
                                                   option LT$([\phi]\,P)$
  hypMatch $\phi$ $P$ hyp  =   let $\langle P'$, pf'$\rangle$ = hyp in
                        holcase $P$ of
                           $P'$    $\mapsto$    Some pf'
                         | _     $\mapsto$    None

  hyplistFind : $\Pi\phi$ : ctx.$\Pi P$ : $[\phi]$ Prop.hyplist $\phi \rightarrow$ option LT$([\phi]\,P)$
  hyplistFind $\phi$ $P$ hl  =
     fold_left ($\lambda$res.$\lambda$hyp.res || hypMatch $P$ hyp) None hl
```

Note that we use the notation $\langle\cdot,\,\cdot\rangle$ as the introduction form for existential packages, and let $\langle\cdot,\,\cdot\rangle = \cdot$ in $\cdots$ as the elimination form.

The tauto tactic should itself be modified as follows. When trying to prove $P_1 \rightarrow P_2$, we want to add P_1 as a hypothesis to the current hypotheses list hl. In order to provide a proof object for this hypothesis, we introduce a new free variable pf_1 representing the proof of P_1 and try to prove P_2 recursively in this extended context using the extended hypotheses list. Note that we need to lift all terms in the hypotheses list to the new context before being able to use it; this is what hyplistWeaken is used for. The proof object returned for P_2 might mention the new free variable pf_1. This extra dependence is discharged using the implication introduction axiom of the underlying logic, yielding a proof of $P_1 \rightarrow P_2$. Details are shown below.

```
tauto : $\Pi\phi$ : ctx.$\Pi P$ : $[\phi]$ Prop.hyplist $\phi \rightarrow$ option LT$([\phi]\,P)$
tauto $\phi$ $P$ hl = holcase $P$ of
     ...
   | $P_1 \rightarrow P_2 \mapsto$ let hl' = hyplistWeaken $\phi$ $P_1$ hl in
                   let hl'' = cons $\langle P_1, \langle[\phi, \mathrm{pf}_1 : P_1]\,\mathrm{pf}_1\rangle\rangle$ hl' in
                   do x $\leftarrow$ tauto $(\phi, \mathrm{pf}_1 : P_1)$ $P_2$ hl'';
                      $\langle \cdots$ proof of $P_1 \rightarrow P_2 \cdots\rangle$
   | _ $\mapsto$ hyplistFind $\phi$ $P$ hl
```

3.3 General recursion

We have extended this procedure in order to deconstruct the hypothesis before entering it in the hypothesis list (e.g. entering two different hypotheses for $P_1 \wedge P_2$ instead of just one, etc.), but this extension does not give us any new insights with respect to the use of our language so we do not show it here.

A more interesting modification we have done is to extend the procedure that searches the hypothesis list for the current goal, so that when trying to prove the goal G, a hypothesis like $H' \rightarrow G$ can be used, making H' a new goal. This is easy to achieve: we can have the hyplistFind procedure used above be mutually recursive with tauto, and have it pattern-match on the hypotheses, calling tauto recursively for the newly generated goals. Still, we need to be careful in order to avoid recursive cycles. A naive implementation would be thrown into an endless loop if a proof for a proposition like $(A \rightarrow B) \rightarrow (B \rightarrow A) \rightarrow A$ was attempted.

The way to solve this is to have the two procedures maintain a list of "already visited" goals, so that we avoid entering a cycle. Using the techniques we have seen so far, this is easy to encode in our language. This extension complicates the termination argument for our tactic substantially, but since we are working in a computational language allowing non-termination, we do not need to formalize this argument. This is a point of departure compared to an implementation of this tactic based on proof-by-reflection, e.g. similar to what is described in Chlipala [2008]. In that case, the most essential parts of the code we are describing would be written inside the computational language embedded inside the logic, and as such would need to be provably terminating. The complicated termination argument required would make the effort required for this extension substantial. Compared to programming a similar tactic in a language like ML (following the traditional LCF approach), in our implementation the partial correctness of the tactic is established statically. This is something that would otherwise be only achievable by using a proof-by-reflection based implementation.

3.4 Imperative features

The second example that we will consider is a decision procedure that handles the theory of equality, one of the basic theories that SMT solvers support. This is the theory generated from the axioms:

$$\forall x.x = x$$
$$\forall x, y.x = y \rightarrow y = x$$
$$\forall x, y, z.x = y \rightarrow y = z \rightarrow x = z$$

In a logic like the one we are using, the standard definition of equality includes a single constructor for the reflexivity axiom; the other axioms above can then be proved as theorems using inductive elimination. We will see how this is done in the next section.

Usually this theory is extended with the axiom:

$$\forall x, y.x = y \rightarrow f\ x = f\ y$$

which yields the theory of equality with uninterpreted functions (EUF). We have implemented this extension but we will not consider it here because it does not add any new insights. To simplify our presentation, we will also assume that all terms taking part in equations are of a fixed type A.

We want to create a decision procedure that gets a list of equations as hypotheses, and then prove whether two terms are equal or not, according to the above axioms and based on the given equations. The standard way to write such a decision procedure is to use a union-find data structure to compute the equivalence classes of terms, based on the given equations. We will use a simple algorithm, described in Bradley and Manna [2007], which still requires imperative features in order to be implemented efficiently.

Terms are assigned nodes in a tree-like data structure, which gets usually implemented as an array. Each equivalence class has one representative; each node representing a term has a pointer to a parent term, which is another member of its equivalence class; if a term's node points to itself, then it is the representative of its class. We can thus find the representative of the equivalence class where a term belongs by successively following pointers, and we can merge two equivalence classes by making the representative of one class point to the representative of the other.

We want to stay as close as possible to this algorithm, yet have our procedure yield proof objects for the claimed equations. We choose to encode the union-find data structure as a hash table; this table will map each term into a (mutable) value representing its parent term. Since we also want to yield proofs, we need to also store information on how the two terms are equal. We can encode such a hash table using the following type (assuming that terms inhabit a context ϕ):

$$\mathsf{eqhash} = \mathsf{array}\ (\Sigma X : [\phi]A.\Sigma X' : [\phi]A.\mathsf{LT}([\phi]X = X'))$$

We can read the type of elements in the array as key-value pairs, where the key is the first term of type A, and the value is the existential package of its parent along with an appropriately typed proof object.

Implementing such a hash-table structure is straightforward, provided that there exists an appropriate construct in our computational language to compute a hash value for a logical term. We can have dependent types for the get/set functions as follows:

$$\begin{array}{l}\mathsf{eqhashGet} : \Pi X : [\phi]A.\mathsf{eqhash} \to \mathsf{option}\ (\Sigma X' : [\phi]A.\mathsf{LT}([\phi]X = X'))\\ \mathsf{eqhashSet} : \Pi X : [\phi]A.\Pi X' : [\phi]A.\Pi\mathsf{pf} : [\phi]X = X'.\mathsf{eqhash} \to \mathsf{unit}\end{array}$$

The find operation for the union-find data structure can now be simply implemented using the following code. Given a term, we need to return the representative of its equivalence class, along with a proof of equality of the two terms. We look up a given term in the hash table, and keep following links to parents until we end up in a term that links to itself, building up the equality proof as we go; if the term does not exist in the hash table, we simply add it.

```
find : $\Pi X : ([\phi]A).\mathsf{eqhash} \to \Sigma X' : ([\phi]A).\mathsf{LT}([\phi]X = X')$
find $X$ h =
    (do x ← eqhashGet $X$ h;
        let $\langle X', \mathsf{pf} :: X = X'\rangle$ = x in
        holcase $X$ of
            $X'$    ↦    $\langle X', \mathsf{pf}\rangle$
          | _       ↦    let $\langle X'', \mathsf{pf'} :: X' = X''\rangle$ = find $X'$ h in
                         $\langle X'', \langle \cdots\ \textit{proof of}\ X = X''\ \cdots\rangle\rangle$)
    || (let self = $\langle X, \cdots\ \textit{proof of}\ X = X\ \cdots\rangle$ in
        (eqhashSet $X$ self); self)
```

The union operation is given two terms along with a proof that they are equal, and updates the hash-table accordingly: it uses find to get the representatives of the two terms, and if they do not match, it updates the one representative to point to the other.

```
union : $\Pi X : [\phi]A.\Pi X' : [\phi]A.\Pi\mathsf{pf} : [\phi]X = X'.\mathsf{eqhash} \to \mathsf{unit}$
union $X$ $X'$ pf h =
    let $\langle X_{rep}, \mathsf{pf}_1 :: X = X_{rep}\rangle$ = find $X$ h in
    let $\langle X'_{rep}, \mathsf{pf}_2 :: X' = X'_{rep}\rangle$ = find $X'$ h in
    holcase $X_{rep}$ of
      $X'_{rep}$ ↦ ()
    | _      ↦ let upd = $\langle X'_{rep}, \langle \cdots\ \textit{proof of}\ X_{rep} = X'_{rep}\ \cdots\rangle\rangle$ in
               eqhashSet $X_{rep}$ upd
```

A last function is needed, which will be used to check whether in the current hash table, two terms are equal or not. Its type will be:

$$\begin{array}{r}\mathsf{areEqual?} : \Pi X : ([\phi]A).\Pi X' : ([\phi]A).\mathsf{eqhash} \to\\ \mathsf{option}\ \mathsf{LT}([\phi]X = X')\end{array}$$

Its implementation is very similar to the above function.

In the implementation we have seen above, we have used an imperative data-structure with a dependent data type, that imposes an algorithm-specific invariant. Because of this, rich type information is available while developing the procedure, and the type restrictions impose a principled programming style. At the same time, a multitude of bugs that could occur in an ML-based implementation are avoided: at each point where a proof object is explicitly given in the above implementation, we know that it proves the expected proposition, while in an ML-based implementation, no such guarantee is given. Still, adapting the standard implementation of the algorithm to our language is relatively straightforward, and we do not need to use fundamentally different data structures, as we would need to do if we were developing this inside the computational language of a logic (since only functional data structures could be used).

4. The logic language $\lambda\mathrm{HOL}^{\mathrm{ind}}$

We will now focus on the formal logic that we are using. We use a higher-order logic with support for inductive definitions of datatypes, predicates and logical connectives; such inductive definitions give rise to inductive elimination axioms. Also, total recursive functions can be defined, and terms of the logic are identified up to evaluation of these functions. Our logical framework also consists of explicit proof objects, which can be viewed as witnesses of derivations in the logic.

This framework, which we call $\lambda\mathrm{HOL}^{\mathrm{ind}}$, is based on λHOL as presented in Barendregt and Geuvers [1999], extended with inductive definitions and a reduction relation for total recursive functions, in the style of CIC [Barras et al. 2010]. Alternatively, we can view this framework as a subset of CIC, where we have omitted universes other than Prop and Type, as well as polymorphism and dependent types in Type. Logical consistency of CIC [Werner 1994] therefore implies logical consistency of our system. Still, a simpler metatheory based on reducibility candidates is possible.

We view this logical framework as a common core between proof assistants like Coq and the HOL family, that is still expressible enough for many applications. At the same time, we believe that it captures most of the complexities of their logics (e.g. the notion of computation in CIC), so that the results that we have for this framework can directly be extended to them.

The syntax of our framework is presented in Figure 2. The syntactic category d includes propositions (which we denote as P) and predicates, as well as objects of our domain of discourse: terms of inductively defined data types, as well as total functions between them. Inductive definitions come from a definitions environment Δ; total functions are defined by primitive recursion (using the $\mathsf{Elim}(\cdot,\cdot)$ construct). Terms of this category get assigned kinds of the syntactic category $\mathcal{K}$, with all propositions being assigned kind Prop. Inductive datatypes are defined at this level of kinds. We

$$\begin{array}{rrl}
\text{(sorts)} & s & ::= \mathsf{Type} \mid \mathsf{Type}' \\
\text{(kinds)} & \mathcal{K} & ::= \mathsf{Prop} \mid c_{\mathcal{K}} \mid \mathcal{K}_1 \to \mathcal{K}_2 \mid x \\
\text{(domain obj./props.)} & d, P & ::= d_1 \to d_2 \mid \forall x : \mathcal{K}.d \mid \lambda x : \mathcal{K}.d \\
 & & \quad \mid d_1\ d_2 \mid c_d \mid x \mid \mathsf{Elim}(c_{\mathcal{K}}, \mathcal{K}') \\
\text{(proof objects)} & \pi & ::= x \mid \lambda x : P.\pi \mid \pi_1\ \pi_2 \mid \lambda x : \mathcal{K}.\pi \\
 & & \quad \mid \pi\ d \mid c_\pi \mid \mathsf{elim}\ c_{\mathcal{K}} \mid \mathsf{elim}\ c_d \\
\text{(HOL terms)} & t & ::= s \mid \mathcal{K} \mid d \mid \pi \\
\text{(logic variables env.)} & \Phi & ::= \bullet \mid \Phi,\ x : t \\
\text{(definitions env.)} & \Delta & ::= \bullet
\end{array}$$

$$\mid \Delta,\ \mathsf{Inductive}\ c_{\mathcal{K}} : \mathsf{Type} := \{\overrightarrow{c_d : \mathcal{K}}\}$$

$$\mid \Delta,\ \mathsf{Inductive}\ c_t(\overrightarrow{x : \mathcal{K}}) : \mathcal{K}_1 \to \cdots \to \mathcal{K}_n \to \mathsf{Prop} := \{\overrightarrow{c_\pi : P}\}$$

Figure 2. Syntax of the base logic language $\lambda\text{HOL}^{\text{ind}}$

Typing for domain objects, propositions and predicates:

$$\frac{x : \mathcal{K} \in \Phi}{\Phi \vdash x : \mathcal{K}}\ \textsc{dp-Var} \qquad \frac{\Phi \vdash P_1 : \mathsf{Prop} \qquad \Phi \vdash P_2 : \mathsf{Prop}}{\Phi \vdash P_1 \to P_2 : \mathsf{Prop}}\ \textsc{dp-Impl}$$

$$\frac{\Phi, x : \mathcal{K} \vdash P : \mathsf{Prop}}{\Phi \vdash \forall x : \mathcal{K}.P : \mathsf{Prop}}\ \textsc{dp-Forall} \qquad \frac{\Phi, x : \mathcal{K} \vdash d : \mathcal{K}'}{\Phi \vdash \lambda x : \mathcal{K}.d : \mathcal{K} \to \mathcal{K}'}\ \textsc{dp-Lam}$$

$$\frac{\Phi \vdash d_1 : \mathcal{K} \to \mathcal{K}' \qquad \Phi \vdash d_2 : \mathcal{K}}{\Phi \vdash d_1\ d_2 : \mathcal{K}'}\ \textsc{dp-App}$$

Typing for proof objects:

$$\frac{x : P \in \Phi}{\Phi \vdash x : P}\ \textsc{po-Var} \qquad \frac{\Phi,\ x : P \vdash \pi : P' \qquad \Phi \vdash P \to P' : \mathsf{Prop}}{\Phi \vdash \lambda x : P.\pi : P \to P'}\ \textsc{po-impI}$$

$$\frac{\Phi \vdash \pi_1 : P \to P' \qquad \Phi \vdash \pi_2 : P}{\Phi \vdash \pi_1\ \pi_2 : P'}\ \textsc{po-impE}$$

$$\frac{\Phi,\ x : \mathcal{K} \vdash \pi : P' \qquad \Phi \vdash \forall x : \mathcal{K}.P' : \mathsf{Prop}}{\Phi \vdash \lambda x : \mathcal{K}.\pi : \forall x : \mathcal{K}.P'}\ \textsc{po-forallI}$$

$$\frac{\Phi \vdash \pi : \forall x : \mathcal{K}.P' \qquad \Phi \vdash d : \mathcal{K}}{\Phi \vdash \pi\ d : P'[d/x]}\ \textsc{po-forallE}$$

$$\frac{\Phi \vdash \pi : P \qquad P =_{\beta\iota} P'}{\Phi \vdash \pi : P'}\ \textsc{po-Convert}$$

Figure 3. Main typing judgements of $\lambda\text{HOL}^{\text{ind}}$ (selected rules)

can view Prop as a distinguished datatype, whose terms can get extended through inductive definitions of predicates and logical connectives. Kinds get assigned the sort Type, which in turn gets assigned the (external) sort Type$'$, so that contexts can include variables over Type.

The last syntactic category of our framework is π, representing proof objects, which get assigned a proposition as a type. We can think of terms at this level as corresponding to different axioms of our logic, e.g. function application will witness the implication elimination rule (modus-ponens). We include terms for performing proof by induction on inductive datatypes and inductive predicates. Using the syntactic category t we represent terms at any level out of the ones we described; at the level of variables we do not distinguish between these different levels.

To see how inductive definitions are used, let us consider the case of natural numbers. Their definition would be as follows:

$$\begin{array}{l}
\mathsf{Inductive\ Nat : Type :=} \\
\qquad \mathsf{zero : Nat} \\
\mid \qquad \mathsf{succ : Nat \to Nat.}
\end{array}$$

This gives rise to the Nat kind, the zero and succ constructors at the domain objects level, and the elimination axiom elim Nat at the proof object level, that witnesses induction over natural numbers, having the following type:

$$\forall P : \mathsf{Nat} \to \mathsf{Prop}.P\ \mathsf{zero} \to (\forall x : \mathsf{Nat}.P\ x \to P\ (\mathsf{succ}\ x)) \to \forall x : \mathsf{Nat}.P\ x$$

Similarly we can define predicates, like equality of natural numbers, or logical connectives, through inductive definitions at the level of propositions:

$$\begin{array}{l}
\mathsf{Inductive}\ (=_{\mathsf{Nat}})\ (x : \mathsf{Nat}) : \mathsf{Nat} \to \mathsf{Prop} := \\
\quad \mathsf{refl} : x =_{\mathsf{Nat}} x.
\end{array}$$

$$\begin{array}{l}
\mathsf{Inductive}\ (\wedge)\ (A\ B : \mathsf{Prop}) : \mathsf{Prop} := \\
\quad \mathsf{conj} : A \to B \to A \wedge B.
\end{array}$$

From the definition of $=_{\mathsf{Nat}}$ we get Leibniz equality as the elimination principle, from which the axioms mentioned in the previous section are easy to prove.

$$\mathsf{elim}\ (=_{\mathsf{Nat}}) : \forall x : \mathsf{Nat}.\forall P : \mathsf{Nat} \to \mathsf{Prop}.P\ x \to \forall y : \mathsf{Nat}.x =_{\mathsf{Nat}} y \to P\ y$$

Last, recursive functions over natural numbers can also be defined through the $\mathsf{Elim}(\mathsf{Nat}, \mathcal{K})$ construct: $\mathsf{Elim}(\mathsf{Nat}, \mathcal{K})\ n\ f_z\ f_s$ proceeds by performing primitive recursion on the natural number n given $f_z : \mathcal{K}$ and $f_s : \mathsf{Nat} \to \mathcal{K} \to \mathcal{K}$, returning a term of kind $\mathcal{K}$. For example, we can define the addition function for Nat as:

$$\mathsf{plus}\ =\ \lambda x, y : \mathsf{Nat}.\mathsf{Elim}(\mathsf{Nat}, \mathsf{Nat})\ x\ y\ (\lambda x', r_{x'} : \mathsf{Nat}.\mathsf{succ}\ r_{x'})$$

Functions defined through primitive recursion are permitted to return propositions (where $\mathcal{K} = \mathsf{Prop}$), something that is crucial in order to prove theorems like $\forall x : \mathsf{Nat}.\mathsf{zero} \neq \mathsf{succ}\ x$.

We present the main typing judgements of $\lambda\text{HOL}^{\text{ind}}$ in Fig 3. These judgements use the logic variables environment Φ. To simplify the presentation, we assume that the definitions environment Δ is fixed and we therefore do not explicitly include it in our judgements. We have not included its well-formedness rules; these should include the standard checks for positivity of inductive definitions and are defined following CIC (see for example [Paulin-Mohring 1993]). Similarly we have omitted the typing rules for the elim constructs. We can view this as a standard PTS with sorts $\mathcal{S} = \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$, axioms $\mathcal{A} = \{(\mathsf{Prop}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Type}')\}$ and rules $\mathcal{R} = \{(\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Type}, \mathsf{Type})\}$, extended with inductive definitions and elimination at the levels we described earlier. In later sections we follow this "collapsed" view, using the single typing judgement $\Phi \vdash t : t'$ for terms of all levels.

Of interest is the PO-CONVERT typing rule for proof objects. We define a limited notion of computation within the logic language, composed by the standard β-reduction for normal β-redeces and by an additional ι-reduction (defined as in CIC), which performs case reduction and evaluation of recursive function applications. With this rule, logical terms (propositions, terms of inductive datatypes, etc.) that are $\beta\iota$-equivalent are effectively identified for type checking purposes. Thus a proof object for the proposition $2 =_{\mathsf{Nat}} 2$ can also be seen as a proof object for the proposition $1 + 1 =_{\mathsf{Nat}} 2$, since both propositions are equivalent if they are evaluated to normal forms. Because of this particular feature of having a notion of computation within the logic language, $\lambda\text{HOL}^{\text{ind}}$ follows the Poincaré principle, which we view as one of the points of departure of CIC compared to HOL. We have included this in our logic to show that a computational language as the one we propose in the next section is still possible for such a framework.

4.1 Extension with meta-variables

As we have mentioned in the previous section, our computational language will manipulate logical terms living in different contexts.

$$\begin{array}{rrcl}
\text{(contextual terms)} & T & ::= & [\Phi]\,t \\
\text{(meta-variables env.)} & \mathcal{M} & ::= & \bullet \mid \mathcal{M},\, X : T \\
\text{(substitution)} & \sigma & ::= & \bullet \mid \sigma,\, t \\
 & \mathcal{K} & ::= & \cdots \mid X/\sigma \\
 & d, P & ::= & \cdots \mid X/\sigma \\
 & \pi & ::= & \cdots \mid X/\sigma
\end{array}$$

Figure 4. Syntax extension of $\lambda\text{HOL}^{\text{ind}}$ with contextual terms and meta-variables

$$\begin{array}{rrcl}
\text{(context env.)} & \mathcal{W} & ::= & \bullet \mid \mathcal{W},\, \phi : \mathsf{ctx} \\
 & \Phi & ::= & \cdots \mid \Phi,\, \phi \\
 & \sigma & ::= & \cdots \mid \sigma,\, \mathsf{id}_\phi
\end{array}$$

Figure 5. Syntax extension of $\lambda\text{HOL}^{\text{ind}}$ with parametric contexts

In order to be able to type-check these terms properly, we introduce a new class of terms T called contextual terms, which package a logical term along with its free variables environment. We write a contextual term as $[\Phi]\,t$ where t can mention variables out of Φ. We identify these terms up to alpha equivalence (that is, renaming of variables in Φ and t).

Furthermore, we need a notion of contextual variables or meta-variables. Our computational language will provide ways to abstract over such variables, which stand for contextual terms T. We denote meta-variables as X and use capital letters for them. To use a meta-variable X inside a logical term t, we need to make sure that when it gets substituted with a contextual term $T = [\Phi']\,t'$, the resulting term $t[T/X]$ will still be properly typed. Since t' refers to different free variables compared to t, we need a way to map them into terms that only refer to the same variables as t. This mapping is provided by giving an explicit substitution when using the variable X. The syntax of our logic is extended accordingly, as shown in Figure 4.

Since logical terms t now include meta-variables, we need to refer to an additional meta-context $\mathcal{M}$. Thus the main typing judgement of the base logic, $\Phi \vdash t : t'$ is extended to include this new environment, resulting in a typing judgement of the form $\mathcal{M};\, \Phi \vdash t : t'$. Existing typing rules ignore the extra $\mathcal{M}$ environment; what is interesting is the rule for the use of a meta-variable. This is as follows:

$$\frac{X : T \in \mathcal{M} \qquad T = [\Phi']\,t \qquad \mathcal{M};\, \Phi \vdash \sigma : \Phi'}{\mathcal{M};\, \Phi \vdash X/\sigma : t[\sigma/\Phi']}$$

We use the typing judgement $\mathcal{M};\, \Phi \vdash \sigma : \Phi'$ to check that the provided explicit substitution provides a term of the appropriate type under the current free variable context, for each one of the free variables in the context associated with the meta-variable X. The judgment $\mathcal{M};\, \Phi \vdash \sigma : \Phi'$ is defined below.

$$\frac{}{\mathcal{M};\, \Phi \vdash \bullet : \bullet} \qquad \frac{\mathcal{M};\, \Phi \vdash \sigma : \Phi' \quad \mathcal{M};\, \Phi \vdash t : t'[\sigma/\Phi']}{\mathcal{M};\, \Phi \vdash (\sigma,\, t) : (\Phi',\, x : t')}$$

A little care is needed since there might be dependencies between the types of the elements of the context. Thus when type-checking a substitution against a context, we might need to apply part of the substitution in order to get the type of another element in the context. This is done by the simultaneous substitution $[\sigma/\Phi]$ of variables in Φ by terms in σ. To simplify this procedure, we treat the context Φ and the substitution σ as ordered lists that adhere to the same variable order.

$\mathcal{W};\, \mathcal{M};\, \Phi \vdash t : t'$	Typing for logical terms
$\mathcal{W};\, \mathcal{M} \vdash T : T'$	Typing for contextual terms
$\mathcal{W};\, \mathcal{M};\, \Phi \vdash \sigma : \Phi'$	Typing for substitutions
$\mathcal{W};\, \mathcal{M} \vdash \Phi$	Well-formedness for logical variables contexts

Figure 6. Summary of extended $\lambda\text{HOL}^{\text{ind}}$ typing judgements

To type a contextual term $T = [\Phi]\,t$, we use the normal typing judgement for our logic to type the packaged term t under the free variables context Φ. The resulting type will be another contextual term T' associated with the same free variables context. Thus the only information that is needed in order to type-check a contextual term T is the meta-context $\mathcal{M}$. The judgement $\mathcal{M} \vdash T : T'$ for typing contextual terms will therefore look as follows:

$$\frac{T = [\Phi]\,t \qquad \mathcal{M} \vdash \Phi \qquad \mathcal{M};\Phi \vdash t : t'}{\mathcal{M} \vdash T : [\Phi]\,t'}$$

We use the judgement $\mathcal{M} \vdash \Phi$ to make sure that Φ is a well-formed context, i.e. that all the variables defined in it have a valid type; dependencies between them are allowed.

$$\frac{}{\mathcal{M} \vdash \bullet} \qquad \frac{\mathcal{M} \vdash \Phi \qquad \mathcal{M};\, \Phi \vdash t : t'}{\mathcal{M} \vdash \Phi,\, x : t}$$

Last, let us consider how to apply the substitution $[T/X]$ (where $T = [\Phi]\,t$) on X inside a logical term t'. In most cases the substitution is simply recursively applied to the subterms of t'. The only special case is when $t' = X/\sigma$. In this case, the meta-variable X should be substituted by the logical term t; its free variables Φ are mapped to terms meaningful in the same context as the original term t' using the substitution σ. Thus:

$$(X/\sigma)[T/X] = t[\sigma/\Phi],\ \text{when}\ T = [\Phi]\,t$$

For example, consider the case where $X : [a : \mathsf{Nat}, b : \mathsf{Nat}]\,\mathsf{Nat}$, $t' = \mathsf{plus}\ (X/1,2)\ 0$ and $T = [a : \mathsf{Nat}, b : \mathsf{Nat}]\,\mathsf{plus}\ a\ b$. We have:

$$t'[T/X] = \mathsf{plus}\ ((\mathsf{plus}\ a\ b)[1/a, 2/b])\ 0 = \mathsf{plus}\ (\mathsf{plus}\ 1\ 2)\ 0 =_{\beta\iota} 3$$

The above rule is not complete: the substitution σ is still permitted to use X based on our typing rules, and thus we have to re-apply the substitution of T for X in σ. Note that no circularity is involved, since at some point a substitution σ associated with X will need to not refer to it – the term otherwise would have infinite depth. Thus the correct rule is:

$$(X/\sigma)[T/X] = t[(\sigma[T/X])/\Phi],\ \text{when}\ T = [\Phi]\,t$$

4.2 Extension with parametric contexts

As we saw from the programming examples in the previous section, it is also useful in our computational language to be able to specify that a contextual term depends on a parametric context. Towards that effect, we extend the syntax of our logic in Figure 5, introducing a notion of context variables, denoted as ϕ, which stand for an arbitrary free variables context Φ. These context variables are defined in the environment $\mathcal{W}$. The definition of the logical variables context Φ is extended so that context variables can be part of it; thus Φ contexts become parametric. In essence, a context variable ϕ inside a context Φ serves as a placeholder, where more free variables can be substituted; this is permitted because of weakening.

We extend the typing judgements we have seen so that the $\mathcal{W}$ environment is also included; a summary of this final form of the judgements is given in Figure 6. The typing judgement that checks well-formedness of a context Φ is extended so that context variables defined in the context $\mathcal{W}$ are permitted to be part of it:

$$\frac{\mathcal{W};\ \mathcal{M} \vdash \Phi \qquad \phi : \mathsf{ctx} \in \mathcal{W}}{\mathcal{W};\ \mathcal{M} \vdash \Phi,\ \phi}$$

With this change, meta-variables X and contextual terms T can refer to a parametric context Φ, by including a context variable ϕ at some point. Explicit substitutions σ associated with use of meta-variables must also be extended so that they can correspond to such parametric contexts; this is done by introducing the identity substitution id_ϕ for each context variable ϕ. The typing rule for checking a substitution σ against a context Φ is extended accordingly:

$$\frac{\mathcal{W};\ \mathcal{M};\ \Phi \vdash \sigma : \Phi' \qquad \phi \in \Phi}{\mathcal{W};\ \mathcal{M};\ \Phi \vdash (\sigma,\ \mathsf{id}_\phi) : (\Phi',\ \phi)}$$

When substituting a context Φ for a context variable ϕ inside a logical term t, this substitution gets propagated inside the subterms of t. Again the interesting case is what happens when t corresponds to a use of a meta-variable ($t = X/\sigma$). In that case, we need to replace the identity substitution id_ϕ in the explicit substitution σ by the actual identity substitution for the context Φ. This is done using the $\mathsf{idsubst}(\cdot)$ function:

$$\mathsf{id}_\phi[\Phi/\phi] = \mathsf{idsubst}(\Phi)$$

$$\text{where:}\quad \begin{array}{lcl} \mathsf{idsubst}(\bullet) & = & \bullet \\ \mathsf{idsubst}(\Phi,\ x : t) & = & \mathsf{idsubst}(\Phi),\ x \\ \mathsf{idsubst}(\Phi,\ \phi) & = & \mathsf{idsubst}(\Phi),\ \mathsf{id}_\phi \end{array}$$

With the above in mind, it is easy to see how a proof object for $P_1 \to P_2$ living in context ϕ can be created, when all we have is a proof object X for P_2 living in context $\phi,\ \mathrm{pf}_1 : P_1$.

$$[\phi]\, \lambda y : P_1.(X/(\mathsf{id}_\phi,\ y))$$

This can be used in the associated case in the tautology prover example given earlier, filling in as the term $\langle \cdots$ *proof of* $P_1 \to P_2 \cdots \rangle$.

4.3 Metatheory

We have proved that substitution of a contextual term T for a meta-variable X and the substitution of a context Φ for a context variable ϕ, preserve the typing of logical terms t. The statements of these substitution lemmas are:

Lemma 4.1 *If* $\mathcal{M},\ X_0 : T,\ \mathcal{M}';\ \Phi \vdash t : t'$ *and* $\mathcal{M} \vdash T_0 : T$, *then* $\mathcal{M},\ \mathcal{M}'[T_0/X_0];\ \Phi[T_0/X_0] \vdash t[T_0/X_0] : t'[T_0/X_0]$.

Lemma 4.2 *If* $\mathcal{M},\ \mathcal{M}';\ \Phi,\ \phi_0,\ \Phi' \vdash t : t'$ *and* $\mathcal{W};\mathcal{M};\Phi \vdash \Phi_0$, *then* $\mathcal{M},\ \mathcal{M}'[\Phi_0/\phi_0];\ \Phi,\ \Phi_0,\ \Phi' \vdash t[\Phi_0/\phi_0] : t'[\Phi_0/\phi_0]$.

These are proved by straightforward mutual structural induction, along with similar lemmas for explicit substitutions σ, contextual terms T and contexts Φ, because of the inter-dependencies between them. The proofs only depend on a few lemmas for the core of the logic that we have described, namely the standard simultaneous substitution lemma, weakening lemma, and preservation of $\beta\iota$-equality under simultaneous substitution. Details are provided in the extended version of this paper [Stampoulis and Shao 2010].

These extensions are inspired by contextual modal type theory and the Beluga framework; here we show how they can be adapted to a different logical framework, like the one we have described. Compared to Beluga, one of the main differences is that we do not support first-class substitutions, because so far we have not found abstraction over substitutions in our computational language to be necessary. Also, context variables are totally generic, not constrained by a context schema. The reason for this is that we will not use our computational language as a proof meta-language, so coverage and totality of definitions in it is not needed; thus context schemata are not necessary too. Last, our Φ contexts are ordered and therefore permit multiple context variables ϕ in them; this is mostly presentational.

$$\begin{array}{l} K ::= * \mid K_1 \to K_2 \\ \tau ::= \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha : K.\tau \\ \quad \mid \forall\alpha : K.\tau \mid \alpha \mid \mathsf{array}\,\tau \mid \lambda\alpha : K.\tau \mid \tau_1\ \tau_2 \mid \cdots \\ e ::= () \mid n \mid e_1 + e_2 \mid e_1 \le e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else} e_2 \\ \quad \mid \lambda\mathbf{x} : \tau.e \mid e_1\ e_2 \mid (e_1,\ e_2) \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e \\ \quad \mid \mathsf{case}(e,\ \mathbf{x}_1.e_1,\ \mathbf{x}_2.e_2) \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \Lambda\alpha : K.e \mid e\ \tau \\ \quad \mid \mathsf{fix}\ \mathbf{x} : \tau.e \mid \mathsf{mkarray}(e, e') \mid e[e'] \mid e[e'] := e'' \mid l \mid \mathsf{error} \mid \cdots \\ \Gamma ::= \bullet \mid \Gamma,\ \mathbf{x} : \tau \mid \Gamma,\ \alpha : K \\ \Sigma ::= \bullet \mid \Sigma,\ l : \mathsf{array}\ \tau \end{array}$$

Figure 7. Syntax for the computational language (ML fragment)

$$\begin{array}{lll} K & ::= & \cdots \mid \Pi x : T.K \mid \Pi\phi : \mathsf{ctx}.K \\ \tau & ::= & \cdots \mid \Pi X : T.\tau \mid \Sigma X : T.\tau \mid \Pi\phi : \mathsf{ctx}.\tau \mid \Sigma\phi : \mathsf{ctx}.\tau \\ & & \mid \lambda X : T.\tau \mid \tau\ T \mid \lambda\phi : \mathsf{ctx}.\tau \mid \tau\ \Phi \\ e & ::= & \cdots \mid \lambda X : T.e \mid e\ T \mid \langle T,\ e \rangle \mid \mathsf{let}\ \langle X,\ \mathbf{x} \rangle = e\ \mathsf{in}\ e' \\ & & \mid \lambda\phi : \mathsf{ctx}.e \mid e\ \Phi \mid \langle \Phi,\ e \rangle \mid \mathsf{let}\ \langle \phi,\ \mathbf{x} \rangle = e\ \mathsf{in}\ e' \\ & & \mid \mathsf{holcase}\ T\ \mathsf{of}\ (p_1 \mapsto e_1) \cdots (p_n \mapsto e_n) \\ p & ::= & c_d \mid p_1 \to p_2 \mid \forall x : p_1.p_2 \mid \lambda x : p_1.p_2 \mid p_1\ p_2 \mid x \mid X/\sigma \\ & & \mid \mathsf{Elim}(c_{\mathcal{K}}, \mathcal{K}') \mid c_{\mathcal{K}} \mid \mathsf{Prop} \end{array}$$

Figure 8. Syntax for the computational language (new constructs)

5. The computational language

Having described the logical framework we are using, we are ready to describe the details of our computational language. The ML fragment that we support is shown in Figure 7 and consists of algebraic datatypes, higher-order function types, the native integer and boolean datatypes, mutable arrays, as well as polymorphism over types. For presentation purposes, we regard mutable references as one-element arrays. We use bold face for variables $\mathbf{x}$ of the computational language in order to differentiate them from logical variables. In general we assume that we are given full typing derivations for well-typed terms; issues of type reconstruction are left as future work.

The syntax for the new kinds, types and expressions of this language is given in Figure 8, while the associated typing rules and small-step operational semantics are given in Figures 9 and 10 respectively. Other than the pattern matching construct, typing and operational semantics for the other constructs are entirely standard. We will describe them briefly, along with examples of their use.

Functions and existentials over contextual terms Abstraction over a contextual logical term ($\lambda X : T.e$) results in a dependent function type ($\Pi X : T.\tau$), assigning a variable name to this term so that additional arguments or results of the function can be related to it. Still, because of the existence of the pattern matching construct, such logical terms are runtime entities. We should therefore not view this abstraction as similar to abstraction over types or type indexes in other dependently typed programming languages; rather, it is a construct that gets preserved at runtime. Similarly, existential packages over contextual terms are also more akin to normal tuples.

To lift an arbitrary HOL term to the computational language, we can use an existential package where the second member is of unit type. This operation is very common, so we introduce the following syntactic sugar at the type level, and for the introduction

$$\frac{\mathcal{W};\mathcal{M}\vdash T : T' \qquad \mathcal{W};\mathcal{M},\ X : T;\Sigma;\Gamma\vdash e : \tau}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \lambda X : T.e : \Pi X : T.\tau} \qquad \frac{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e : \Pi X : T.\tau \qquad \mathcal{W};\mathcal{M}\vdash T' : T}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e\ T' : \tau[T'/X]}$$

$$\frac{\mathcal{W};\mathcal{M}\vdash T' : T \qquad \mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e : \tau[T'/X]}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \left\langle T',\ e\right\rangle : \Sigma X : T.\tau} \qquad \frac{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e : \Sigma X : T.\tau \qquad \mathcal{W};\mathcal{M},\ X' : T;\Sigma;\Gamma,\ \mathbf{x} : \tau[X'/X]\vdash e' : \tau' \qquad X'\notin \mathsf{fv}(\tau')}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \mathsf{let}\ \left\langle X',\ \mathbf{x}\right\rangle = e\ \mathsf{in}\ e' : \tau'}$$

$$\frac{\mathcal{W},\ \phi : \mathsf{ctx};\mathcal{M};\Sigma;\Gamma\vdash e : \tau}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \lambda\phi : \mathsf{ctx}.e : \Pi\phi : \mathsf{ctx}.\tau} \qquad \frac{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e : \Pi\phi : \mathsf{ctx}.\tau \qquad \mathcal{W};\mathcal{M}\vdash \Phi\ \mathsf{wf}}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e\ \Phi : \tau[\Phi/\phi]} \qquad \frac{\mathcal{W};\mathcal{M}\vdash \Phi\ \mathsf{wf} \qquad \mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e : \tau[\Phi/\phi]}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \langle\Phi,\ e\rangle : \Sigma\phi : \mathsf{ctx}.\tau}$$

$$\frac{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash e : \Sigma\phi : \mathsf{ctx}.\tau \qquad \mathcal{W},\phi' : \mathsf{ctx};\mathcal{M};\Sigma;\Gamma,\ \mathbf{x} : \tau[\phi'/\phi]\vdash e' : \tau' \qquad \phi'\notin \mathsf{fv}(\tau')}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \mathsf{let}\ \left\langle \phi',\ \mathbf{x}\right\rangle = e\ \mathsf{in}\ e' : \tau'}$$

$$\frac{\mathcal{W};\mathcal{M}\vdash T : T' \qquad T' = [\Phi]t' \qquad \mathcal{W};\ \mathcal{M};\ \Phi\vdash t' : \mathsf{Type} \qquad \forall i, \big(\mathcal{M};\ \Phi\vdash (p_i \Leftarrow t')\Rightarrow \mathcal{M}_i \quad \mathcal{W};\mathcal{M},\ \mathcal{M}_i;\Sigma;\Gamma\vdash e_i : \tau[[\Phi]\,p_i/X]\big)}{\mathcal{W};\mathcal{M};\Sigma;\Gamma\vdash \mathsf{holcase}\ T\ \mathsf{of}\ (p_1\mapsto e_1)\cdots(p_n\mapsto e_n) : \tau[T/X]}$$

Figure 9. Typing judgement of the computational language (selected rules)

$$\begin{array}{rl}
v & ::= \lambda X : T.e \mid \langle T,\ v\rangle \mid \lambda\phi : \mathsf{ctx}.e \mid \langle\Phi,\ v\rangle \mid \cdots \\
\mathcal{E} & ::= \bullet \mid \mathcal{E}\ T \mid \langle T,\ \mathcal{E}\rangle \mid \mathsf{let}\ \langle X,\ \mathbf{x}\rangle = \mathcal{E}\ \mathsf{in}\ e' \mid \mathcal{E}\ \Phi \mid \langle\Phi,\ \mathcal{E}\rangle \mid \mathsf{let}\ \langle\phi,\ \mathbf{x}\rangle = \mathcal{E}\ \mathsf{in}\ e' \mid \cdots \\
\sigma_{\mathcal{M}} & ::= \bullet \mid \sigma_{\mathcal{M}},\ X\mapsto T \\
\mu & ::= \bullet \mid \mu, l\mapsto [v_1,\cdots,v_n]
\end{array}$$

$$\frac{\mu, e\longrightarrow \mu', e'}{\mu, \mathcal{E}[e]\longrightarrow \mu', \mathcal{E}[e']} \qquad \mu, \mathcal{E}[\mathsf{error}]\longrightarrow \mu, \mathsf{error} \qquad \mu, (\lambda X : T.e)\ T'\longrightarrow \mu, e[T'/X] \qquad \mu, \mathsf{let}\ \langle X,\ \mathbf{x}\rangle = \langle T,\ v\rangle\ \mathsf{in}\ e'\longrightarrow \mu, e'[T/X][v/\mathbf{x}]$$

$$\mu, (\lambda\phi : \mathsf{ctx}.e)\ \Phi\longrightarrow \mu, e[\Phi/\phi] \qquad \mu, \mathsf{let}\ \langle\phi,\ \mathbf{x}\rangle = \langle\Phi,\ v\rangle\ \mathsf{in}\ e'\longrightarrow \mu, e'[\Phi/\phi][v/\mathbf{x}] \qquad \frac{T = [\Phi]t \qquad \Phi\vdash \mathsf{unify}(p_1,t) = \sigma_{\mathcal{M}}}{\mu, \mathsf{holcase}\ T\ \mathsf{of}\ (p_1\mapsto e_1)\cdots(p_n\mapsto e_n)\longrightarrow \mu, e_1[\sigma_{\mathcal{M}}]}$$

$$\frac{T = [\Phi]t \qquad \Phi\vdash \mathsf{unify}(p_1,t) = \bot}{\mu, \mathsf{holcase}\ T\ \mathsf{of}\ (p_1\mapsto e_1)\cdots(p_n\mapsto e_n)\longrightarrow \mu, \mathsf{holcase}\ T\ \mathsf{of}\ (p_2\mapsto e_2)\cdots(p_n\mapsto e_n)} \qquad \mu, \mathsf{holcase}\ T\ \mathsf{of}\ \bullet\longrightarrow \mu, \mathsf{error}$$

Figure 10. Operational semantics for the computational language (selected rules)

and elimination operations:

$$\begin{array}{lcl}
\mathsf{LT}(T) & = & \Sigma X : T.\mathsf{unit} \\
\langle T\rangle & = & \langle T,\ ()\rangle \\
\mathsf{let}\ \langle X\rangle\ = e\ \mathsf{in}\ \ e' & = & \mathsf{let}\ \langle X,\ \_\rangle = e\ \mathsf{in}\ \ e'
\end{array}$$

An example of the use of existential packages is the hyplistWeaken function of Section 3.2, which lifts a list of hypotheses from one context to an extended one. This works by lifting each package of a hypothesis and its associated proof object in turn to the extended context. We open up each package, getting two contextual terms referring to context ϕ; we then repackage them, having them refer to the extended context $\phi,\ x : A$:

$$\begin{array}{l}
\mathsf{hypWeaken} : \Pi\phi : \mathsf{ctx}.\Pi A : [\phi]\,\mathsf{Prop}.(\Sigma H : [\phi]\,\mathsf{Prop}.\mathsf{LT}([\phi]\,H))\rightarrow \\
\qquad\qquad\qquad\qquad\qquad (\Sigma H : [\phi, x : A]\,\mathsf{Prop}.\mathsf{LT}([\phi, x : A]\,H))
\end{array}$$

$$\begin{array}{rl}
\mathsf{hypWeaken}\ \phi\, A\ \mathbf{hyp} & = \mathsf{let}\ \langle X,\ \mathbf{x}_1\rangle = \mathbf{hyp}\ \mathsf{in} \\
& \ \mathsf{let}\ \langle X',\ \_\rangle = \mathbf{x}_1\ \mathsf{in} \\
& \ \big\langle([\phi, x : A]\,(X/\mathsf{id}_\phi)),\ \big\langle[\phi, x : A]\,(X'/\mathsf{id}_\phi)\big\rangle\big\rangle
\end{array}$$

$$\begin{array}{l}
\mathsf{hyplistWeaken} : \Pi\phi : \mathsf{ctx}.\Pi A : [\phi]\,\mathsf{Prop}.\mathsf{hyplist}\ \phi\rightarrow \mathsf{hyplist}\ (\phi, x : A) \\
\mathsf{hyplistWeaken}\ \phi\, A\ \mathbf{hl}\ = \mathsf{map}\ \mathsf{hypWeaken}\ \mathbf{hl}
\end{array}$$

Functions and existentials over contexts Abstraction over contexts works as seen previously: we use it in order to receive the free variables context that further logical terms refer to. Existential packages containing contexts can be used in cases where we cannot statically determine the resulting context of a term. An example would be a procedure for conversion of a propositional formula to CNF, based on Tseitin's encoding [Tseitin 1968]. In this case, a number of new propositional variables might need to be introduced. We could therefore give the following type to such a function:

$$\mathsf{cnf}\ ::\ \Pi\phi : \mathsf{ctx}.\Pi P : [\phi]\,\mathsf{Prop}.\Sigma\phi' : \mathsf{ctx}.\mathsf{LT}([\phi, \phi']\,\mathsf{Prop})$$

Erasure semantics are possible for these constructs, since there is no construct that inspects the structure of a context.

Type constructors At the type level we allow type constructors abstracting both over contexts and over contextual terms. This is what enables the definition of the hyplist type constructor in Section 3.2. Similarly we could take advantage of type constructors to define a generic type for hash tables where keys are logical terms of type $[\phi]A$ and where the type of values is dependent on the key. The type of values is thus given as another type constructor, and the overall type constructor for the hashtable should be:

$$\begin{array}{l}
\mathsf{hash}\ :\ (\Pi X : [\phi]A.*)\rightarrow *\\
\qquad = \lambda\mathsf{res} : (\Pi X : [\phi]A.*).\mathsf{array}\ (\Sigma X : [\phi]A.\mathsf{res}\ X)
\end{array}$$

Implementation of such a data structure is possible, because of a built-in hashing function in our computational language, that maps any logical term to an integer. Some care is needed with this construct, since we want terms that are $\beta\iota$-equivalent to generate the same hashing value (α-equivalence is handled implicitly by using deBruijn indices). To that effect we need to reduce such terms to full $\beta\iota$-normal forms before computing the hash value.

Static and dynamic semantics of pattern matching The last new construct in our computational language is the pattern matching construct for logical terms. Let us first describe its typing rule, as seen in Figure 9. First we make sure that the logical term to be pattern matched upon (the scrutinee) is well-typed. Furthermore, only logical terms that correspond to propositions, predicates or objects of the domain of discourse are allowed to be pattern matched upon; we thus require that the scrutinee's kind is Type. Patterns can be viewed as normal logical terms (of syntactic levels $\mathcal{K}$ and d of the logic) that contain certain unification variables; we will discuss their typing shortly. Unification variables are normal meta-variables that are newly introduced. The result type of the pattern matching construct is dependent on the scrutinee, enabling each branch to have a different type depending on its associated pattern.

At runtime, patterns are attempted to be unified against the scrutinee in sequence, and only the first succeeding branch is evaluated; an error occurs when no pattern can be matched. Unification merely checks whether the pattern and the scrutinee match up to $\beta\iota$-equivalence; if they do, it returns a substitution for the unification variables, which gets applied to the body of the branch.

Higher-order unification in a setting like the logic we are describing is undecidable. We therefore restrict the patterns allowed to linear patterns where unification variables are used at most once. For efficiency reasons, we also impose the restriction that when we use a unification variable in a certain context, it must be applied to the identity substitution of that context. These restrictions are imposed using the pattern typing judgement $\mathcal{M};\Phi \vdash (p \Leftarrow t) \Rightarrow \mathcal{M}'$. This judgement checks that p is a valid pattern corresponding to a logical term of type t, and outputs the unification variables environment $\mathcal{M}'$ that p uses. We further check that patterns are terms in normal form; that is, patterns should only be neutral terms. The details for this judgement, as well as the dynamic semantics of the unification procedure, are given in the extended version of this paper [Stampoulis and Shao 2010].

Revisiting the example of the tautology prover from Section 3.1, the pattern matching would more accurately be written as follows. Note that we also use a return clause in order to specify the result type of the construct. Though this is verbose, most of the contexts and substitutions would be easy to infer from the context.

$$\begin{array}{l}\mathsf{holcase}\ P\ \mathsf{return\ option\ LT}(P)\ \mathsf{with}\\ \quad P_1/\mathsf{id}_\phi \wedge P_2/\mathsf{id}_\phi \qquad \mapsto \cdots\\ \quad \mid \forall x:A/\mathsf{id}_\phi.P'/(\mathsf{id}_\phi,\ x) \quad \mapsto \cdots\end{array}$$

Metatheory We have studied the metatheory for this language, and have found it relatively straightforward using standard techniques. Preservation depends primarily on the substitution lemmas:

Lemma 5.1 (Substitution of contextual terms into expressions) *If* $\mathcal{W};\mathcal{M},\ X:T;\Sigma;\Gamma \vdash e:\tau$ *and* $\mathcal{W};\mathcal{M} \vdash T':T$ *, then* $\mathcal{W};\mathcal{M};\Sigma[T'/X];\Gamma[T'/X] \vdash e[T'/X]:\tau[T'/X]$.

Lemma 5.2 (Substitution of contexts into expressions) *If* $\mathcal{W},\ \phi:\mathit{ctx};\mathcal{M},\ X:T;\Sigma;\Gamma \vdash e:\tau$ *and* $\mathcal{W};\mathcal{M} \vdash \Phi$ *wf, then* $\mathcal{W};\mathcal{M}[\Phi/\phi];\Sigma[\Phi/\phi];\Gamma[\Phi/\phi] \vdash e[\Phi/\phi]:\tau[\Phi/\phi]$.

A lemma detailing the correct behavior of pattern unification is also needed, intuitively saying that by applying the substitution yielded by the unification procedure to the pattern, we should get the term we are matching against:

Lemma 5.3 (Soundness of unification) *If* $\Phi \vdash \mathit{unify}(p,t) = \sigma_{\mathcal{M}}$, $\mathcal{W};\mathcal{M} \vdash [\Phi]t:T_0$ *and* $\mathcal{W};\mathcal{M} \vdash (p \Leftarrow T_0) \Rightarrow \mathcal{M}'$ *then* $\mathcal{W};\mathcal{M} \vdash \sigma_{\mathcal{M}}:\mathcal{M}'$ *and* $([\Phi]\,p)[\sigma_{\mathcal{M}}/\mathcal{M}'] =_{\beta\iota} T$.

Theorem 5.4 (Preservation) *If* $\bullet;\bullet;\Sigma;\bullet \vdash e:\tau$, $\bullet;\bullet;\Sigma;\bullet \vdash \mu$ *and* $\mu,e \longrightarrow \mu',e'$ *then there exists a* $\Sigma' \supseteq \Sigma$ *so that* $\bullet;\bullet;\Sigma';\bullet \vdash e':\tau$ *and* $\bullet;\bullet;\bullet;\Sigma';\bullet \vdash \mu'$.

Proof by structural induction on the step relation $\mu,e \longrightarrow \mu',e'$, made relatively simple for the new constructs by use of the above lemmas. The proofs for common constructs do not require special provisions and follow standard practice [Pierce 2002].

Progress depends on the following canonical forms lemma:

Lemma 5.5 (Canonical forms) *If* $\bullet;\bullet;\Sigma;\bullet \vdash v:\tau$ *then:*
If $\tau = \Pi X:T.\tau'$, *then* $v = \lambda X:T.e$.
If $\tau = \Sigma X:T.\tau'$, *then* $v = \langle T',\ v'\rangle$ *and* $\bullet;\bullet \vdash T':T$.
If $\tau = \Pi\phi:\mathit{ctx}.\tau'$, *then* $v = \lambda\phi:\mathit{ctx}.e$.
If $\tau = \Sigma\phi:\mathit{ctx}.\tau'$, *then* $v = \langle \Phi',\ v'\rangle$ *and* $\bullet;\bullet \vdash \Phi'$ *wf.*
...

Theorem 5.6 (Progress) *If* $\bullet;\bullet;\Sigma;\bullet \vdash e:\tau$ *then either e is a value, or, for every* μ *such that* $\bullet;\bullet;\Sigma;\bullet \vdash \mu$, $\mu,e \longrightarrow$ *error, or there exists* e' *and* μ' *such that* $\mu,e \longrightarrow \mu',e'$.

The proof is a straightforward structural induction on the typing derivation of e. The only cases for the error result come from non-exhaustive pattern matching or out-of-bounds access in arrays. More details about the proofs can be found in the extended version of this paper [Stampoulis and Shao 2010]. We have found these proofs to be relatively orthogonal to proofs about the type safety of the basic constructs of the computational language.

From type safety we immediately get the property that if an expression evaluates to an existential package containing a proof object π, then it will be a valid proof of the proposition that its type reflects. This means that, at least in principle, the type checker of the logic language does not need to be run again, and decision procedures and tactics written in our language always return valid proof objects. In practice, the compiler for a language like this will be much larger than a type checker for the logic language so we will still prefer to use the second as our trusted base. Furthermore, if we are only interested in type checking proof objects yielded from our language, our trusted base can be limited to a type checker for the base logic language, and does not need to include the extensions with meta-variables and parametric contexts.

6. Implementation and examples

We have created a prototype implementation of a type-checker and interpreter for VeriML, along with an implementation of the higher-order logic we use. Readers are encouraged to download it from `http://flint.cs.yale.edu/publications/veriml.html`. The implementation is about 4.5kLOC of OCaml code, and gives a VeriML toplevel to the user where examples can be tested out. Concrete syntax for VeriML is provided through Camlp4 syntax extensions.

The implementation of the logic can be used in isolation as a proof checker, and is done using the PTS style; thus generalization to a theory like CIC is relatively straightforward. Binding is represented using the locally nameless approach. Both at the level of the logic, as well as at the level of the computational language, we allow named definitions. In the computational language we allow some further constructs not shown here, like mutually recursive function definitions, as well as a printing function for HOL terms, used primarily for debugging purposes.

We are currently working on a code generator that translates well-typed programs of our computational language into normal OCaml code for efficiency. This translation essentially is a type-erasure operation, where the annotations needed to support our dependent type system are removed and we are thus left with code similar to what one would write for an LCF-style theorem prover. We are investigating the possibility of emitting code for existing frameworks, like Coq or HOL.

We have used this implementation to test two larger examples that we have developed in VeriML; these are included in the language distribution. The first example is an extension of the decision procedure for the theory of equality given in Section 3, so that uninterpreted functions are also handled. Furthermore, we use this decision procedure as part of a version of the tauto tactic that we showed earlier. Equality hypotheses are used in order to create the equivalence class hash table; terms contained in goals are then viewed up to equivalence based on this hash table, by using the functions provided by the decision procedure.

The second example is a function that converts a proposition P into its negation normal form P', returning a proof object witnessing the fact that P' implies P. Such proof objects are not built manually. They get produced by a version of the tauto tactic with all the extensions that we described in Section 3, along with handling of the False proposition. This tactic is enough to prove all the propositional tautologies required in NNF conversion.

7. Related work

There is a large body of existing related work that we should compare the work we described here to. We will try to cover other language and framework designs that are similar in spirit or goals to the language design we have described here.

The LTac language [Delahaye 2000, 2002] available in the Coq proof assistant is an obvious point of reference for this work. LTac is an untyped domain-specific language that can be used to define new tactics by combining existing ones, employing pattern matching on propositions and proof contexts. Its untyped nature is sometimes viewed as a shortcoming (e.g. in [Nanevski et al. 2010]), and there are problems with handling variables and matching under binders. Our language does not directly support all of the features of LTac. Still, we believe that our language can serve as a kernel where missing features can be developed in order to recover the practicality that current use of LTac demonstrates. Also, our language is strongly typed, statically guarantees correct behavior with regards to binding, and gives access to a richer set of programming constructs, including effectful ones; this, we believe, enables the development of more robust and complex tactics and decision procedures. Last, our language has formal operational semantics, which LTac lacks to the best of our knowledge, so the behaviour of tactics written in it can be better understood.

The comparison with the LCF approach [Gordon et al. 1979] to building theorem provers is interesting both from a technical as well as from a historical standpoint, seeing how ML was originally developed toward the same goals as its extension that we are proposing here. The LCF approach to building a theorem prover for the logic we have presented here would basically amount to building a library inside ML that contained implementations for each axiom, yielding a term of the abstract thm datatype. By permitting the user to only create terms of this type through these functions, we would ensure that all terms of this datatype correspond to valid derivations – something that depends of course on the type safety of ML. Our approach is different in that the equivalent of the thm datatype is dependent on the proposition that the theorem shows. Coupled with the rest of the type system, we are able to specify tactics, tacticals, and other functions that manipulate logical terms and theorems in much more detail, yielding stronger static guarantees. Essentially, where such manipulation is done in an untyped manner following the usual LCF approach, it is done in a strongly typed way using our approach. We believe that this leads to a more principled and modular programming paradigm, a claim that we aim to further substantiate with future work.

In recent years many languages with rich dependent type systems have been proposed, which bear similarity to the language design we proposed here; unfortunately they are too numerous to cover here but we refer the reader to three of the most recent and relevant proposals [Norell 2007, Fogarty et al. 2007, Chen and Xi 2005] and the references therein. The way that our approach contrasts with languages like these is that we are not primarily interested in certifying properties of code written in our language. We rather view our language as a foundation for an “extensible” proof assistant, where proofs about code written in other (however richly typed or untyped) languages can be developed in a scalable manner.

Of the above languages, we believe Concoqtion [Fogarty et al. 2007] is the one that is closest to our language, as it embeds the full CIC universe as index types for use inside a version of the ML language. Our language does the same thing, even though only a subset of CIC is covered; the point of departure compared to Concoqtion is that our language also includes a computational-level pattern matching construct on such terms. Thus logical terms are not to be viewed only as index types, and actually have a runtime representation that is essential for the kind of code we want to write. Pattern matching for logical terms would amount to a typecase-like construct in languages like Concoqtion, a feature that is generally not available in them.

It is interesting to contrast our framework with computational languages that deal with terms of the LF logical framework [Harper et al. 1993], like Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008]. Especially Beluga has been an inspiration for this work, and our use of meta-variables and context polymorphism is closely modeled after it. LF provides good support for encoding typing judgements like the ones defining our logic; in principle our logic could be encoded inside LF, and languages such as the above could be used to write programs manipulating such encodings with static guarantees similar to the ones provided through our language. In practice, because of the inclusion of a notion of computation inside our logic, for which LF does not provide good support, this encoding would be a rather intricate exercise. The $\beta\iota$-reduction principles would have to be encoded as relations inside LF, and explicit witnesses of $\beta\iota$-equivalence would have to be provided at all places where our logic alludes to it. We are not aware of an existing encoding of a logic similar to the one we describe inside LF, and see it as a rather complicated endeavour. This situation could potentially be remedied by a framework like Licata et al. [2008] that considers adding computation to a subset of LF; still, this framework currently lacks dependent types, which are essential for encoding the judgements of our logic in LF.

Even if this encoding of our logic was done, the aforementioned computational languages do not provide the imperative constructs that we have considered. Last, writing procedures that could be part of a proof assistant, even for simpler logics, has not been demonstrated yet in these languages, as we do here.

Another framework that is somewhat related is Hoare Type Theory (HTT) and the associated YNot project [Chlipala et al. 2009, Nanevski et al. 2010]. This framework attempts to extend the programming model available in the Coq proof assistant with support for effectful features like mutable references. This is done by axiomatically extending Coq’s logic with support for a stateful monad; essentially, imperative computational features are integrated inside the logical framework. Our approach, instead, integrates the logical framework inside a computational language, keeping the two as orthogonal as possible. Thus, it does not require any significant metatheoretic additions to the logic. Additional features in our computational language like concurrency could be easily added, as long as they are type safe. In principle, one could use HTT in conjunction with the standard proof-by-reflection technique in order to program decision procedures in an imperative style inside the proof assistant. We are not aware of a development based on this idea; for example, even though a decision procedure for the EUF theory is proved correct in Nanevski et al. [2010], it is not evident whether this procedure can be used as a tactic in order to prove further goals. We would be interested in attempting this approach in future work.

8. Future work and conclusion

There are many directions for extending the work we have presented here. One is to investigate how to replicate most of the functionality of a language like LTac inside our language, by providing ways to perform pattern matching with back-tracking, and pattern matching on contexts. Furthermore, we want to explore issues of type and term inference in our language in order to limit its current verbosity, especially with respect to context quantification and instantiation. Last, the context manipulation currently allowed by our language is relatively limited, e.g. with respect to contraction of variables out of contexts. We are investigating how such limitations can be lifted, without complicating the language design or its metatheory.

We have described VeriML, a new language design that introduces first-class support for a logical framework modeled after HOL and CIC inside a computational language with effects. The language allows pattern matching on arbitrary logical terms. A dependent type system is presented, which allows for strong specifications of effectful computation involving such terms. We have shown how tactics and decision procedures can be implemented in our language, providing strong static guarantees while at the same time allowing a rich programming model with non-termination and mutable references.

Acknowledgments

We thank anonymous referees for their comments on this paper. This work is supported in part by NSF grants CCF-0811665, CNS-0915888, and CNS-0910670.

References

Henk P. Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual (version 8.3), 2010.

S. Boutin. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science*, 1281:515–529, 1997.

A.R. Bradley and Z. Manna. *The calculus of computation: decision procedures with applications to verification*. Springer-Verlag New York Inc, 2007.

C. Chen and H. Xi. Combining programming with theorem proving. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, page 77. ACM, 2005.

A. Chlipala. Certified Programming with Dependent Types, 2008. URL `http://adam.chlipala.net/cpdt/`.

Adam J. Chlipala, J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90. ACM, 2009.

D. Delahaye. A tactic language for the system Coq. *Lecture notes in computer science*, pages 85–95, 2000.

D. Delahaye. A proof dedicated meta-language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002.

X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proc. 2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 54–69. Springer, October 2008.

S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121. ACM New York, NY, USA, 2007.

M.J. Gordon, R. Milner, and C.P. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Springer-Verlag Berlin*, 10:11–25, 1979.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

J. Harrison. HOL Light: A tutorial introduction. *Lecture Notes in Computer Science*, pages 265–269, 1996.

C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. *ACM SIGPLAN Notices*, 44(1):441–453, 2009.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

D.R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *Logic in Computer Science, 2008. LICS'08*, pages 241–252, 2008.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73. ACM New York, NY, USA, 2006.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 261–274. ACM, 2010.

T. Nipkow, L.C. Paulson, and M. Wenzel. Isabelle/HOL : A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS, 2002.

Ulf Norell. Towards a practical programming language based on dependent type theory. Technical report, Goteborg University, 2007.

C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. *Lecture Notes in Computer Science*, pages 328–328, 1993.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–382. ACM, 2008.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 163–173. ACM New York, NY, USA, 2008.

B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. *Lecture Notes in Computer Science*, 4960:93, 2008.

K. Slind and M. Norrish. A brief overview of HOL4. In *TPHOLs*, pages 28–32. Springer, 2008.

M. Sozeau. Subset coercions in Coq. *Types for Proofs and Programs*, pages 237–252, 2007.

A. Stampoulis and Z. Shao. VeriML: Typed computation of logical terms inside a language with effects (extended version). Technical Report YALEU/DCS/TR-1430, Dept. of Computer Science, Yale University, New Haven, CT, 2010. URL `http://flint.cs.yale.edu/publications/veriml.html`.

G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.