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

In this paper, we propose a novel language design that couples a type-safe effectful computational language with first-class support for logical terms like propositions and proofs. As our logical level we use a small subset of CIC, while our computational language is 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 changes to the logic language, so soundness is guaranteed. We argue that such a language design is well-tailored to write tactics for proof assistants like Coq and Isabelle, and show some initial examples.

A key technical challenge in designing such a language is dealing successfully with open terms of the logical level. We present a novel solution to this problem, that maintains type safety while requiring relatively small programming overhead. We show how it is related to similar solutions developed for computation with LF terms.

1. Introduction

In the process of software certification through the use of a mechanized logic with full, machine-checkable proof objects, it is often the case that some sort of computation involving the terms of this logic is being defined. The simplest and most common such case is writing a recursive function inside the logic itself, for example using the Fixpoint mechanism in the case of the Coq proof assistant [Bertot et al. 2004], or the "fun" mechanism of Isabelle/HOL [Nipkow et al. 2002]. The very proof scripts that we use in a tactics-based proof assistant also involve a limited notion of computation of logical terms, if we view tactics as functions from proof states to proof states plus proof objects substantiating the implication of the initial proof state from the resulting one. Proof scripts enable us to compose such tactics together in a number of ways in order to compute a proof object for the original proposition. Through the use of a tactical language like LTac [Delahaye 2000], this notion of computation is substantially extended, for example with the ability to programmatically perform pattern matching on propositions. A further way that one can use to perform computation with logical terms is to actually go into the implementation of the logical framework we are using, which is usually written in a language like ML, where we have direct access to the encodings of the logical terms.

In many developments a mix of these different ways to define computation involving logical terms is essential. For example, when defining a decision procedure for a certain class of propositions using the technique of proof-by-reflection, the steps we could take are the following: first, define a datatype inside the logic to represent the class of propositions that our decision procedure works on. Second, write a recursive function inside the logic deciding whether a particular term of that datatype is true. Third, prove a theorem showing that whenever this function returns true, the proposition that it represents is also true, through the use of a proof script. Last, write a function using a mechanism like LTac that converts a proposition of our logic belonging in the specific class of propositions, into the associated defined datatype.

Furthermore, if the algorithm that we want to use in order to decide the truth of a proposition involves heavy use of imperative features, it would probably be best to develop the decision procedure inside the implementation language of the logical framework. In that case, we could either directly use the existing encodings of the logical terms, or we could use forms more well-suited to the decision procedure we will be implementing and provide the appropriate conversion functions. A further stage of writing a verified validator for the results of this procedure could be required.

The primary reason why this many different languages are used is that each one has its own set of benefits and shortcomings. Functions defined in the computational language of a logic are the easiest ones to use since they can freely be used inside any logical term; but because of this, the computational model permitted is limited so that the soundness of the logic is maintained. Such functions need to be total, and also use of effectful programming constructs like mutable references is not permitted. Furthermore, pattern matching on terms like propositions in this level is not a widespread feature in current logical frameworks. Mechanisms like LTac include support for pattern matching on propositions, and permit general recursion, but being domain-specific languages for writing tactics, they do not include programming constructs present in general-purpose languages, like rich data structures. Last, using the implementation language of the logical framework provides the richest programming model compared to the above approaches, involving use of arbitrary data structures and effectful operations like mutable references, non-terminating recursion, and exceptions. This comes at the price of having to deal directly with the internal representation of logical terms inside that language, which might be cumbersome to work with. Also, the static guarantees about the logical term manipulation code written in such a language are limited; even specifications as simple as saying that a function accepts a proposition as an argument and returns a proof object of that particular proposition are not supported.

We propose that a single, unified language that integrates a logical framework as a first-class citizen inside a general-purpose, effectful language, could be used in many places where the above languages or a mix thereof is currently used. We would want that such a language provides explicit support for propositions, proof objects, and other terms of a logical framework; also, a rich type system enabling the user to specify relationships between such terms. We believe that such a language can enable the user to define new tactics and decision procedures in a more principled and modular way through the use of rich data structures, while at

the same time lowering the barrier that is currently associated with using programming language features like imperativity.

In this paper we present the beginnings of such a language design. Our design is based on extending ML with support for a small logical framework, which shares certain characteristics with CIC, the logic behind Coq. We identify one key technical challenge in this endeavour, namely designing a type-system for this language that deals effectively with open logical terms. We present our solution to it, which we consider to be the main technical contribution of the present paper. Last, we show examples of simple tactics written in this language.

2. The logic language

We begin our presentation with a description of the logic that we will be using to base our discussion. We use a higher-order logic with support for inductively defined data-types, 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, that can be viewed as witnesses of derivations in the logic.

This framework is based on λHOL as presented in [Barendregt and Geuvers 2001], extended with inductive definitions and a reduction relation for total recursive functions, in the style of CIC [Bertot et al. 2004]. Alternatively, we can view this framework as a subset of CIC, were we have omitted universes other than Prop and Type, as well as polymorphism and dependent types in Type[1]. Logical consistency of CIC ([Werner 1994]) therefore implies logical consistency of our system.

The syntax of our framework is presented in 1. The syntactic category t corresponds to logical terms, that include propositions, terms of inductively defined datatypes, and higher-order functions over such terms. Such terms get assigned kinds of the syntactic category $\mathcal{K}$, with all propositions being assigned kind Prop. The syntactic category M represents proof objects, whose type corresponds to one proposition. The definitions context Δ is populated by definitions of inductive datatypes (each datatype gets assigned a distinct kind T), inductive predicates and connectives, as well as definitions of total recursive functions over terms of inductive datatypes.

Each one of these definitions gives us a set of different constants at the logical term and proof object level. Inductive datatype definitions introduce other than the newly defined-kind, a set of constructor logical term constants, as well as a proof object constant whose propositional type corresponds to the induction principle for that datatype. Similarly, inductive predicate definitions introduce a constant logical term representing the predicate, as well as proof object constants corresponding to its constructors and its inductive elimination principle. Some examples of such definitions are given in figure 2: natural numbers, the addition function between two numbers, the natural number equality predicate, and logical conjunction.

We present the main typing judgements for this framework in figure 3. These judgements use the logic term variables environment Φ, and the proof object variables environment Ψ. We haven't included the judgements $\Delta \vdash c_\tau : \mathcal{K}$ and $\Delta \vdash c_M : \tau$ that have to do with inductive definitions and recursive functions; these should include the standard checks for positivity of inductive definitions, totality of recursive functions, and define proper inductive elimination schemata. These are defined following CIC (see for example [Paulin-Mohring 1993]). Note that in the rest of the paper we

[1] Since no polymorphism over Type is allowed, no question of impredicativity for Type arises, so there is no need for the stratified Type universe of CIC.

$$
\begin{array}{rrcl}
\textit{(kinds)} & \mathcal{K} & ::= & \mathsf{Prop} \mid T \mid \mathcal{K}_1 \to \mathcal{K}_2 \\
\textit{(logical terms)} & t & ::= & u \mid t_1 \Rightarrow t_2 \mid \forall u : \mathcal{K}.t \mid \lambda u : \mathcal{K}.t \mid t_1\, t_2 \mid c_t \\
\textit{(proof objects)} & M & ::= & X \mid \lambda X : t.M \mid M_1\, M_2 \mid \lambda u : \mathcal{K}.M \mid M_1\, t_2 \mid c_M \\
\textit{(definitions)} & \Delta & ::= & \bullet \mid \Delta,\ \textbf{Inductive}\ T : \mathsf{Type} := \{\overrightarrow{c_t : \mathcal{K}}\} \\
 & & & \mid \Delta,\ \textbf{Inductive}\ c_t(\overrightarrow{u : \mathcal{K}}) : \mathcal{K}_1 \to \cdots \to \mathcal{K}_n \to \mathsf{Prop} := \{\overrightarrow{c_M : t}\} \\
 & & & \mid \Delta,\ \textbf{Fixpoint}\ c_t(\overrightarrow{u : T}) : \mathcal{K} := \{\cdots\} \\
\textit{(logical terms env.)} & \Phi & ::= & \bullet \mid \Phi,\ u : \mathcal{K} \\
\textit{(proof objects env.)} & \Psi & ::= & \bullet \mid \Psi,\ X : t
\end{array}
$$

Figure 1. Syntax of the logic language

$$
\begin{array}{l}
\textbf{Inductive}\ \mathsf{Nat} : \mathsf{Type} := \\
\quad \mathsf{Zero} : \mathsf{Nat} \\
\quad \mid \mathsf{Succ} : \mathsf{Nat} \to \mathsf{Nat}.
\end{array}
\qquad
\begin{array}{l}
\textbf{Fixpoint}\ (+)\ (u : \mathsf{Nat})\ (v : \mathsf{Nat}) : \mathsf{Nat} := \\
\quad \mathsf{match}\ u\ \mathsf{with} \\
\qquad \mathsf{Zero} \mapsto v \\
\qquad \mathsf{Succ}\ u' \mapsto (+)u'v \\
\quad \mathsf{end}.
\end{array}
$$

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

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

Figure 2. Examples of definitions

assume a fixed definitions context and therefore omit Δ from our typing rules.

Of interest is the PO-CONVERT typing rule for proof objects. As we have seen, the logical terms include terms for λ abstractions and total recursive functions. 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 evaluation of recursive function applications. With this rule, logical terms that are βι-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. We believe that this particular feature of having a notion of computation within the logic language is one of the defining characteristics of CIC, and we have included this in our logic to show that a computational language as the one we propose is still possible for such a framework.

3. Overview of the language design

Having described the logical framework that we will be using, we will continue by providing a high-level overview of a computational language that manipulates logical terms and proof objects of this framework. At the same time, we will consider how the type system for this language should work.

As we have mentioned in the introduction, the aim of designing such a language is to be able to encompass many different aspects of computation with logical terms and proof objects inside a unified language that offers a rich programming model and strong static guarantees. We want to be able to write programs that produce proof objects of specific propositions (similar to writing proof scripts), functions that transform propositions into easier to prove goals (similar to defining tactics), as well as functions that compute terms of inductively defined types using a richer notion of computation than the one provided by the logic language.

As the basis of our computational language, we consider an ML-like call-by-value functional language, with general recursion

Typing for logical terms:

$$\frac{u : \mathcal{K} \in \Phi}{\Delta; \Phi \vdash u : \mathcal{K}} \qquad \frac{\Delta; \Phi \vdash t_1 : \mathsf{Prop} \qquad \Delta; \Phi \vdash t_2 : \mathsf{Prop}}{\Delta; \Phi \vdash t_1 \Rightarrow t_2 : \mathsf{Prop}}$$

$$\frac{\Delta; \Phi, u : \mathcal{K} \vdash t : \mathsf{Prop}}{\Delta; \Phi \vdash \forall u : \mathcal{K}.t : \mathsf{Prop}} \qquad \frac{\Delta; \Phi, u : \mathcal{K} \vdash t : \mathcal{K}'}{\Delta; \Phi \vdash \lambda u : \mathcal{K}.t : \mathcal{K} \to \mathcal{K}'}$$

$$\frac{\Delta; \Phi \vdash t_1 : \mathcal{K} \to \mathcal{K}' \qquad \Delta; \Phi \vdash t_2 : \mathcal{K}}{\Delta; \Phi \vdash t_1\ t_2 : \mathcal{K}'} \qquad \frac{\Delta \vdash c_t : \mathcal{K}}{\Delta; \Phi \vdash c_t : \mathcal{K}}$$

Typing for proof objects:

$$\frac{X : t \in \Psi}{\Delta; \Phi; \Psi \vdash X : t}$$

$$\frac{\Delta; \Phi; \Psi, X : t \vdash M : t' \qquad \Delta; \Phi \vdash t \Rightarrow t' : \mathsf{Prop}}{\Delta; \Phi; \Psi \vdash \lambda X : t.M : t \Rightarrow t'}$$

$$\frac{\Delta; \Phi; \Psi \vdash M_1 : t \Rightarrow t' \qquad \Delta; \Phi; \Psi \vdash M_2 : t}{\Delta; \Phi; \Psi \vdash M_1\ M_2 : t'}$$

$$\frac{\Delta; \Phi, u : \mathcal{K}; \Psi \vdash M : t}{\Delta; \Phi; \Psi \vdash \lambda u : \mathcal{K}.M : \forall u : \mathcal{K}.t}$$

$$\frac{\Delta; \Phi; \Psi \vdash M : \forall u : \mathcal{K}.t' \qquad \Delta \vdash t : \mathcal{K}}{\Delta; \Phi; \Psi \vdash M\ t : t'[t/u]} \qquad \frac{\Delta \vdash c_M : t}{\Delta; \Phi; \Psi \vdash c_M : t}$$

$$\frac{\Delta; \Phi; \Psi \vdash M : t \qquad t =_{\beta\iota} t'}{\Delta; \Phi; \Psi \vdash M : t'}\ \text{PO-CONVERT}$$

Figure 3. Main typing judgements of the logical framework

and mutable references. To add support for our logical framework, we could encode its terms as algebraic datatypes, and implement a logical type-checker based on the typing judgements we presented. To perform any kind of computation with these terms, we could either let the user directly manipulate their encodings, in which case the logical type-checker would need to be run before deeming them to be valid; or, we could leave the datatype abstract and only provide wrapper functions to the user for creating and combining logical terms and proof objects, that perform validation through the type-checker at the same time. We view these approaches as being roughly equivalent to the approaches taken by Coq and LCF-style theorem provers, respectively.

The problem with this approach is that a lot of information that could be known statically is lost. Consider this alternative approach, specifically for proof objects: we explicitly add a distinguished base type for proof objects into our computational language, and provide an introduction form for them, which merely lifts a proof object term into a computational language term. The type system of the logic language is embedded inside the type system of the computational language, therefore enabling us to statically determine whether a proof object lifted into a computational term, is valid or not. (This of course depends on having static information about the variable environments Φ and Ψ that the proof object type checks under.) Furthermore, we can statically determine the proposition that a proof object corresponds to, so we can make that part of the static type of the proof object. This reveals further static information that was lost with the previous approach, since all proof objects need to be typed under the same type – namely, the recursive datatype we've defined for their encodings.

Following this approach, terms representing proof objects inside the computational language become dependently typed on the proposition that they prove. A typing rule for introducing such objects could look like the following:

$$\frac{\Phi; \Psi \vdash M : P}{\Phi; \Psi; \Gamma \vdash \langle M \rangle : \mathsf{prf}(P)}$$

where we have used P for logical terms known to be propositions, and Γ for the environment of computational variables. The associated elimination form should bind a lifted proof object to a new proof object variable (in environment Ψ), which will get replaced for the actual proof object once it has been evaluated. This choice for typing proof objects can be extended to logical terms; since these could appear inside the type of the proof objects, it is only natural to support them by adding constructs for dependent products and sums over such logical terms. These constructs, along with the elimination construct of proof objects, let us populate the contexts Φ and Ψ and provide static information about their contents. We are thus lead to a standard dependent type system, that lets us specify input-output relations of functions that perform computation involving logical terms and proof objects. For example, the type of a tactic-like function that receives a proposition as an argument and returns a (simplified) proposition along with a proof that it implies the original one would be the following:

$$\Pi P : \mathsf{Prop}.\Sigma P' : \mathsf{Prop}.\mathsf{prf}(P' \Rightarrow P)$$

For the aims of the language, it is apparent that we need a further construct, namely a way to perform pattern matching on arbitrary propositions and logical terms. This too is something that can easily be supported, provided that terms are reduced to normal form before being matched against a pattern, and also that unification variables in patterns appear only once. For the stated goals of our computational language, a similar pattern-matching construct for proof objects does not seem necessary, since the specific form that a proof object takes is not deemed important. Therefore our computational language adopts proof-irrelevance.

It is important that this pattern matching construct doesn't only deal with closed logical terms, but can work even in the case where the terms are open, mentioning free variables that might not be instantiated at runtime. To see why this is so, consider the example where we use the pattern matching construct to inspect a first-order proposition, in order to recursively apply some simplification, e.g. to change all propositions of the form $x = x$ into $True$. It is easy to imagine how such a procedure could work for connectives like logical conjunction and disjunction; the case of interest is how the quantification case is handled. The body of the quantifier is a proposition, but it is a proposition that might mention an extra free variable compared to the proposition we're pattern matching against. So even if the pattern matching was originally hapenning on a closed proposition, the recursive call will introduce pattern matching on an open proposition. Similarly, proof objects that the recursive call returns might mention this extra free variable, so we need to consider how open proof objects are handled too.

This concludes the minimum constructs that the language design we propose should have. We will now focus on how a type system for this language can be defined so that the above features are supported.

A naive account for open terms within our type system could easily compromise the static guarantees that our approach has given us so far. We are running the risk of having free variables escape the scope where they are defined, which would invalidate the logical term and proof object variable environments that we have statically

determined. Our type system therefore needs to accurately track the dependence on free variables.

We believe that this issue arises whenever open terms belonging to structures with binding become first-class citizens of a strongly-typed language as the one we describe here. Some such cases that we have come across are the addition of binding support to ML datatypes as done in FreshML [Shinwell et al. 2003, Pottier 2007]; supporting dynamically-scoped variables inside a statically-scoped language (called fluid binding in [Harper 2007]); and when designing computational languages that manipulate LF terms, like Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008].

Our initial attempt at solving this issue was based on the solution used in Delphin. The core idea of this approach is that an extra ambient context of free variables is maintained during typechecking; a programming construct binds a new free variable which can only be used within the construct's scope (written as $\nu u : \mathcal{K}.e$); and dependence on free variables is tracked by the type system by assigning a special ∇-type to the new construct. We have found that this solution has the desirable property that the extra overhead required on the programmer's part to deal effectively with open terms is relatively small, and it is expressive enough for the kind of code we have tried to write. Unfortunately, we have found that the type system of this approach does not interact well with other features of the language, like mutable references, and free variables can escape their scope through the use of such features.

The approach used in Beluga abandons the ambient free variables context in favor of explicitly boxing open terms with the free variables they depend on, and using explicit substitutions for free variables whenever these terms are used. The benefit of this approach compared to the previous one is that its type-system is orthogonal to the rest of the computational language so effectful operations can still be supported without compromising type safety. On the other hand, the programming overhead is relatively large, requiring explicit manipulation of free variable environments and substitutions; also, it would require changing the typing judgements of our logic so that dependence on free variables is tracked and explicit substitutions are accounted for.

We have ultimately arrived at a hybrid approach between these two, which is composed of a surface language that presents a programming interface similar to the first approach, and of an intermediate language that is largely similar to the second, albeit the use of explicit substitutions is greatly reduced. Both languages have a type system, and a translation from well-typed terms of the surface language to well-typed terms of the intermediate language exists. The surface language is meant as a programming convenience, so it is not given operational semantics directly, but can be evaluated through translation to the intermediate language.

We will now summarize the main ideas of our approach, and we will present these languages in more detail in the next sections. First, we require that open terms are locally closed with respect to their free variables environment when they are introduced. Second, we require that such open terms be used only in places where a compatible free variables environment is available; having this notion of compatibility regard the environments as ordered lets us avoid requiring explicit substitutions in many cases. Third, when providing an explicit substitution for a free variable is required, we reuse the β-reduction already available for logic terms, by converting the free variable into a bound one through λ-abstraction. Fourth, polymorphism over free variable environments is permitted. Last, the typing of the surface language ensures that available logical terms and proof objects require environments that are always compatible with the currently available free variables environment, so the programmer does not need to worry about such issues.

$$
\begin{array}{rcl}
F & ::= & \bullet \mid F, u : \mathcal{K} \mid F, \phi \\
\widetilde{\mathcal{K}} & ::= & [F]\mathcal{K} \\
\widetilde{t} & ::= & [F]t \\
\widetilde{M} & ::= & [F]M \\
\widetilde{\Phi} & ::= & \bullet \mid \widetilde{\Phi}, u : \widetilde{\mathcal{K}} \\
\Omega & ::= & \bullet \mid \Omega, \phi : \mathsf{ctx} \\
\\
\sigma & ::= & \mathsf{prf}(\widetilde{t}) \mid \Sigma u : \widetilde{\mathcal{K}}.\sigma \mid \Pi u : \widetilde{\mathcal{K}}.\sigma \mid \Pi \phi.\sigma \\
 & & \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \mid 1 \mid \mathsf{ref}\ \sigma \\
e & ::= & \left\langle \widetilde{M} \right\rangle \mid \mathsf{openprf}\ (e\ \mathsf{binding}\ F)\ \mathsf{as}\ X\ \mathsf{in}\ e' \\
 & & \mid \left\langle \widetilde{t}, e \right\rangle \mid \mathsf{open}\ (e\ \mathsf{binding}\ F)\ \mathsf{as}\ \langle u, x \rangle\ \mathsf{in}\ e' \\
 & & \mid \Lambda u : \widetilde{\mathcal{K}}.e \mid e\ \widetilde{t} \mid \Lambda \phi.e \mid e\ [F] \\
 & & \mid \lambda x : \sigma.e \mid e_1\ e_2 \mid x \mid \mathsf{fix}\ x : \sigma.e \mid (e_1, e_2) \mid \mathsf{proj}_i\ e \\
 & & \mid \mathsf{inj}_i\ e \mid \mathsf{case}(e, x.e_1, x.e_2) \mid \mathsf{unit} \mid \mathsf{ref}\ e \mid e := e' \mid\ !e \mid l \\
 & & \mid [F].\mathsf{Itermcase}\ t\ \mathsf{with}\ (p_1 \mapsto e_1) \cdots (p_n \mapsto e_n) \\
p & ::= & ?u : \mathcal{K} \mid p_1 \Rightarrow p_2 \mid \forall [\mathcal{K}].p \mid t \mid p_1\ p_2
\end{array}
$$

Figure 4. Syntax for the intermediate language

$$
\frac{}{\Omega \vdash \bullet} \qquad \frac{\Omega \vdash F}{\Omega \vdash F, u : \mathcal{K}} \qquad \frac{\Omega \vdash F \qquad \phi : \mathsf{ctx} \in \Omega}{\Omega \vdash F, \phi}
$$

$$
\frac{\Omega \vdash F \qquad \widetilde{\Phi}@F, \lfloor F \rfloor; \widetilde{\Psi}@F \vdash M : t}{\Omega; \widetilde{\Phi}; \widetilde{\Psi} \vdash [F]M : [F]t} \qquad \frac{\Omega \vdash F \qquad \widetilde{\Phi}@F, \lfloor F \rfloor \vdash t : \mathcal{K}}{\Omega; \widetilde{\Phi} \vdash [F]t : [F]\mathcal{K}}
$$

$$
\text{where: } F \subseteq F' = \begin{cases} \bullet \subseteq F' & \\ u : \mathcal{K}, F \subseteq u' : \mathcal{K}, F' & \text{if } F \subseteq F' \\ \phi : \mathsf{ctx}, F \subseteq \phi : \mathsf{ctx}, F' & \text{if } F \subseteq F' \end{cases}
$$

$$
\widetilde{\Phi}@F = \begin{cases} \bullet & \text{if } \widetilde{\Phi} = \bullet \\ \widetilde{\Phi}'@F, u : \mathcal{K} & \text{if } \widetilde{\Phi} = \widetilde{\Phi}', u : [F_1]\mathcal{K} \text{ and } F = F_1, F_2 \\ \widetilde{\Phi}'@F & \text{if } \widetilde{\Phi} = \widetilde{\Phi}', u : [F']\mathcal{K} \wedge F' \not\subseteq F \end{cases}
$$

$$
\lfloor F \rfloor = \begin{cases} \bullet & \text{if } F = \bullet \\ \lfloor F' \rfloor, u : \mathcal{K} & \text{if } F = F', u : \mathcal{K} \\ \lfloor F' \rfloor & \text{if } F = F', \phi : \mathsf{ctx} \end{cases}
$$

Figure 5. Typing for open logical terms and proof objects

4. The intermediate language

In this section we present our intermediate language in detail. The syntax for this language is presented in figure 4. Before we explain the types and expressions of this language, we will first give an account of the different contexts we use. As we have said, our language needs to handle logical terms and proof objects that mention free logical variables (open terms). We therefore distinguish between two different classes of logical variables: normal bound variables, that will eventually be instantiated to actual logical terms, and free variables that will never be instantiated. Free variables occupy the F context, which is handled in an ordered fashion throughout our type system. This context also permits use of polymorphic context variables inside it, that eventually get substituted for another F'.

We define an extended notion of kinds in order to classify logical terms that depend on a specific free variables context F; these extended kinds are of the form $[F]\mathcal{K}$, and are denoted as $\widetilde{\mathcal{K}}$. Similarly, extended logical terms $\widetilde{t}$ exist to classify open proof objects; these terms can also mention the free variables of the specified context. We overload this notation to denote extended proof objects $\widetilde{M}$

that come packaged with the context of free variables they depend on. Also, we need to extend our contexts for normal bound variables Φ and Ψ so that they use the extended notions of kinds and logical terms. In this way, we can track what free variables a term that might be substituted inside them might have.

Returning to the syntax of the language, the syntactic category σ corresponds to computational types; we include types for functions, sum and product types and mutable references. The new types are the following: $\mathsf{prf}(\widetilde{t})$ represents extended proof objects lifted in the computational language; $\Pi u : \widetilde{\mathcal{K}}.\sigma$ represents functions over extended logical terms of extended kind $\mathcal{K}$, returning a result of type σ; similarly $\Sigma u : [F]\mathcal{K}.\sigma$ represent existential packages of an extended logical term along with a value of type σ; last, $\Pi\phi.\sigma$ represents quantification over free variables contexts.

Expressions of this language include the standard introduction and elimination forms for such types. Some expressions that are a bit more complicated will be explained once we have shown the typing judgements. We will just mention that the $\mathsf{ltermcase}$ construct lets us perform pattern matching on propositions and proof objects. The available patterns let us check whether the normal form of a logical term is a logical implication, a universal quantification, any specific logical term, an application of one term to another or an arbitrary term (using unification variables that are to be used linearly in the patterns), or any combination thereof. Logical terms in such patterns accept free variables from the context F, same as the term we are pattern matching against.

The typing judgements for extended proof objects and extended logical terms are shown in figure 5. An extended logical term that explicitly requires a free variables context of F, should have access to two sets of logical variables: one, to exactly those free variables contained in F, and two, to compatible bound variables out of the normal context. A bound variable is considered to be compatible if it requires a prefix of the context F – in this way, we can be sure that we will have an instantiation for the free variables that a term will need if it gets substituted in place of the bound variable. This is the intuitive idea that the rules in figure 5 are trying to grasp. Extended logical terms are typechecked in an environment composed of limiting the context $\widetilde{\Phi}$ to variables compatible with the specified free variables context F, something denoted as $\widetilde{\Phi}@F$, and the variables contained in the F context, which is denoted as $\lfloor F \rfloor$. Note that this process yields a normal environment in the form that the logical framework expects, so the same typing rules that we've shown are used with no modifications. Extended proof objects are handled similarly.

The main typing judgement for the intermediate language is presented in figure 6, while operational semantics are given in figure 8 following the style of [Wright and Felleisen 1994]. Some rules for common types are omitted, since they are entirely standard. We will first make some clarifications before we see some rules in more detail.

The context Ω is populated by polymorphic context variables, introduced through the construct $\Lambda\phi.e$ and eventually substituted for a F. The condition $\Omega \vdash F$ that has been used above merely makes sure that only context variables in Ω get used in F. The context Γ contains computational variables, and the context Σ maps locations to types.

A further clarification is needed with regards to our notation $[F]\mathcal{K}$, $[F]t$ and $[F]M$ that gets used in various places. We consider the $[F]$ part as a binding construct for free variables, and therefore we identify such notations up to alpha equivalence wherever they get used. This means that, for example, types $\mathsf{prf}([x : \mathsf{Nat}]x = x)$ and $\mathsf{prf}([y : \mathsf{Nat}]y = y)$ are identified, as are corresponding expressions like $\langle [x : \mathsf{Nat}]\mathsf{Refl}\, x\rangle$ and $\langle [y : \mathsf{Nat}]\mathsf{Refl}\, y\rangle$.

Substitutions written in the style $e[\widetilde{M}/X]$ are standard – they roughly correspond to the contextual closure of the standard $M'[M/X]$ substitution for proof objects –, with a little additional care in order to make sure that free variables are renamed accordingly. This is done by defining this substitution to be the contextual closure of the rule:

$$\left\langle [F,F']M' \right\rangle [[F]M/X] = \left\langle [F,F'](M'[M/X]) \right\rangle$$

Thus, if the proof object M uses a prefix of the free variables that M' uses, these variables in M should first be alpha-renamed to use the same names as in M' before performing the substitution. Substitutions for logical terms in types and expressions work similarly. Substitutions for context variables are entirely standard, replacing all occurences of a variable ϕ with a new list of free variables F, in a capture- avoiding manner.

The typing rules are mostly standard for dependent type systems, with the exception of the elimination construct for proof objects and existential packages of logical terms. For proof objects, the expected rule would be:

$$\frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma \vdash e : \mathsf{prf}(\widetilde{t}) \qquad \Omega;\widetilde{\Phi};\widetilde{\Psi}, X : \widetilde{t};\Sigma;\Gamma \vdash e' : \sigma'}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma \vdash \mathsf{openprf}\ e\ \mathsf{as}\ X\ \mathsf{in}\ e' : \sigma'}$$

If we inspect the given rule, we will see that in fact the two rules match when $F' = \bullet$. This extra parameter is specified so that the user can choose to have certain free variables converted into bound ones, using the λ-abstractions of the logic language. In this way, explicit substitutions for them can be provided through reuse of the β-reduction rule. We would thus be able to use a proof object that mentions an extra free variable compared to a new proof object we're trying to construct, inside that new proof object. A similar idea is used for opening up existential packages.

An example where this type system succeeds in preventing escape of free variables can be shown when considering the following expression:

$$(\Lambda u : [x : \mathsf{Nat}]\mathsf{Prop}.\, \langle [\bullet]u, \mathsf{unit}\rangle)\ ([x : \mathsf{Nat}]x = x)$$

This should not be well-typed, because if $x = x$ gets substituted for u, the logical term inside the existential package will mention an unknown free variable. The way our type system achieves this is by not permitting us to use the u variable in the logical term of the package, since it requires a larger set of free variables than the one specified in the package.

So far we have not presented the typing judgements and operational semantics for the $\mathsf{ltermcase}$ pattern matching construct. These are shown in figures 7 and 9. Let us explain it in a bit more detail. The logical term and the patterns are checked under the specified free variables context F. Patterns are type-checked using a special typing judgement that outputs the context of unification variables that they use, along with the kind of logical terms they can be matched with. We notice that the context of unification variables is built linearly, so each such variable can only appear once in a pattern. After the patterns are typechecked, their unification variables become available when checking the body of the respective match; after unification, the terms that might get substituted for them will depend on the same free variables that the original term t depends on. The unification variables should not escape into the type of the overall match expression. The last premise of the rule requires that the pattern matching is always exhaustive, since the last pattern must be a catch-all pattern. Out of the pattern typing rules, the quantification case is interesting: it requires that the body of a quantified proposition is matched with a higher-order pattern. A typical use would be the following:

$$\begin{array}{l}\Lambda P : [F]\mathsf{Prop}.[F].\mathsf{ltermcase}\ P\ \mathsf{with}\\ \quad (\forall[\mathsf{Nat}].(?P' : \mathsf{Nat} \to \mathsf{Prop}) \mapsto \langle [F, u : \mathsf{Nat}]P'\ u, \mathsf{unit}\rangle)\\ : \Pi P : [F]\mathsf{Prop}.\Sigma P : [F, u : \mathsf{Nat}]\mathsf{Prop}.1\end{array}$$

$$\frac{\Omega;\widetilde{\Phi};\widetilde{\Psi}\vdash \widetilde{M}:\widetilde{t}}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \left\langle \widetilde{M}\right\rangle : \mathsf{prf}(\widetilde{t})} \qquad \frac{\Omega;\widetilde{\Phi}\vdash \widetilde{t}:\widetilde{\mathcal{K}} \qquad \Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma[\widetilde{t}/u]}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \langle \widetilde{t},e\rangle : \Sigma u:\widetilde{\mathcal{K}}.\sigma}$$

$$\frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\mathsf{prf}([F,F']t) \qquad F'=u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n \qquad \Omega;\widetilde{\Phi};\widetilde{\Psi},X:[F](\forall u_1:\mathcal{K}_1.\cdots.\forall u_n:\mathcal{K}_n.t);\Sigma;\Gamma\vdash e':\sigma'}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \mathsf{openprf}\ (e\ \mathsf{binding}\ F')\ \mathsf{as}\ X\ \mathsf{in}\ e':\sigma'}$$

$$\frac{\begin{array}{c}\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\Sigma u:[F,F']\mathcal{K}.\sigma \\ F'=u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n \qquad \Omega;\widetilde{\Phi},u':[F](\mathcal{K}_1\to\cdots\to\mathcal{K}_n\to\mathcal{K});\widetilde{\Psi};\Sigma;\Gamma,x:\sigma[([F,F'](u'\ u_1\ \cdots\ u_n))/u]\vdash e':\sigma' \qquad u'\notin fv(\sigma')\end{array}}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \mathsf{open}\ (e\ \mathsf{binding}\ F')\ \mathsf{as}\ \langle u',x\rangle\ \mathsf{in}\ e':\sigma'}$$

$$\frac{\Omega;\widetilde{\Phi},u:\widetilde{\mathcal{K}};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \Lambda u:\widetilde{\mathcal{K}}.e:\Pi u:\widetilde{\mathcal{K}}.\sigma} \qquad \frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\Pi u:\widetilde{\mathcal{K}}.\sigma \qquad \Omega;\widetilde{\Phi}\vdash \widetilde{t}:\widetilde{\mathcal{K}}}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e\ \widetilde{t}:\sigma[\widetilde{t}/u]} \qquad \frac{\Omega,\phi:\mathsf{ctx};\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \Lambda\phi.e:\Pi\phi.\sigma}$$

$$\frac{\Omega\vdash F \qquad \Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\Pi\phi.\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e\ [F]:\sigma[F/\phi]} \qquad \frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma \qquad \sigma=_{\beta\iota}\sigma'}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma'} \qquad \frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma,x:\sigma\vdash e:\sigma'}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \lambda x:\sigma.e:\sigma\to\sigma'}$$

$$\frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma\to\sigma' \qquad \Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e':\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e\ e':\sigma'} \qquad \frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma,x:\sigma\vdash e:\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \mathsf{fix}\ x:\sigma.e:\sigma} \qquad \frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash \mathsf{ref}\ e:\mathsf{ref}\ \sigma}$$

$$\frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\mathsf{ref}\ \sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash !e:\sigma} \qquad \frac{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:\mathsf{ref}\ \sigma \qquad \Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e':\sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash e:=e':1} \qquad \frac{l:\sigma\in\Sigma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash l:\mathsf{ref}\ \sigma} \qquad \frac{x:\sigma\in\Gamma}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash x:\sigma}$$

Figure 6. Typing judgement of the intermediate language (selected rules)

$$\frac{\Omega;\widetilde{\Phi}\vdash [F]t:[F]\mathcal{K} \qquad \forall i:\Big(\ \widetilde{\Phi}@F,\lfloor F\rfloor\vdash p_i:\Phi_i\succ\mathcal{K} \quad \Omega;\widetilde{\Phi},[F]\Phi_i;\widetilde{\Psi};\Sigma;\Gamma\vdash e_i:\sigma[([F]\{p_i\})/u] \quad \Phi_i\not\subseteq fv(\sigma)\ \Big) \qquad p_n=?u':\mathcal{K}}{\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma\vdash [F].\mathsf{ltermcase}\ t\ \mathsf{with}\ \ (p_1\mapsto e_1)\ \cdots\ (p_n\mapsto e_n):\sigma[([F]t)/u]}$$

$$\{?u:\mathcal{K}\}=u \qquad \{p_1\Rightarrow p_2\}=\{p_1\}\Rightarrow\{p_2\} \qquad \{\forall[\mathcal{K}].p\}=\forall u:\mathcal{K}.p\ u,\text{with } u\notin fv(p) \qquad \{t\}=t \qquad \{p_1\ p_2\}=\{p_1\}\ \{p_2\}$$

$$\frac{}{\Phi\vdash (?u:\mathcal{K}):\bullet,u:\mathcal{K}\succ\mathcal{K}} \qquad \frac{\Phi\vdash p_1:\Phi_1\succ\mathsf{Prop} \qquad \Phi\vdash p_2:\Phi_2\succ\mathsf{Prop} \qquad fv(\Phi_1)\cap fv(\Phi_2)=\emptyset}{\Phi\vdash p_1\Rightarrow p_2:\Phi_1,\Phi_2\succ\mathsf{Prop}} \qquad \frac{\Phi\vdash p:\Phi'\succ\mathcal{K}\to\mathsf{Prop}}{\Phi\vdash\forall[\mathcal{K}].p:\Phi'\succ\mathsf{Prop}}$$

$$\frac{\Phi\vdash t:\mathcal{K}}{\Phi\vdash t:\bullet\succ\mathcal{K}} \qquad \frac{\Phi\vdash p_1:\Phi_1\succ\mathcal{K}\to\mathcal{K}' \qquad \Phi\vdash p_2:\Phi_2\succ\mathcal{K} \qquad fv(\Phi_1)\cap fv(\Phi_2)=\emptyset}{\Phi\vdash p_1\ p_2:\Phi_1,\Phi_2\succ\mathcal{K}'}$$

$$\text{where: } [F]\Phi=\begin{cases}\bullet & \text{if } \Phi=\bullet\\ [F]\Phi',u:[F]\mathcal{K} & \text{if } \Phi=\Phi',u:\mathcal{K}\end{cases}$$

Figure 7. Typing for pattern match construct

$$\begin{array}{rcl} v & ::= & \left\langle \widetilde{M}\right\rangle \mid \Lambda u:\widetilde{\mathcal{K}}.e \mid \langle \widetilde{t},v\rangle \mid \Lambda\phi.e \mid \lambda x:\sigma.e \mid (v_1,\ v_2) \mid \mathsf{inj}_i\ v \mid \mathsf{unit} \mid l \\ \mathcal{E} & ::= & \mathsf{openprf}\ (\mathcal{E}\ \mathsf{binding}\ F)\ \mathsf{as}\ X\ \mathsf{in}\ e' \mid \mathcal{E}\ \widetilde{t} \mid \langle \widetilde{t},\mathcal{E}\rangle \mid \mathsf{open}\ (\mathcal{E}\ \mathsf{binding}\ F)\ \mathsf{as}\ \langle u,x\rangle\ \mathsf{in}\ e' \mid \mathcal{E}\ [F] \mid \mathcal{E}\ e_2 \mid v_1\ \mathcal{E} \mid (\mathcal{E},\ e_2) \mid (v_1,\ \mathcal{E}) \mid \mathsf{proj}_i\ \mathcal{E} \mid \mathsf{inj}_i\ \mathcal{E} \\ & & \mid \mathsf{case}(\mathcal{E},x.e_1,x.e_2) \mid \mathsf{ref}\ \mathcal{E} \mid \mathcal{E}:=e' \mid v_1:=\mathcal{E} \mid\ !\mathcal{E} \\ \mu & ::= & \bullet \mid \mu,l\mapsto v \end{array}$$

$$\mu,\mathsf{openprf}\ (\langle [F,u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n]M\rangle\ \mathsf{binding}\ u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n)\ \mathsf{as}\ X\ \mathsf{in}\ e'\longrightarrow\mu,e'[[F](\lambda u_1:\mathcal{K}_1.\cdots\lambda u_n:\mathcal{K}_n.M)/X]$$

$$\mu,\mathsf{open}\ (\langle [F,u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n]t,v\rangle\ \mathsf{binding}\ u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n)\ \mathsf{as}\ \langle u,x\rangle\ \mathsf{in}\ e'\longrightarrow\mu,e'[[F](\lambda u_1:\mathcal{K}_1.\cdots\lambda u_n:\mathcal{K}_n.t)/u][v/x]$$

$$\mu,(\Lambda u:\widetilde{\mathcal{K}}.e)\ \widetilde{t}\longrightarrow\mu,e[\widetilde{t}/u] \qquad \mu,(\Lambda\phi.e)\ [F]\longrightarrow\mu,e[F/\phi] \qquad \mu,(\lambda x:\sigma.e)\ v\longrightarrow\mu,e[v/x] \qquad \mu,\mathsf{fix}\ x:\sigma.e\longrightarrow\mu,e[\mathsf{fix}\ x:\sigma.e/x] \qquad \mu,\mathsf{proj}_i\ (v_1,\ v_2)\longrightarrow\mu,v_i$$

$$\mu,\mathsf{case}(\mathsf{inj}_i\ v,x.e_1,x.e_2)\longrightarrow\mu,e_i[v/x] \qquad \mu,\mathsf{ref}\ v\longrightarrow\mu\uplus l\mapsto v,l \qquad \mu,!l\longrightarrow\mu,\mu(l) \qquad \mu,l:=v\longrightarrow\mu\big|l\mapsto v,\mathsf{unit} \qquad \frac{\mu,e\longrightarrow\mu',e'}{\mu,\mathcal{E}[e]\longrightarrow\mu',\mathcal{E}[e']}$$

Figure 8. Operational semantics for the intermediate language

$$\frac{\langle\!\langle p_0|t\rangle\!\rangle \rightsquigarrow \mathsf{fail}}{\mu,[F].\mathsf{ltermcase}\ t\ \mathsf{with}\ \ p_0 \mapsto e_0, \overrightarrow{p \mapsto e} \longrightarrow \mu,[F].\mathsf{ltermcase}\ t\ \mathsf{with}\ \overrightarrow{p \mapsto e}}$$

$$\frac{\langle\!\langle p_0|t\rangle\!\rangle \rightsquigarrow (u_1 = t_1, \cdots, u_n = t_n)}{\mu,[F].\mathsf{ltermcase}\ t\ \mathsf{with}\ \ p_0 \mapsto e_0, \overrightarrow{p \mapsto e} \longrightarrow \mu, e_0[[F]t_1/u_1]\cdots[[F]t_n/u_n]} \qquad \frac{}{\langle\!\langle ?u : \mathcal{K}|t\rangle\!\rangle \rightsquigarrow u = t}$$

$$\frac{t \rightarrow^*_{\beta\iota} t_1 \Rightarrow t_2 \qquad \langle\!\langle p_1|t_1\rangle\!\rangle \rightsquigarrow \theta_1 \qquad \langle\!\langle p_2|t_2\rangle\!\rangle \rightsquigarrow \theta_2}{\langle\!\langle p_1 \Rightarrow p_2|t\rangle\!\rangle \rightsquigarrow \theta_1, \theta_2} \qquad \frac{t \rightarrow^*_{\beta\iota} \forall x : \mathcal{K}.t' \qquad \langle\!\langle p|\lambda x : \mathcal{K}.t'\rangle\!\rangle \rightsquigarrow \theta}{\langle\!\langle \forall[\mathcal{K}].p|t\rangle\!\rangle \rightsquigarrow \theta} \qquad \frac{t =_{\beta\iota} t'}{\langle\!\langle t|t'\rangle\!\rangle \rightsquigarrow \bullet}$$

$$\frac{t \rightarrow^*_{\beta\iota} t_1\ t_2 \qquad \langle\!\langle p_1|t_1\rangle\!\rangle \rightsquigarrow \theta_1 \qquad \langle\!\langle p_2|t_2\rangle\!\rangle \rightsquigarrow \theta_2}{\langle\!\langle p_1\ p_2|t_1\ t_2\rangle\!\rangle \rightsquigarrow \theta_1, \theta_2} \qquad \frac{\text{none of the above rules apply}}{\langle\!\langle p|t\rangle\!\rangle \rightsquigarrow \mathsf{fail}} \qquad \text{where } \theta \text{ is in the form of } u_1 = t_1, \cdots, u_n = t_n$$

Figure 9. Operational semantics for pattern match construct

This involves using a unification variable of the kind $\mathcal{K} \rightarrow \mathsf{Prop}$ to match the body of the quantifier, and boxing this variable in an extended context to get the proposition with an extra free variable that the body of the quantifier corresponds to.

The operational semantics for this construct proceed by trying to match each pattern with the logical term in the order they are given, and stopping at the first successful matching. A set of rules that attempt unification of a pattern with a logical term is given; it is interesting to note that logical terms are converted into $\beta\iota$-normal form before the actual unification is attempted. In practice, using their weak-head normal form is enough.

This concludes our presentation of the intermediate language. We have studied the metatheory for this language, and have found it relatively straightforward using standard techniques. Preservation depends on a number of substitution lemmas like the following:

Lemma 4.1 (Substitution of ext. proof objects into expressions) *If* $\Omega;\widetilde{\Phi};\widetilde{\Psi},X:\widetilde{\tau};\Sigma;\Gamma \vdash e:\sigma$ *and* $\Omega;\widetilde{\Phi};\widetilde{\Psi} \vdash \widetilde{M}:\widetilde{\tau}$ *, then* $\Omega;\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma \vdash e[\widetilde{M}/X]:\sigma$.

Lemma 4.2 (Substitution of ext. logical terms into expressions) *If* $\Omega;\widetilde{\Phi},u:\widetilde{\mathcal{K}};\widetilde{\Psi};\Sigma;\Gamma \vdash e:\sigma$ *and* $\Omega;\widetilde{\Phi} \vdash \widetilde{t}:\widetilde{\mathcal{K}}$ *, then* $\Omega;\widetilde{\Phi};\widetilde{\Psi}[\widetilde{t}/u];\Sigma[\widetilde{t}/u];\Gamma[\widetilde{t}/u] \vdash e[\widetilde{t}/u]:\sigma[\widetilde{t}/u]$.

Lemma 4.3 (Substitution of contexts into expressions) *If* $\Omega,\phi:\mathit{ctx};\widetilde{\Phi};\widetilde{\Psi};\Sigma;\Gamma \vdash e:\sigma$ *and* $\Omega \vdash F$ *then* $\Omega;\widetilde{\Phi}[F/\phi];\widetilde{\Psi}[F/\phi];\Sigma[F/\phi];\Gamma[F/\phi] \vdash e[F/\phi]:\sigma[F/\phi]$.

A lemma detailing the correct behavior of pattern unification with respect to pattern typing is also needed:

Lemma 4.4 *If* $\langle\!\langle p|t\rangle\!\rangle \rightsquigarrow \theta$, $\Phi \vdash p:\Phi' \succ \mathcal{K}$, $\Phi \vdash t:\mathcal{K}$ *and* $\Phi' = u_1:\mathcal{K}_1,\cdots,u_n:\mathcal{K}_n$ *, then* $\theta = (u_1 = t_1,\cdots,u_n = t_n)$ *, with* $\Phi \vdash t_i:\mathcal{K}_i$ *and* $\{p\}[t_1/u_1]\cdots[t_n/u_n] =_{\beta\iota} t$.

Theorem 4.5 (Preservation) *If* $\bullet;\bullet;\bullet;\Sigma;\bullet \vdash e:\sigma$, $\bullet;\bullet;\bullet;\Sigma;\bullet \vdash \mu$ *and* $\mu,e \longrightarrow \mu',e'$ *then for some* $\Sigma' \supseteq \Sigma$, $\bullet;\bullet;\bullet;\Sigma';\bullet \vdash e':\sigma$ *and* $\bullet;\bullet;\bullet;\Sigma';\bullet \vdash \mu'$.

Proof by structural induction on the step relation $\mu,e \longrightarrow \mu',e'$, made relatively simple by use of the above lemmas. The most interesting case is the existential package elimination rule; the key step is noticing that the type of x becomes $\beta\iota$-equivalent to the type of the contained value v after performing the substitution of $\lambda u_1:\mathcal{K}_1.\cdots\lambda u_n:\mathcal{K}.n.t$ for u. 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 4.6 (Canonical forms) *If* $\bullet;\bullet;\bullet;\Sigma;\bullet \vdash v:\sigma$ *then:*
If $\sigma = \mathit{prf}([F]t)$*, then* $v = \langle[F]M\rangle$ *and* $\lfloor F\rfloor;\bullet \vdash M:t$.
If $\sigma = \Sigma u:[F]\mathcal{K}.\sigma'$*, then* $v = \langle[F]t,v'\rangle$ *and* $\lfloor F\rfloor \vdash t:\mathcal{K}$.
If $\sigma = \Pi u:[F]\mathcal{K}.\sigma'$*, then* $v = \Lambda u:[F]\mathcal{K}e$.
If $\sigma = \Pi\phi.\sigma'$*, then* $v = \Lambda\phi.e$.

Theorem 4.7 (Progress) *If* $\bullet;\bullet;\bullet;\Sigma;\bullet \vdash e:\sigma$ *then either* e *is a value or there exists* e' *and* μ' *such that* $\mu,e \longrightarrow \mu',e'$ *for every* μ *such that* $\bullet;\bullet;\bullet;\Sigma;\bullet \vdash \mu$.

The proof is a straightforward structural induction on the typing derivation of e.

More details about the proofs can be found in the extended version of this paper [Stampoulis and Shao 2009]. We have found these proofs to be relatively orthogonal to proofs about the type safety of the basic constructs of the computational language. In our previous attempts to arriving a type system for our language, we have made use of an ambient context F that was part of the typing judgements. Free variables were drawn from that context, which in turn required some dependence between the types we assign and this ambient context. This complicated the proofs substantially, and often made it impossible to support at the same time all the desired features. Our key insight from our current solution is that orthogonality can best be insured by having dependence on free variables be locally closed.

From type safety we immediately get the property that if an expression evaluates to a lifted proof object, then the contained proof object 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 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.

In cases where strong guarantees about validity of the produced proof objects are not needed, we can actually skip computing the full forms of the proof objects. By inspection of the operational semantics of the language it is apparent that the specific values of the proof objects are computationally irrelevant, since we never pattern match against them. We can therefore prove that a "proof-erasure" semantics corresponds to the semantics that we have given so far.

5. The surface language

The intermediate language we presented in the previous section is relatively complex, and so far we haven't arrived at a simpler type

$$
\begin{array}{rcl}
\sigma & ::= & \mathsf{prf}(\widetilde{t}) \mid \Pi u : \widetilde{\mathcal{K}}.\sigma \mid \Sigma u : \widetilde{\mathcal{K}}.\sigma \mid \Pi\phi \ll F.\sigma \mid \cdots \\
\varepsilon & ::= & \langle M \rangle \mid \mathsf{openprf}\ \varepsilon\ \mathsf{as}\ X\ \mathsf{in}\ \varepsilon' \mid \Lambda u : \mathcal{K}.\varepsilon \mid \varepsilon\ t \\
& & \mid \langle t, \varepsilon \rangle \mid \mathsf{open}\ \varepsilon\ \mathsf{as}\ \langle u, x \rangle\ \mathsf{in}\ \varepsilon' \mid \nu u : \mathcal{K}.\varepsilon \mid \nu\phi.\varepsilon \\
& & \mid \varepsilon\ [\mathsf{curctx}] \mid \mathsf{ltermcase}\ t\ \mathsf{with}\ \overrightarrow{p \mapsto \varepsilon} \mid \cdots
\end{array}
$$

Figure 10. Syntax for the surface language

system for which the requirement of dealing with open terms in a type-safe manner is met. Programmers using this language would have to explicitly mention full free variables contexts whenever a proof object or logical term is involved, and also keep track of which variables of the $\widetilde{\Phi}$ and $\widetilde{\Psi}$ contexts are actually available for use. We believe that this is a relatively large programming overhead, and in this section we aim to reduce it by defining a surface language with a simpler interface to the programmer. This surface language will be typed, and well-typed terms in it will always be translatable to well-typed terms in the intermediate language.

The main idea of this language is to manage an ambient F context of free variables, variables which all proof objects and logical terms that we introduce can mention, so there is no need to explicitly box them. Dually, we maintain the property that all proof objects and logical terms that we can access only mention variables in that context, so contexts of the form of $\widetilde{\Phi}$ and $\widetilde{\Psi}$ are not required, helping to avoid confusion. We have an explicit construct to introduce a new free variable in the ambient context, that is modeled after Delphin's ν-construct. This free variable can only be used within the scope of this construct, something guaranteed by the type system. Last, there is a similar construct to introduce a set of free variables, that corresponds to the support for context polymorphism of the intermediate language.

The syntax for this language is given in figure 10. The types for this language are for the most part identical to the ones of the intermediate language; the only difference is that the context product type needs to keep information about the free variables context where it was introduced, information that gets erased during translation to the intermediate language. The expressions are also similar, with the changes discussed above. Note that $\varepsilon\ [\mathsf{curctx}]$ represents the equivalent of the elimination construct for context polymorphism, and it too does not need to mention explicitly a context. We give the typing rules for the surface language in figure 11. For standard types the rules are identical to the ones for the intermediate language (save for the different set of contexts). The judgements are of the form $F;\Phi;\Psi;\Gamma \vdash \varepsilon : \sigma$, where the new context F is the ambient free variables context that we have mentioned. This context includes all the information that was stored in the Ω context of the previous language, so that context is retired here. Also, the store typing context Σ is not used, since this language is not directly evaluated and therefore does not need to include locations.

The two rules describing the elimination of lifted proof objects deserve special mention. The distinction between the two is that the first gets used when the free variables that the proof object depends on exist in the ambient free variables context; the second only gets used when the proof object uses some extra free variables. In that case, an explicit substitution for them should be provided, so the solution of converting them into bound variables through the logic's λ-abstraction is used, as in the intermediate language.

In figure 12 we give the translation of surface expressions into intermediate expressions. This is a normal type-directed translation, so it is only to be used with well-typed terms; we have used a different style rather than the standard one for presentation purposes. The basic idea of the translation is to use the current ambient context of free variables in order to fill-in the missing information compared to the intermediate language; for example, introducing a proof object is always translated into the equivalent operation of boxing it with the current ambient context.

We have proved the following theorem for this translation, that shows that well-typed surface-level terms can always be translated into well-typed terms of the intermediate language.

Theorem 5.1 (Validity of translation) *Given a derivation $F;\Phi;\Psi;\Gamma \vdash \varepsilon : \sigma$, the derivation yielded by the type-directed translation, $[\![F;\Phi;\Psi;\Gamma \vdash \varepsilon : \sigma]\!]$, is valid.*

Details about the proof can be found in the extended version [Stampoulis and Shao 2009].

Thus we can directly use the simpler language described in this section and its simpler set of typing rules; using the above theorem and the type safety theorem of the intermediate language we are guaranteed that the composition of the translation function with the operational semantics of the intermediate language results in an evaluation function for which type safety holds. Of course the values yielded by such an evaluation will belong to the intermediate language rather than this surface language.

The surface language maintains a key property that makes this translation possible: the free variables available at any expression can not decrease as we look further inside the structure of that expression. This is in contrast with the intermediate language, where the available free variables are specified at the point where they are needed and thus no such limitation exists. Due to this property, the free variables that a lifted proof object or a packaged logical term require will always be available in the body of the construct we use to open them up. Another property that is maintained is that elimination constructs are only usable for packaged proof objects and logical terms that depend on compatible contexts with the ambient one. For example, if the ambient context is $x : \mathsf{Nat}, y : \mathsf{Nat}$, then a proof object (which might have been introduced through a lambda abstraction) depending on a context of $x : \mathsf{List}$ will not be able to be opened up. In this way, terms that will not be usable in the ambient context and its extensions never enter the variable contexts Φ and Ψ. Together these properties explain why dependence on free variables doesn't need to be tracked in the Φ and Ψ contexts we use here.

These properties give us an intuitive understanding of the limitations of the surface language compared to the intermediate one: if they are enforced by an intermediate level expression, then the inverse transformation to a surface level expression should be possible.

6. Examples

In this section we will present a number of programming examples in the surface language, in order to demonstrate how its constructs are used. The programming examples take the form of simple tactics that one would have to implement in order to build a proof assistant for the logical framework we have described.

First, we will present a tactic that proves goals that look like $x =_{\mathsf{Nat}} x$, similar to the reflexivity tactic used in Coq.

$$
\begin{array}{l}
\mathsf{reflexivity} : \Pi\phi \ll \bullet.\Pi P : [\phi]\mathsf{Prop.option}\ \mathsf{prf}([\phi]P) = \\
\quad \nu\phi.\Lambda P : \mathsf{Prop.ltermcase}\ P\ \mathsf{with} \\
\qquad =_{\mathsf{Nat}}\ (?a : \mathsf{Nat})\ (?b : \mathsf{Nat}) \mapsto \mathsf{ltermcase}\ b\ \mathsf{with} \\
\qquad\quad a \mapsto \mathsf{Just}\ \langle \mathsf{Refl}\ a \rangle \\
\qquad\quad \mid (?b : \mathsf{Nat}) \mapsto \mathsf{None} \\
\qquad \mid (?P : \mathsf{Prop}) \mapsto \mathsf{None}
\end{array}
$$

This example demonstrates the use of the pattern matching operator, and also shows that even though only linear unification

$$\frac{\Phi, \lfloor F \rfloor; \Psi \vdash M : t}{F; \Phi; \Psi; \Gamma \vdash \langle M \rangle : \mathsf{prf}([F]t)} \qquad \frac{F, F'; \Phi; \Psi; \Gamma \vdash \varepsilon : \mathsf{prf}([F]t) \qquad F, F'; \Phi; \Psi, X : t; \Gamma \vdash \varepsilon' : \sigma'}{F, F'; \Phi; \Psi; \Gamma \vdash \mathsf{openprf}\ \varepsilon\ \mathsf{as}\ X\ \mathsf{in}\ \varepsilon' : \sigma'}$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \mathsf{prf}([F, F']t) \qquad F' = u_1 : \mathcal{K}_1, \cdots, u_n : \mathcal{K}_n \qquad F; \Phi; \Psi, X : \forall u_1 : \mathcal{K}_1. \cdots \forall u_n : \mathcal{K}_n.t; \Gamma \vdash \varepsilon' : \sigma'}{F; \Phi; \Psi; \Gamma \vdash \mathsf{openprf}\ \varepsilon\ \mathsf{as}\ X\ \mathsf{in}\ \varepsilon' : \sigma'} \qquad \frac{F, u : \mathcal{K}; \Phi; \Psi; \Gamma \vdash \varepsilon : \sigma}{F; \Phi; \Psi; \Gamma \vdash \nu u : \mathcal{K}.\varepsilon : \sigma}$$

$$\frac{F, \phi; \Phi; \Psi; \Gamma \vdash \varepsilon : \sigma \qquad \phi \notin F}{F; \Phi; \Psi; \Gamma \vdash \nu\phi.\varepsilon : \Pi\phi \ll F.\sigma} \qquad \frac{F, F'; \Phi; \Psi; \Gamma \vdash \varepsilon : \Pi\phi \ll F.\sigma}{F, F'; \Phi; \Psi; \Gamma \vdash \varepsilon\ [\mathsf{curctx}] : \sigma[F'/\phi]} \qquad \frac{F; \Phi, u : \mathcal{K}; \Psi; \Gamma \vdash \varepsilon : \sigma}{F; \Phi; \Psi; \Gamma \vdash \Lambda u : \mathcal{K}.\varepsilon : \Pi u : [F]\mathcal{K}.\sigma}$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \Pi u : [F, F']\mathcal{K}.\sigma \qquad F' = u_1 : \mathcal{K}_1, \cdots, u_n : \mathcal{K}_n \qquad \Phi, \lfloor F \rfloor \vdash t : \mathcal{K}}{F; \Phi; \Psi; \Gamma \vdash \varepsilon\ t : \sigma[[F, F']t/u]} \qquad \frac{\Phi, \lfloor F \rfloor \vdash t : \mathcal{K} \qquad F; \Phi; \Psi; \Gamma \vdash \varepsilon : \sigma[[F]t/u]}{F; \Phi; \Psi; \Gamma \vdash \langle t, \varepsilon \rangle : \Sigma u : [F]\mathcal{K}.\sigma}$$

$$\frac{F, F'; \Phi; \Psi; \Gamma \vdash \varepsilon : \Sigma u : [F]\mathcal{K}.\sigma \qquad F, F'; \Phi, u : \mathcal{K}; \Psi; \Gamma, x : \sigma \vdash \varepsilon' : \sigma' \qquad u \notin fv(\sigma')}{F, F'; \Phi; \Psi; \Gamma \vdash \mathsf{open}\ \varepsilon\ \mathsf{as}\ \langle u, x \rangle\ \mathsf{in}\ \varepsilon' : \sigma'}$$

$$\frac{\begin{array}{c} F; \Phi; \Psi; \Gamma \vdash \varepsilon : \Sigma u : [F, F']\mathcal{K}.\sigma \\ F' = u_1 : \mathcal{K}_1, \cdots, u_n : \mathcal{K}_n \qquad F; \Phi, u' : \mathcal{K}_1 \to \cdots \to \mathcal{K}_n \to \mathcal{K}; \Psi; \Gamma, x : \sigma[[F, F']u'\ u_1\ \cdots\ u_n/u] \vdash \varepsilon' : \sigma' \qquad u' \notin fv(\sigma') \end{array}}{F; \Phi; \Psi; \Gamma \vdash \mathsf{open}\ \varepsilon\ \mathsf{as}\ \langle u', x \rangle\ \mathsf{in}\ \varepsilon' : \sigma'}$$

$$\frac{\Phi, \lfloor F \rfloor \vdash t : \mathcal{K} \qquad \forall i : \big(\ \Phi, \lfloor F \rfloor \vdash p_i : \Phi_i \succ \mathcal{K} \quad F; \Phi, \Phi_i; \Psi; \Gamma \vdash \varepsilon_i : \sigma[[F]\{p_i\}/u] \quad \Phi_i \notin fv(\sigma)\ \big) \qquad p_n = ?u' : \mathcal{K}}{F; \Phi; \Psi; \Gamma \vdash \mathsf{ltermcase}\ t\ \mathsf{with}\ \overrightarrow{(p_i \mapsto \varepsilon_i)} : \sigma[[F]t/u]}$$

Figure 11. Selected typing rules for the surface language

$$[\![\langle M \rangle]\!]_F = \langle [F]M \rangle$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \mathsf{prf}([F']t) \text{ with } F = F', F''}{[\![\mathsf{openprf}\ \varepsilon\ \mathsf{as}\ X\ \mathsf{in}\ \varepsilon']\!]_F = \mathsf{openprf}\ ([\![\varepsilon]\!]_F\ \mathsf{binding}\ \bullet)\ \mathsf{as}\ X\ \mathsf{in}\ [\![\varepsilon']\!]_F}$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \mathsf{prf}([F, F']t)}{[\![\mathsf{openprf}\ \varepsilon\ \mathsf{as}\ X\ \mathsf{in}\ \varepsilon']\!]_F = \mathsf{openprf}\ ([\![\varepsilon]\!]_F\ \mathsf{binding}\ F')\ \mathsf{as}\ X\ \mathsf{in}\ [\![\varepsilon']\!]_F}$$

$$[\![\nu u : \mathcal{K}.\varepsilon]\!]_F = [\![\varepsilon]\!]_{F, u:\mathcal{K}} \qquad [\![\nu\phi.\varepsilon]\!]_F = \Lambda\phi : \mathsf{ctx}.\ [\![\varepsilon]\!]_{F,\phi}$$

$$\frac{F, F'; \Phi; \Psi; \Gamma \vdash \varepsilon : \Pi\phi \ll F.\sigma}{[\![\varepsilon\ [\mathsf{curctx}]]\!]_{F,F'} = [\![\varepsilon]\!]_{F,F'}\ [F']} \qquad [\![\Lambda u : \mathcal{K}.\varepsilon]\!]_F = \Lambda u : [F]\mathcal{K}.\ [\![\varepsilon]\!]_F$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \Pi u : [F, F']\mathcal{K}.\sigma}{[\![\varepsilon\ t]\!]_F = [\![\varepsilon]\!]_F\ [F, F']t} \qquad [\![\langle t, \varepsilon \rangle]\!]_F = \langle [F]t, [\![\varepsilon]\!]_F \rangle$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \Sigma u : [F']\mathcal{K}.\sigma \text{ with } F = F', F''}{[\![\mathsf{open}\ \varepsilon\ \mathsf{as}\ \langle u', x \rangle\ \mathsf{in}\ \varepsilon']\!]_F = \mathsf{open}\ ([\![\varepsilon]\!]_F\ \mathsf{binding}\ \bullet)\ \mathsf{as}\ \langle u', x \rangle\ \mathsf{in}\ [\![\varepsilon']\!]_F}$$

$$\frac{F; \Phi; \Psi; \Gamma \vdash \varepsilon : \Sigma u : [F, F']\mathcal{K}.\sigma}{[\![\mathsf{open}\ \varepsilon\ \mathsf{as}\ \langle u', x \rangle\ \mathsf{in}\ \varepsilon']\!]_F = \mathsf{open}\ ([\![\varepsilon]\!]_F\ \mathsf{binding}\ F')\ \mathsf{as}\ \langle u', x \rangle\ \mathsf{in}\ [\![\varepsilon']\!]_F}$$

$$[\![x]\!]_F = x$$

Figure 12. Selected translation rules for the surface language

is allowed for our patterns, non-linear unification can be achieved as well by nested pattern matches.

The type of this function can be understood as the type of tactics that can directly prove certain goals, which we abbreviate as prfTac. We can write combinator functions that compose such tactics in order to produce more complicated ones; such combinators are called tacticals in the LCF terminology. A very simple such example is a tactical that tries to use two tactics sequentially:

$$\mathsf{ifFail} : \mathsf{prfTac} \to \mathsf{prfTac} \to \mathsf{prfTac} = \\ \lambda f_1.\lambda f_2.\nu\phi.\Lambda P : \mathsf{Prop.match}\ f_1\ P\ \mathsf{with}\ \mathsf{None} \mapsto f_2\ P \mid x \mapsto x$$

A more interesting tactical is one that generalizes a local tactic as reflexivity into a global tactic which can prove goals that are in the contextual closure of the goals handled by the first tactic. This is presented in figure 13. It works by traversing the structure of a proposition, and applying the supplied tactic at all propositions visited. We show some example cases; more connectives should be covered in a real implementation. The most interesting case is the case of the universal quantifier, which demonstrates the handling of open terms. Consider the example where we're pattern matching on the proposition $\forall x : \mathsf{Nat}.x =_{\mathsf{Nat}} x$, our base tactic being reflexivity as defined above. Unification will replace the instances of the variable P in the body with the term $\lambda x : \mathsf{Nat}.x =_{\mathsf{Nat}} x$, which represents the body of the quantification. Now, before we can recursively apply the same function, we have to instantiate the body for a new free variable; this is done using the ν-construct. The recursive call will thus work on the proposition $x =_{\mathsf{Nat}} x$ which can directly be handled by the reflexivity tactic, but the returned proof object, $\langle [\phi, x : \mathsf{Nat}]\mathsf{Refl}\ x \rangle$, depends on an extra free variable. When we open the proof object up, this extra dependence will be internalized into the proof object through a λ-abstraction, yielding the proof object $\langle [\phi]\lambda x : \mathsf{Nat}.\mathsf{Refl}\ x \rangle$, which has a universally quantified type.

Another class of tactics that we could define do not directly prove a goal, but transform it into a proposition that would be simpler to prove. In 13, we show the tactic rewrite, which is used to

perform substitution of a term for another. For example, we could use it to substitute x for 5 in the goal $x + x =_{\mathsf{Nat}} 10$, yielding the transformed goal $5+5 =_{\mathsf{Nat}} 10 \wedge x =_{\mathsf{Nat}} 5$, which would be easier to prove if we had $x =_{\mathsf{Nat}} 5$ as an assumption, since the first part of the conjunction could be easily proved by reflexivity. The core of this function is the rewriteTerm function, which identifies occurrences inside a term of the term x we want to substitute, substitutes them with the term y we want to substitute with, and proves that the resulting term is equal to the original one, provided that $x = y$. This gets used inside the main part of the function, that matches propositions that consist of some predicate applied to terms of the same type as the term we're substituting; the rewriteTerm function is used to do the required substitutions in these arguments and prove that the resulting proposition implies the original one. Note that this function only handles two-place predicates like $=_{\mathsf{Nat}}$ and two-place functions like $+$, but it could be generalized to handle more cases.

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 both other language and framework designs that are similar in spirit or goals to the language design we've described here; as well as work that deals with similar type system issues as the one we have tried to address in this work.

The framework that we feel is closest to our stated goals is the YNot project [Nanevski et al. 2008, Chlipala et al. 2009]. This project attempts to extend the programming model available in the Coq proof assistant with support for effectful features like mutable references and I/O [Wisnesky et al. 2009]; support for concurrency [Nanevski et al. 2007] is also planned. The primary aim is to be able to write certified programs that use these features, exploiting the rich dependently-typed theory underlying the Coq proof assistant and its proof development features for the certification process, and have certified executable code extracted from such developments. Furthermore, the extracted code can use efficient native implementations for the added effectful features. The approach taken to achieve these goals is to axiomatically extend Coq's logic with support for the stateful monad of Hoare Type Theory [Nanevski et al. 2006]; such an extension is deemed to be sound thanks to the extensive metatheory that has been developed for HTT (e.g. [Petersen et al. 2008]). Our primary aim is different: we want to enhance the practice of proof development inside a proof assistant like Coq by giving access to a richer programming model at all the involved levels; therefore we do not consider program extraction or the correctness of such code since the produced proofs can always be validated by a kernel proof checker. Our hope is that in this way richer, domain-specific automation can be built during the proof development process, in order to enhance its scalability. We consider the recent reimplementation of YNot [Chlipala et al. 2009] as exhibiting the sheer benefits that domain-specific tactic implementation yields, and we believe our language design allows for implementation of significantly more complex such tactics. Our overall approach is also very different compared to YNot. Instead of adding support for imperative features directly inside the logic itself, we choose to keep the logic as is and embed it inside a computational language with all the desired imperative features. We have shown that this can be done in an orthogonal way to features of the computational language, and still yield strong static guarantees about the terms of the logic we produce. Using our approach, features can be added to the computational language safely as long as standard type safety is still guaranteed; there is no need to re-evaluate their soundness with respect to the logical framework we use.

It is interesting to contrast our framework with computational languages that deal with terms of the LF logical framework [Pfenning 1991], like Twelf [Pfenning and Schürmann 1999], Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008]. 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 were our logic alludes to it. Also, a large part of the complex metatheory of such a logic would have to be developed inside LF. To the best of our knowledge, 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. Last, even if this encoding was done, the aforementioned computational languages do not provide the imperative constructs that we have considered here.

The framework described in [Licata et al. 2008, Licata and Harper 2009b] tries to reconcile the HOAS approach to binding present in the LF framework with a rich notion of computation; such a framework could be used to provide a more natural encoding of a logic as the one we describe. Unfortunately, this framework has not yet been extended to fully dependent LF terms, even if some work has been done towards that effect in a similar framework that does not include binding [Licata and Harper 2009a]; thus our logic can not be directly represented in this framework. Furthermore, it has yet to be shown how such a framework can be reconciled with effectful constructs like mutable references.

This work, along with Beluga and Delphin, are also interesting to us in that they provide solutions to a similar issue that we have tried to address in our type system, namely type-safe computation with open terms of an object language with binding. LF is such an object language, so this issue naturally arises in the setting of these languages. The approach that we have taken is similarly expressive to the approach taken in Delphin, and less so compared to Beluga and the framework of [Licata and Harper 2009b]; still, it has so far proven to be enough in our setting.

We believe that our approach is novel compared to Beluga due to the following reasons: we show how an intermediate language which is inspired by Beluga can largely avoid the need for explicit substitutions through viewing the free variables context in an ordered way; how providing explicit substitutions for free variables can be recovered when required in the setting of our logic; how the object language that we use does not need to have its typing rules extended to account for context management and explicit substitutions; and how a surface language with a simpler type system can be built to lessen the burden placed on the programmer.

Furthermore, the novelty compared to Delphin is that while retaining a similar programming interface at the surface level, our overall approach maintains type safety in the presence of features like mutable references. The type system of Delphin cannot easily be reconciled with such a feature, essentially because the set of free variables that values inhabiting a type can mention always depends on the current ambient context; but values stored in references should always mention the same set of free variables to avoid free variables escaping their defined scope. Consider the following example:

$$x := \langle \mathsf{True} \rangle ; (\nu a : \mathsf{Nat}.x := \langle (\lambda b : \mathsf{Nat}.\mathsf{True})\ a \rangle); \mathsf{openprf}\ !x \cdots$$

The free variable a has escaped its proper scope. The problem is that the assignment to the reference is allowed, something possible since the proof object can have the same type as the one that was originally assigned to the reference; and this is because the

```
ctxclosure : prfTac → prfTac =
  λf_t.νϕ.ΛP : Prop.ifFail (f_t [curctx] P) (ltermcase P with
      ∧ (?P_1 : Prop) (?P_2 : Prop) ↦  do x_1 ← ctxclosure f_t [curctx] P_1
                                          x_2 ← ctxclosure f_t [curctx] P_2
                                          return (openprf x_1,x_2 as X_1,X_2 in ⟨AndI P_1 P_2 X_1 X_2⟩)
    | (?P_1 : Prop) ⇒ (?P_2 : Prop) ↦  do x_2 ← ctxclosure f_t [curctx] P_2
                                          return (openprf x_2 as X_2 in ⟨λX_1 : P_1.X_2⟩)
    | ∀[K].(?P' : Nat → Prop) ↦  do x_1 :: prf([ϕ,u : Nat]P' x) ← νu : Nat.ctxclosure f_t [curctx] (P' u)
                                    return (openprf x_1 as X_1 :: ∀u : Nat.P'u in ⟨X_1⟩)
    | (?P : Prop) ↦ None)

rewrite : Πϕ ≪ •.Πx : [ϕ]Nat.Πy : [ϕ]Nat.ΠP : [ϕ]Prop.ΣP' : [ϕ]Prop.prf([ϕ]P' ⇒ P) =
  νϕ.Λx : Nat.Λy : Nat.ΛP : Prop.ltermcase P with
      (?P_0 : Nat → Nat → Prop) (?u_1 : Nat) (?u_2 : Nat) ↦  open rewriteTerm [curctx] x y u_1 as ⟨u'_1, ⟨X_1 :: ∀P : P u'_1 ∧ x = y ⇒ P u_1⟩⟩ in
                                                             open rewriteTerm [curctx] x y u_2 as ⟨u'_2, ⟨X_2 :: ∀P : P u'_2 ∧ x = y ⇒ P u_2⟩⟩ in
                                                             ⟨P_0 u'_1 u'_2 ∧ x = y, ⟨proof of P_0 u'_1 u'_2 ∧ x = y ⇒ P_0 u_1 u_2⟩⟩
    | (?P : Prop) ↦ ⟨P, ⟨λp : P.p⟩⟩
rewriteTerm : Πϕ ≪ •.Πx : [ϕ]Nat.Πy : [ϕ]Nat.Πu : [ϕ]Nat.Σu' : [ϕ]Nat.prf([ϕ]∀P : Prop.P u' ∧ x = y ⇒ P u) =
  νϕ.Λx : Nat.Λy : Nat.ΛP : Prop.ltermcase P with
      x ↦ ⟨y, ⟨proof of ∀P : Prop.P y ∧ x = y ⇒ P x⟩⟩
    | (?f : Nat → Nat → Nat) (?u_1 : Nat) (?u_2 : Nat) ↦  open rewriteTerm [curctx] x y u_1 as ⟨u'_1, ⟨X_1 :: ∀P : P u'_1 ∧ x = y ⇒ P u_1⟩⟩ in
                                                          open rewriteTerm [curctx] x y u_2 as ⟨u'_2, ⟨X_2 :: ∀P : P u'_2 ∧ x = y ⇒ P u_2⟩⟩ in
                                                          ⟨f u'_1 u'_2, ⟨proof of ∀P : Prop.P (f u'_1 u'_2) ∧ x = y ⇒ P (f u_1 u_2)⟩⟩
    | (?u : Nat) ↦ ⟨n, ⟨proof of ∀P : Prop.P u ∧ x = y ⇒ P u⟩⟩
```

Figure 13. Programming examples

dependence on the extra variable is not recorded locally on the type. We have yet to find a simpler way to deal with this shortcoming of Delphin's type system than the approach we propose here. Last, we believe that our approach helps elucidate the connection of Delphin with Beluga, and that the translation between the two levels we've defined can serve as the basis of a similar translation of Delphin terms into Beluga terms.

FreshML [Shinwell et al. 2003] is a language that extends the ML datatype mechanism so that datatypes with binding can naturally be encoded; the issue of free variables escaping their expected scope also shows up in that setting too. The original type system for the language cannot effectively guard against this issue. This situation was rectified with Pure FreshML [Pottier 2007], which introduces a program logic that permits the user to establish that free variables are correctly managed. Compared to FreshML, we have chosen to have fresh variable introduction have a lexically scoped effect instead of a global one, and thus we can arrive at a simpler solution based purely on a type system.

In recent years many languages with rich dependent type systems have been proposed, which bear similarity to the language design we propose here; unfortunately they are too numerous to cover here but we refer the reader to two of the most recent proposals [Norell 2007, Fogarty et al. 2007] and the references therein. Of these, 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 used only as index types, and actually have a runtime representation that is essential for the kind of code we want to write. Furthermore, the inclusion of such a construct requires that we also consider the issue of type-safe handling of open terms, which requires a non-trivial extension of the type system as we have shown. To the best of our knowledge, this issue does not arise in most of the work on languages with dependent type systems, other than the languages we have previously mentioned.

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 contained in our logic, 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 in our logic – something that depends of course in 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 will lead to a more principled and modular programming paradigm, a claim that we aim to substantiate with future work.

Last, 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 in powerful ways, employing pattern matching on propositions as well as on full proof contexts (sets of hypotheses coupled with a goal), general recursion and also backtracking. Our language, being in its initial stages, is far from supporting most of these features; pattern matching on propositions is limited to linear unification (even though non-linear unification can be achieved with nested matches), the support for

proof states is lacking at best, and we do not yet have an implementation for a backtracking search strategy in the style of LTac. Still, we believe that our language can serve as a kernel where such features can be developed in order to recover the practicality that current use of LTac demonstrates. Also, our language is strongly typed, and gives access to a richer set of programming constructs, including effectful ones; this, we believe, will enable development of more robust and complex tactics. 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.

8. Summary and future work

In this paper we have shown the beginnings of a language design that introduces first-class support for a logical framework modeled after 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 logical terms and proof objects, even when these terms are open. This is done by having a two-layer approach. At the surface level an ambient free variables context is used, which requires straightforward use of constructs that manipulate it. At the intermediate level, all terms are closed with respect to their free variables, which allows for normal type safety proofs. A typed translation exists from the surface to the intermediate level. The approach we use is designed in a way so that it is orthogonal to the rest of the constructs in the language, while at the same time it does not require changes to the typing judgements of our logical framework. Last, we have shown how a number of tactics can be developed inside the proposed language.

As part of our future work, we would like to extend the handling of proof objects in the computational language so that free proof object variables are also allowed, instead of just logical term variables. This would enable an enhanced treatment of logical implication compared to what is currently possible in our language. Also, we would like to extend our logical framework to handle full CIC terms. We trust that the approach to free variables that we've presented generalizes to incorporate both these additions.

References

H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. 2001.

Y. Bertot, P. Casteran, G. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, 2004.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective Interactive Proofs for Higher-Order Imperative Programs. In *Proceedings of International Conference on Functional Programming (to appear)*, 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.

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. Practical foundations for programming languages. *Working draft*, 2007.

D.R. Licata and R. Harper. Positively dependent types. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 3–14. ACM New York, NY, USA, 2009a.

D.R. Licata and R. Harper. A Universe of Binding and Computation. In *Proceedings of International Conference on Functional Programming (to appear)*, 2009b.

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.

A. Nanevski, P. Govereau, and G. Morrisett. Type-theoretic semantics for transactional concurrency. Technical report, Technical Report TR-09-07, Harvard University, 2007.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of International Conference on Functional Programming*, volume 8, 2008.

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

U. Norell. *Towards a practical programming language based on dependent type theory*. 2007.

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

R.L. Petersen, L. Birkedal, A. Nanevski, and G. Morrisett. A realizability model for impredicative Hoare Type Theory. *Lecture Notes in Computer Science*, 4960:337, 2008.

F. Pfenning. Logic programming in the LF logical framework. *Logical Frameworks*, pages 149–181, 1991.

F. Pfenning and C. Schürmann. System description: Twelf-a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, pages 202–206, 1999.

B. Pientka and J. 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.

F. Pottier. Static name control for FreshML. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 356–365, 2007.

M.R. Shinwell, A.M. Pitts, and M.J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 263–274. ACM New York, NY, USA, 2003.

A. Stampoulis and Z. Shao. Typed computation of logical terms inside a language with effects (extended version), 2009. URL `http://zoo.cs.yale.edu/~ams257/popl2010/TR.pdf`.

B. Werner. Une Théorie des Constructions Inductives. 1994.

R. Wisnesky, G. Malecha, and G. Morrisett. Certified Web Services in Ynot. 2009.

A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.