A Computational Approach to Reflective Meta-Reasoning about Languages with Bindings *

Aleksey Nogin Alexei Kopylov Xin Yu Jason Hickey

Department of Computer Science
California Institute of Technology
M/C 256-80, Pasadena, CA 91125
{nogin,kopylov,xiny,jyh}@cs.caltech.edu

Abstract

We present a foundation for a computational meta-theory of languages with bindings implemented in a computer-aided formal reasoning environment. Our theory provides the ability to reason abstractly about operators, languages, open-ended languages, classes of languages, *etc.* The theory is based on the ideas of higher-order abstract syntax, with an appropriate induction principle parameterized over the language (*i.e.* a set of operators) being used. In our approach, both the bound and free variables are treated uniformly and this uniform treatment extends naturally to variable-length bindings. The implementation is reflective, namely there is a natural mapping between the meta-language of the theorem-prover and the object language of our theory. The object language substitution operation is mapped to the meta-language substitution and does not need to be defined recursively. Our approach does not require designing a custom type theory; in this paper we describe the implementation of this foundational theory within a general-purpose type theory. This work is fully implemented in the MetaPRL theorem prover, using the pre-existing NuPRL-like Martin-Löf-style computational type theory. Based on this implementation, we lay out an outline for a framework for programming language experimentation and exploration as well as a general reflective reasoning framework. This paper also includes a short survey of the existing approaches to syntactic reflection.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; F.4.3 [*Mathematical Logic and Formal Languages*]: Formal Languages—Operations on languages

General Terms Languages, Theory, Verification

Keywords Higher-Order Abstract Syntax, Reflection, Type Theory, MetaPRL, NuPRL, Programming Language Experimentation, Languages with Bindings.

* An extended version of this paper is available as Caltech Technical Report CaltechCSTR:2005.003 [NKYH05]

MERLIN'05 September 30, 2005, Tallinn, Estonia.

1. Introduction

1.1 Reflection

Very generally, reflection is the ability of a system to be "self-aware" in some way. More specifically, by reflection we mean the property of a computational or formal system to be able to access and internalize some of its own properties.

There are many areas of computer science where reflection plays or should play a major role. When exploring properties of programming languages (and other languages) one often realizes that languages have at least two kinds of properties — *semantic* properties that have to do with the *meaning* of what the language's constructs express and *syntactic* properties of the language itself.

Suppose for example that we are exploring some language that contains arithmetic operations. And in particular, in this language one can write polynomials like x^2+2x+1. In this case the number of roots of a polynomial is a semantic property since it has to do with the *valuation* of the polynomial. On the other hand, the degree of a polynomial could be considered an example of a syntactic property since the most natural way to define it is as a property of the *expression* that *represents* that polynomial. Of course, syntactic properties often have semantic consequences, which is what makes them especially important. In this example, the number of roots of a polynomial is bounded by its degree.

Another area where reflection plays an important role is run-time code generation — in most cases, a language that supports run-time code generation is essentially reflective, as it is capable of manipulating its own syntax. In order to reason about run-time code generation and to express its semantics and properties, it is natural to use a reasoning system that is reflective as well.

There are many different flavors of reflection. The *syntactic reflection* we have seen in the examples above, which is the ability of a system to internalize its own syntax, is just one of these many flavors. Another very important kind of reflection is *logical reflection*, which is the ability of a reasoning system or logic to internalize and reason about its own logical properties. A good example of a logical reflection is reasoning about knowledge — since the result of reasoning about knowledge is knowledge itself, the logic of knowledge is naturally reflective [Art04].

In most cases it is natural for reflection to be iterated. In the case of syntactic reflection we might care not only about the syntax of our language, but also about the syntax used for expressing the syntax, the syntax for expressing the syntax for expressing the syntax and so forth. In the case of the logic of knowledge it is natural to have iterations of the form "I know that he knows that I know . . .".

When a formal system is used to reason about properties of programming languages, iterated reflection magnifies the power of the system, making it more natural to reason not just about individual languages, but also about *classes* of languages, language *schemas*, and so on. More generally, reflection adds a lot of additional power to a formal reasoning system [GS89, Art99]. In particular, it is well-known [Göd36, Mos52, EM71, Par71] that reflection allows a super-exponential reduction in the size of certain proofs. In addition, reflection could be a very useful mechanism for implementing proof search algorithms [ACU93, GWZ00, CFW04]. See also [Har95] for a survey of reflection in theorem proving.

1.2 Uniform Reflection Framework

For each of the examples in the previous section there are many *ad-hoc* ways of achieving the specific benefits of a specific flavor of reflection. This work aims at creating a *unifying reflective framework* that would allow achieving most of these benefits in a uniform manner, without having to reinvent and re-implement the basic reflective methodology every time. We believe that such a framework will increase the power of the formal reasoning tools, and it may also become an invaluable tool for exploring the properties of novel programming languages, for analyzing run-time code generation, and for formalizing logics of knowledge.

This paper establishes a foundation for the development of this framework — a new approach to reflective meta-reasoning about languages with bindings. We present a theory of syntax that:

- in a natural way provides both a higher-order abstract syntax (HOAS) approach to bindings and a de Bruijn-style approach to bindings, with easy and natural translation between the two;
- provides a uniform HOAS-style approach to both bound and free variables that extends naturally to variable-length "vectors" of binders;
- permits meta-reasoning about languages — in particular, the operators, languages, open-ended languages, classes of languages *etc*. are all first-class objects that can be reasoned about both abstractly and concretely;
- comes with a natural induction principle for syntax that can be parameterized by the language being used;
- provides a natural mapping between the object syntax and meta-syntax that is free of exotic terms, and allows mapping the object-level substitution operation directly to the meta-level one (*i.e.* β-reduction);
- is fully derived in a pre-existing type theory in a theorem prover;
- is designed to serve as a foundation for a general reflective reasoning framework in a theorem prover;
- is designed to serve as a foundation for a programming language experimentation framework.

The paper is structured as follows. Our work inherits a large number of ideas from previous efforts and we start in Section 2 with a brief survey of existing techniques for formal reasoning about syntax. Next in Section 3 we outline our approach to reasoning about syntax and in Section 4 we present a formal account of our theory based on a Martin-Löf style computational type theory [CAB+86, HAB+] and the implementation of that account in the MetaPRL theorem prover [Hic97, Hic99, Hic01, HNC+03, HNK+, HAB+]. Then in Section 5 we outline our plan for building a uniform reflection framework based on the syntactic reflection. Finally, in Section 6 we resume the discussion of related work that was started in Section 2.

1.3 Notation and Terminology

We believe that our approach to reasoning about syntax is fairly general and does not rely on any special features of the theorem prover we use. However, since we implement this theory in MetaPRL, we introduce some basic knowledge about MetaPRL terms.

A MetaPRL term consists of:

1. An operator name (like "sum"), which is a unique name indicating the logic and component of a term;
2. A list of parameters representing constant values; and
3. A set of subterms with possible variable bindings.

We use the following syntax to describe terms, based on the NuPRL definition [ACHA90]:

$$\underbrace{\mathit{opname}}_{\textit{operator name}}\ \underbrace{[p_1;\cdots;p_n]}_{\textit{parameters}}\underbrace{\{\vec{v_1}.t_1;\cdots;\vec{v_m}.t_m\}}_{\textit{subterms}}$$

In addition, MetaPRL has a meta-syntax somewhat similar to the higher-order abstract syntax presented in Pfenning and Elliott [PE88]. MetaPRL uses the second-order variables in the style of Huet and Lang [HL78] to describe term schemas. For example, $\lambda x.V[x]$, where V is a second-order variable of arity 1, is a schema that stands for an arbitrary term whose top-level operator is λ.

This meta-syntax requires that every time a binding occurrence is explicitly specified in a schema, all corresponding bound occurrences have to be specified as well. This requirement makes it very easy to specify free variable restrictions — for example, $\lambda x.V$, where V is a second-order meta-variable of arity 0, is a schema that stands for an arbitrary term whose top-level operator is λ *and* whose body does not have any free occurrences of the variable bound by that λ. In particular, the schema $\lambda x.V$ matches the term $\lambda y.1$, but not the term $\lambda x.x$.

In addition, this meta-language allows specifying certain term transformations, including implicit substitution specifications. For example, a beta reduction transformation may be specified using the following schema:

$$(\lambda x.V_1[x])\ V_2 \leftrightarrow V_1[V_2]$$

Here the substitution of V_2 for x in V_1 is specified implicitly.

Throughout this paper we will use this second-order notation to denote arbitrary terms — namely, unless stated otherwise, when we write "$\lambda x.t[x]$" we mean an arbitrary term of this form, not a term containing a concrete second-order variable named "t".

As in LF [HHP93] we assume that object level variables (*i.e.* the variables of the language whose syntax we are expressing) are directly mapped to meta-theory variables (*i.e.* the variable of the language that we use to express the syntax). Similarly, we assume that the object-level binding structure is mapped to the meta-level binding structure. In other words, the object-level notion of the "binding/bound occurrence" is a subset of that in the meta-language. We also consider α-equal terms — both on the object level and on the meta-level — to be identical and we assume that substitution avoids capture by renaming.

The sequent schema language we use [NH02] contains a number of more advanced features in addition to those outlined here. However, for the purposes of this presentation, the basic features outlined above are sufficient.

2. Previous Models of Reflection

In 1931 Gödel used reflection to prove his famous incompleteness theorem [Göd31]. To express arithmetic in arithmetic itself, he assigned a unique number (a *Gödel number*) to each arithmetic formula. A Gödel number of a formula is essentially a numeric code of a string of symbols used to represent that formula.

A modern version of the Gödel's approach was used by Aitken *et al.* [ACHA90, AC92, ACU93, Con94] to implement reflection in the NuPRL theorem prover [CAB+86, ACE+00]. A large part of this effort was essentially a reimplementation of the core of the NuPRL prover inside NuPRL's logical theory.

In Gödel's approach and its variations (including Aitken's one), a general mechanism that could be used for formalizing one logical theory in another is applied to formalizing a logical theory in itself. This can be very convenient for reasoning *about* reflection, but for our purposes it turns out to be extremely impractical. First, when formalizing a theory in itself using generic means, the identity between the theory being formalized and the one in which the formalization happens becomes very obfuscated, which makes it almost impossible to relate the reflected theory back to the original one. Second, when one has a theorem proving system that already implements the logical theory in question, creating a completely new implementation of this logical theory inside itself is a very tedious redundant effort. Another practical disadvantage of the Gödel numbers approach is that it tends to blow up the size of the formulas; and iterated reflection would cause the blow-up to be iterated as well, making it exponential or worse.

A much more practical approach is being used in some programming languages, such as Lisp and Scheme. There, the common solution is for the implementation to *expose* its internal syntax representation to user-level code by the `quote` constructor (where `quote` (t) prevents the evaluation of the expression t). The problems outlined above are solved instantly by this approach: there is no blow-up, there is no repetition of structure definitions, there is even no need for verifying that the reflected part is equivalent to the original implementation since they are *identical*. Most Scheme implementations take this even further: the `eval` function is the internal function for evaluating a Scheme expression, which is exposed to the user-level; Smith [Smi84] showed how this approach can achieve an infinite tower of processors. A similar language with the quotation and antiquotation operators was introduced in [GMO03].

This approach, however, violates the *congruence property* with respect to computation: if two terms are computationally equal then one can be substituted for the other in any context. For instance, although $2 * 2$ is equal to 4, the expressions "`2*2`" and "`4`" are syntactically different, thus we can not substitute `2*2` by `4` in the expression `quote(2*2)`. The congruence property is essential in many logical reasoning systems, including the NuPRL system mentioned above and the MetaPRL system [HNC+03, HNK+, HAB+] that our group uses.

A possible way to expose the internal syntax without violating the congruence property is to use the so-called "quoted" or "shifted" operators [AA99, Bar01, Bar05] rather than quoting the whole expression at once. For any operator *op* in the original language, we add the *quoted operator* (denoted as $\underline{op}$) to represent a term built with the operator *op*. For example, if the original language contains the constant "0" (which, presumably, represents the number 0), then in the reflected language, $\underline{0}$ would stand for the term that denotes the expression "0". Generally, the quoted operator has the same arity as the original operator, but it is defined on syntactic terms rather than on semantic objects. For instance, while $*$ is a binary operator on numbers, $\underline{*}$ is a binary operator on terms. Namely, if t_1 and t_2 are syntactic terms that stand for expressions e_1 and e_2 respectively, then $t_1 \underline{*} t_2$ is a new syntactic term that stands for the expression $e_1 * e_2$. Thus, the quotation of the expression $1*2$ would be $\underline{1} \underline{*} \underline{2}$.

In general, the well-formedness (typing) rule for a quoted operator is the following:

$$\frac{t_1 \in \text{Term} \qquad \ldots \qquad t_n \in \text{Term}}{\underline{op}\{t_1; \ldots; t_n\} \in \text{Term}} \tag{1}$$

where Term is a type of terms.

Note that quotations can be iterated arbitrarily many times, allowing us to quote quoted terms. For instance, $\underline{\underline{1}}$ stands for the term that denotes the term that denotes the numeral 1.

Problems arise when quoting expressions that contain binding variables. For example, what is the quotation of $\lambda x.x$? There are several possible ways of answering this question. A commonly used approach [PE88, DH94, DFH95, ACM02, ACM03] in logical frameworks such as Elf [Pfe89], LF [HHP93], and Isabelle [PN90, Pau94] is to construct an object logic with a concrete $\underline{\lambda}$ operator that has a type like

$$(\text{Term} \to \text{Term}) \to \text{Term} \quad \text{or} \quad (\text{Var} \to \text{Term}) \to \text{Term}.$$

In this approach, the quoted $\lambda x.x$ might look like $\underline{\lambda}(\lambda x.x)$ and the quoted $\lambda x.1$ might look like $\underline{\lambda}(\lambda x.\underline{1})$. Note that in these examples the quoted terms have to make use of both the syntactic (*i.e.* quoted) operator $\underline{\lambda}$ and the semantic operator λ.

Exotic Terms. Naïve implementations of the above approach suffer from the well-known problem of exotic terms [DH95, DFH95]. The issue is that in general we can not allow applying the $\underline{\lambda}$ operator to an arbitrary function that maps terms to terms (or variables to terms) and expect the result of such an application to be a "proper" reflected term.

Consider for example the following term:

$$\underline{\lambda}(\lambda x.\, \textbf{if } x = \underline{1} \textbf{ then } \underline{1} \textbf{ else } \underline{2})$$

It is relatively easy to see that it is not a real syntactic term and can not be obtained by quoting an actual term. (For comparison, consider $\underline{\lambda}(\lambda x.\, \underline{\textbf{if}}\ x\ \underline{=}\ \underline{1}\ \underline{\textbf{then}}\ \underline{1}\ \underline{\textbf{else}}\ \underline{2})$, which is a quotation of $\lambda x.\, \textbf{if } x = 1 \textbf{ then } 1 \textbf{ else } 2$).

How can one ensure that $\underline{\lambda}e$ denotes a "real" term and not an "exotic" one? That is, is it equal to a result of quoting an actual term of the object language? One possibility is to require e to be a *substitution function*; in other words it has to be equal to an expression of the form $\lambda x.t[x]$ where t is composed entirely of term constructors (*i.e.* quoted operators) and x, while using *destructors* (such as case analysis, the **if** operator used in the example above, *etc*) is prohibited.

There are a number of approaches to enforcing the above restriction. One of them is the usage of logical frameworks with restricted function spaces [PE88, HHP93], where λ-terms may only contain constructors. Another is to first formalize the larger type that does include exotic terms and then to define recursively a predicate describing the "validity" or "well-formedness" of a term [DH94, DFH95] thus removing the exotic terms from consideration. Yet another approach is to create a specialized type theory that combines the idea of restricted function spaces with a modal type operator [DPS97, DL99, DL01]. There the case analysis is disallowed on objects of "pure" type T, but is allowed on objects of a special type $\Box T$. This allows expressing both the restricted function space "$T_1 \to T_2$" and the unrestricted one "$(\Box T_1) \to T_2$" within a single type theory.

Another way of regarding the problem of exotic terms is that it is caused by the attempt to give a semantic definition to a primarily syntactic property. A more syntax-oriented approach was used by Barzilay *et al.* [BA02, BAC03, Bar05]. In Barzilay's approach, the quoted version of an operator that introduces a binding has the same *shape* (*i.e.* the number of subterms and the binding structure) as the original one and the variables (both the binding and the bound occurrences) are unaffected by the quotation. For instance, the quotation of $\lambda x.x$ is just $\underline{\lambda} x.x$.

The advantages of this approach include:

- This approach is simple and clear.
- Quoted terms have the same structure as original ones, inheriting a lot of properties of the object syntax.
- In all the above approaches, the α-equivalence relation for quoted terms is inherited "for free". For example, $\underline{\lambda} x.x$ and $\underline{\lambda} y.y$ are automatically considered to be the same term.
- Substitution is also easy: we do not need to re-implement the substitution that renames binding variables to avoid the capture of free variables; we can use the substitution of the original language instead.

To prune exotic terms, Barzilay says that $\underline{\lambda} x.t[x]$ is a valid term when $\lambda x.t[x]$ is a *substitution function*. He demonstrates that it is possible to formalize this notion in a *purely syntactical* fashion. In this setting, the general well-formedness rule for quoted terms with bindings is the following:

$$\frac{\mathit{is\_subst}_k\ \{x_1, \cdots, x_k.t[\vec{x}]\} \quad \cdots \quad \mathit{is\_subst}_l\ \{z_1, \cdots, z_l.s[\vec{z}]\}}{\underline{op}\{x_1, \cdots, x_k.t[\vec{x}]; \quad \cdots; \quad z_1, \cdots, z_l.s[\vec{z}]\} \in \text{Term}} \tag{2}$$

where $\mathit{is\_subst}_n\ \{x_1, \cdots, x_n.t[\vec{x}]\}$ is the proposition that t is a substitution function over variables $x_1, \cdots, x_n$ (in other words, it is a syntactic version of the `Valid` predicate of [DH94, DFH95]). This proposition is defined syntactically by the following two rules:

$$\frac{}{\mathit{is\_subst}_n\ \{x_1, \cdots, x_n.\, x_i\}}$$

and

$$\frac{\begin{array}{c}\mathit{is\_subst}_{n+k}\ \{x_1, \cdots, x_n, y_1, \cdots, y_k.t[\vec{x}; \vec{y}]\} \\ \vdots \\ \mathit{is\_subst}_{n+l}\ \{x_1, \cdots, x_n, z_1, \cdots, z_l.s[\vec{x}; \vec{z}]\}\}\end{array}}{\mathit{is\_subst}_n\ \{x_1 \cdots x_n.\underline{op}\{y_1 \cdots y_k.t[\vec{x}; \vec{y}]; \ \cdots; \ z_1 \cdots z_l.s[\vec{x}; \vec{z}]\}\}}$$

In this approach the $\mathit{is\_subst}_n\ \{\}$ and $\underline{\lambda}$ operators are essentially *untyped* (in NuPRL type theory, the computational properties of untyped terms are at the core of the semantics; types are added on top of the untyped computational system).

Recursive Definition and Structural Induction Principle. A difficulty shared by both the straightforward implementations of the (Term $\rightarrow$ Term) $\rightarrow$ Term approach and by the Barzilay's one is the problem of recursively defining the Term type. We want to define the Term type as the smallest set satisfying rules (1) and (2). Note, however, that unlike rule (1), rule (2) is not monotonic in the sense that $\mathit{is\_subst}_k\ \{x_1, \cdots, x_k.t[\vec{x}]\}$ depends non-monotonically on the Term type. For example, to say whether $\underline{\lambda} x.t[x]$ is a term, we should check whether t is a substitution function over x. It means at least that *for every* x in Term, $t[x]$ should be in Term as well. Thus we need to define the whole type Term before using (2), which produces a logical circle. Moreover, since $\underline{\lambda}$ has type (Term $\rightarrow$ Term) $\rightarrow$ Term, it is hard to formulate the structural induction principle for terms built with the $\underline{\lambda}$ term constructor.

Variable-Length Lists of Binders. In Barzilay's approach, for each number n, $\mathit{is\_subst}_n\ \{\}$ is considered to be a separate operator — there is no way to quantify over n, and there is no way to express variable-length lists of binders. This issue of expressing the unbounded-length lists of binders is common to some of the other approaches as well.

Meta-Reasoning. Another difficulty that is especially apparent in Barzilay's approach is that it only allows reasoning about *concrete* operators in concrete languages. This approach does not provide the ability to reason about operators *abstractly*; in particular, there is no way to state and prove meta-theorems that quantify over operators or languages, much less *classes* of languages.

3. Higher-Order Abstract Syntax with Inductive Definitions

Although it is possible to solve the problems outlined in the previous Section (and we will return to the discussion of some of those solutions in Section 6), our desire is to avoid these difficulties from the start. We propose a natural model of reflection that manages to work around those difficulties. We will show how to give a simple *recursive definition* of terms with binding variables, which *does not allow* the construction of exotic terms and *does allow* structural induction on terms.

In this Section we provide a conceptual overview of our approach; details are given in Section 4.

3.1 Bound Terms

One of the key ideas of our approach is how we deal with terms containing free variables. We extend to free variables the principle that *variable names do not really matter*. In fact, we model free variables as *bindings* that can be arbitrarily α-renamed. Namely, we will write $\mathit{bterm}\{x_1, \cdots, x_n.t[\vec{x}]\}$ for a term t over variables $x_1, \cdots, x_n$. For example, instead of term $x \underline{*} y$ we will use the term $\mathit{bterm}\{x, y.x \underline{*} y\}$ when it is considered over variables x and y and $\mathit{bterm}\{x, y, z.x \underline{*} y\}$ when it is considered over variables x, y and z. Free occurrences of x_i in $t[\vec{x}]$ are considered bound in $\mathit{bterm}\{x_1, \cdots, x_n.t[\vec{x}]\}$ and two α-equal $\mathit{bterm}\{\}$ expressions ("bterms") are considered to be *identical*.

Not every bterm is necessarily well-formed. We will define the type of terms in such a way as to eliminate exotic terms. Consider for example a definition of lambda-terms.

EXAMPLE 1. *We can define a set of reflected lambda-terms as the smallest set such that*

- $\mathit{bterm}\{x_1, \cdots, x_n.x_i\}$*, where* $1 \le i \le n$*, is a lambda-term (a variable);*
- *if* $\mathit{bterm}\left\{x_1, \cdots, x_n, x_{n+1}.t[\vec{x}]\right\}$ *is a lambda-term, then*

  $$\mathit{bterm}\left\{x_1, \cdots, x_n.\underline{\lambda} x_{n+1}.t[\vec{x}]\right\}$$

  is also a lambda-term (an abstraction);
- *if* $\mathit{bterm}\{x_1, \cdots, x_n.t_1[\vec{x}]\}$ *and* $\mathit{bterm}\{x_1, \cdots, x_n.t_2[\vec{x}]\}$ *are lambda-terms, then*

  $$\mathit{bterm}\{x_1; \cdots; x_n.\underline{apply}\{t_1[\vec{x}]; t_2[\vec{x}]\}\}$$

  is also a lambda-term (an application).

In a way, bterms could be understood as an explicit coding for Barzilay's substitution functions. And indeed, some of the basic definitions are quite similar. The notion of bterms is also very similar to that of *local variable contexts* [FPT99].

3.2 Terminology

Before we proceed further, we need to define some terminology.

DEFINITION 1. *We change the notion of* subterm *so that the subterms of a bterm are also bterms. For example, the immediate subterms of* $\mathit{bterm}\{x, y.x \underline{*} y\}$ *are* $\mathit{bterm}\{x, y.x\}$ *and* $\mathit{bterm}\{x, y.y\}$*; the immediate subterm of* $\mathit{bterm}\{x.\underline{\lambda} y.x\}$ *is* $\mathit{bterm}\{x, y.x\}$*.*

DEFINITION 2. *We call the number of outer binders in a bterm expression its* binding depth*. Namely, the* binding depth *of the bterm* $\mathit{bterm}\{x_1, \cdots, x_n.t[\vec{x}]\}$ *is* n*.*

DEFINITION 3. *Throughout the rest of the paper we use the notion of operator* shape*. The* shape *of an operator is a list of natural numbers each stating how many new binders the operator introduces on* *the corresponding subterm. The length of the shape list is therefore the arity of the operator. For example, the shape of the* $+$ *operator is* $[0; 0]$ *and the shape of the* λ *operator is* $[1]$.

The mapping from operators to shapes is also sometimes called a *binding signature* of a language [FPT99, Plo90].

DEFINITION 4. *Let op be an operator with shape* $[d_1; \cdots ; d_N]$, *and let btl be a list of bterms* $[b_1; \cdots ; b_M]$. *We say that btl is* compatible *with op at depth n when,*

1. $N = M$;
2. *the binding depth of bterm* b_j *is* $n + d_j$ *for each* $1 \le j \le N$.

3.3 Abstract Operators

Expressions of the form $\mathit{bterm}\{\vec{x}.\underline{op}\{\cdots\}\}$ can only be used to express syntax with *concrete* operators. In other words, each expression of this form contains a specific constant operator $\underline{op}$. However, we would like to reason about operators abstractly; in particular, we want to make it possible to have variables of the type "Op" that can be quantified over and used in the same manner as operator constants. In order to address this we use explicit term constructors in addition to $\mathit{bterm}\{\vec{x}.\underline{op}\{\cdots\}\}$ constants.

The expression $\mathit{mk\_bterm}\{n; \text{“}\underline{op}\text{”}; \mathit{btl}\}$, where "$\underline{op}$" is some encoding of the quoted operator $\underline{op}$, stands for a bterm with binding depth n, operator $\underline{op}$ and subterms *btl*. Namely,

$$\begin{aligned}\mathit{mk\_bterm}\{n;\ \underline{op};\ \ & \mathit{bterm}\{x_1, \cdots, x_n, \vec{y_1}.t_1[\vec{x}; \vec{y_1}]\} :: \cdots :: \\ & \mathit{bterm}\{x_1, \cdots, x_n, \vec{y_k}.t_k[\vec{x}; \vec{y_k}]\} :: \mathtt{nil}\}\end{aligned}$$

is $\mathit{bterm}\{x_1, \cdots, x_n.\underline{op}\,\{\vec{y_1}.t_1[\vec{x}; \vec{y_1}]; \cdots ; \vec{y_k}.t_k[\vec{x}; \vec{y_k}]\}\}$. Here, `nil` is the empty list and :: is the list `cons` operator and therefore the expression $b_1 :: \cdots :: b_n :: \mathtt{nil}$ represents the concrete list $[b_1; \cdots ; b_n]$.

Note that if we know the shape of the operator *op* and we know that the *mk_bterm* expression is well-formed (or, more specifically, if we know that *btl* is compatible with *op* at depth n), then it would normally be possible to deduce the value of n (since n is the difference between the binding depth of any element of the list *btl* and the corresponding element of the shape(*op*) list). There are two reasons, however, for supplying n explicitly:

- When *btl* is empty (in other words, when the arity of *op* is 0), the value of n can not be deduced this way and still needs to be supplied somehow. One could consider 0-arity operators to be a special case, but this results in a significant loss of uniformity.
- When we do *not* know whether an *mk_bterm* expression is necessarily well-formed (and as we will see it is often useful to allow this to happen), then a lot of definitions and proofs are greatly simplified when the binding depth of *mk_bterm* expressions is explicitly specified.

Using the *mk_bterm* constructor and a few other similar constructors that will be introduced later, it becomes easy to reason abstractly about operators. Indeed, the second argument to *mk_bterm* can now be an arbitrary expression, not just a constant. This has a cost of making certain definitions slightly more complicated. For example, the notion of "compatible with *op* at depth n" now becomes an important part of the theory and will need to be explicitly formalized. However, this is a small price to pay for the ability to reason abstractly about operators, which easily extends to reasoning abstractly about languages, classes of languages and so forth.

3.4 Inductively Defining the Type of Well-Formed Bterms

There are two equivalent approaches to inductively defining the general type (set) of all well-formed bterms. The first one follows the same idea as in Example 1:

- $\mathit{bterm}\{x_1, \cdots, x_n.x_i\}$ is a well-formed bterm for $1 \le i \le n$;
- $\mathit{mk\_bterm}\{n; \mathit{op}; \mathit{btl}\}$ is a well-formed bterm when *op* is a well-formed quoted operator and *btl* is a list of well-formed bterms that is compatible with *op* at some depth n.

If we denote $\mathit{bterm}\{x_1, \cdots, x_l, y, z_1, \cdots, z_r.y\}$ as $\mathit{var}\{l; r\}$, we can restate the base case of the above definition as "$\mathit{var}\{l; r\}$, where l and r are arbitrary natural numbers, is a well-formed bterm". Once we do this it becomes apparent that the above definition has a lot of similarities with de Bruijn-style indexing of variables [dB72]. Indeed, one might call the numbers l and r the *left and right indices of the variable* $\mathit{var}\{l; r\}$.

It is possible to provide an alternate definition that is closer to pure HOAS:

- $\mathit{bnd}\{x.t[x]\}$, where t is a well-formed substitution function, is a well-formed bterm (the *bnd* operation increases the binding depth of t by one by adding x to the beginning of the list of t's outer binders).
- $\mathit{mk\_term}\{\mathit{op}; \mathit{btl}\}$, where *op* is a well-formed quoted operator, and *btl* is a list of well-formed bterms that is compatible with *op* at depth 0, is a well-formed bterm (of binding depth 0).

Other than better capturing the idea of HOAS, the latter definition also makes it easier to express the reflective correspondence between the meta-syntax (the syntax used to express the theory of syntax, namely the one that includes the operators *mk_bterm*, *bnd*, *etc.*) and the meta-meta-syntax (the syntax that is used to express the theory of syntax and the underlying theory, in other words, the syntax that includes the second-order notations.) Namely, provided that we define the $\mathit{subst}\{\mathit{bt}; t\}$ operation to compute the result of substituting a closed term t for the first outer binder of the bterm *bt*, we can state that

$$\mathit{subst}\{\mathit{bnd}\{x.t_1[x]\}\,; t_2\} \equiv t_1[t_2] \tag{3}$$

(where t_1 and t_2 are literal second-order variables). In other words, we can state that the substitution operator *subst* and the implicit second-order substitution in the "meta-meta-" language are equivalent.

The downside of the alternate definition is that it requires defining the notion of "being a substitution function".

3.5 Our Approach

In our work we try to combine the advantages of both approaches outlined above. In the next Section we present a theory that includes both the HOAS-style operations (*bnd*, *mk_term*) and the de Bruijn-style ones (*var*, *mk_bterm*). Our theory also allows deriving the equivalence (3). In our theory the definition of the basic syntactic operations is based on the HOAS-style operators; however, the recursive definition of the type of well-formed syntax is based on the de Bruijn-style operations. Our theory includes also support for variable-length lists of binders.

4. Formal Implementation in a Theorem Prover

In this Section we describe how the foundations of our theory are formally defined and derived in the NuPRL-style Computational Type Theory in the MetaPRL Theorem Prover. For brevity, we will present a slightly simplified version of our implementation; full details are available in the extended version of this paper [NKYH05, Appendix].

4.1 Computations and Types

In our work we make heavy usage of the fact that our type theory allows us to define computations *without* stating upfront (or even knowing) what the relevant types are. In NuPRL-style type theories (which some even dubbed "untyped type theory"), one may define arbitrary recursive functions (even potentially nonterminating ones). Only when proving that such function belongs to a particular type, one may have to prove termination. See [All87a, All87b] for a semantics that justifies this approach.

The formal definition of the syntax of terms consists of two parts:

- The definition of untyped term constructors and term operations, which includes both HOAS-style operations and de Bruijn-style operations. As it turns out, we can establish most of the reduction properties without explicitly giving types to all the operations.
- The definition of the type of terms. We will define the type of terms as the type that contains all terms that can be legitimately constructed by the term constructors.

4.2 HOAS Constructors

At the core of our term syntax definition are two basic HOAS-style constructors:

- $\mathit{bnd}\{x.t[x]\}$ is meant to represent a term with a free variable x. The intended semantics (which will not become explicit until later) is that $\mathit{bnd}\{x.t[x]\}$ will only be considered well-formed when t is a substitution function.

  Internally, $\mathit{bnd}\{x.t[x]\}$ is implemented simply as the pair $\langle 0, \lambda x.t[x]\rangle$. This definition is truly internal and is used only to prove the properties of the two destructors presented below; it is never used outside of this Section (Section 4.2).
- $\mathit{mk\_term}\{op; ts\}$ pairs op with ts. The intended usage of this operation (which, again, will only become explicit later) is that it represents a closed term (*i.e.* a *bterm* of binding depth 0) with operator op and subterms ts. It will be considered well-formed when op is an operator and ts is a list of terms that is *compatible* with op at depth 0. For example, $\mathit{mk\_term}\{\underline{\lambda}; \mathit{bnd}\{x.x\}\}$ is $\underline{\lambda x.x}$.

  Internally, $\mathit{mk\_term}\{op; ts\}$ is implemented as the nested pair $\langle 1, \langle op, ts\rangle\rangle$. Again, this definition is never used outside of this Section.

We also implement two destructors:

- $\mathit{subst}\{bt; t\}$ is meant to represent the result of substituting term t for the first variable of the bterm bt. Internally, $\mathit{subst}\{bt; t\}$ is defined simply as an application $(bt.2)\ t$ (where $bt.2$ is the second element of the pair bt).

  We derive the following property of this substitution operation:

  $$\mathit{subst}\{\mathit{bnd}\{x.t_1[x]\}; t_2\} \equiv t_1[t_2]$$

  where "$\equiv$" is the computational equality relation[1] and t_1 and t_2 may be absolutely arbitrary, even ill-typed. This derivation is the only place where the internal definition of $\mathit{subst}\{bt; t\}$ is used.

  Note that the above equality is exactly the "reflective property of substitution" (3) that was one of the design goals for our theory.
- $\mathit{weak\_dest}\{bt; bcase; op, ts.\mathit{mkt\_case}[op; ts]\}$ is designed to provide a way to find out whether bt is a $\mathit{bnd}\{\}$ or a $\mathit{mk\_term}\{op; ts\}$ and to "extract" the op and ts in the latter case. In the rest of this paper we will use the "pretty-printed" form for *weak_dest* — "$\mathtt{match}\ bt\ \mathtt{with}\ \mathit{bnd}\{\_\} \to bcase \mid \mathit{mk\_term}\{op; ts\} \to \mathit{mkt\_case}[op; ts]$". Internally, it is defined as
  if $bt.1 = 0$ **then** $bcase$ **else** $\mathit{mkt\_case}[bt.2.1; bt.2.2]$.

  From this internal definition we derive the following properties of *weak_dest*:

  $$\begin{pmatrix} \mathtt{match}\ \mathit{bnd}\{x.t[x]\}\ \mathtt{with} \\ \quad \mathit{bnd}\{\_\} \to bcase \\ \mid \mathit{mk\_term}\{op; ts\} \to \mathit{mkt\_case}[op; ts] \end{pmatrix} \equiv bcase$$

  $$\begin{pmatrix} \mathtt{match}\ \mathit{mk\_term}\{op; ts\}\ \mathtt{with} \\ \quad \mathit{bnd}\{\_\} \to bcase \\ \mid \mathit{mk\_term}\{o; t\} \to \mathit{mkt\_case}[o; t] \end{pmatrix} \equiv \mathit{mkt\_case}[op; ts]$$

4.3 Vector HOAS Operations

As we have mentioned at the end of Section 2, some approaches to reasoning about syntax make it hard or even impossible to express arbitrary-length lists of binders. In our approach, we address this challenge by allowing operators where a single binding in the meta-language stands for a list of object-level bindings. In particular, we allow representing $\mathit{bnd}\{x_1.\mathit{bnd}\{x_2.\cdots \mathit{bnd}\{x_n.t[x_1; \ldots; x_n]\}\cdots\}\}$ as
$\mathit{vbnd}\{n; x.t[\mathit{nth}\{1; x\}; \ldots; \mathit{nth}\{n; x\}]\}$, where "$\mathit{nth}\{i; l\}$" is the "$i$-th element of the list l" function.

We define the following vector-style operations:

- $\mathit{vbnd}\{n; x.t[x]\}$ represents a "telescope" of nested *bnd* operations. It is defined by induction[2] on the natural number n as follows:

  $$\begin{array}{rcl} \mathit{vbnd}\{0; x.t[x]\} & := & t[\mathtt{nil}] \\ \mathit{vbnd}\{n+1; x.t[x]\} & := & \mathit{bnd}\{v.\mathit{vbnd}\{n; x.t[v :: x]\}\} \end{array}$$

  We also introduce $\mathit{vbnd}\{n; t\}$ as a simplified notation for $\mathit{vbnd}\{n; x.t\}$ when t does not have free occurrences of x.
- $\mathit{vsubst}\{bt; ts\}$ is a "vector" substitution operation that is meant to represent the result of simultaneous substitution of the terms in the ts list for the first $|ts|$ variables of the bterm bt (here $|l|$ is the length of the list l). $\mathit{vsubst}\{bt; ts\}$ is defined by induction on the list ts as follows:

  $$\begin{array}{rcl} \mathit{vsubst}\{bt; \mathtt{nil}\} & := & bt \\ \mathit{vsubst}\{bt; t :: ts\} & := & \mathit{vsubst}\{\mathit{subst}\{bt; t\}; ts\} \end{array}$$

Below are some of the derived properties of these operations:

$$\mathit{bnd}\{v.t[v]\} \equiv \mathit{vbnd}\{1; \mathtt{hd}(v)\} \tag{4}$$

$$\forall m, n \in \mathbb{N}. \big(\mathit{vbnd}\{m+n; x.t[x]\} \equiv \mathit{vbnd}\{m; y.\mathit{vbnd}\{n; z.t[y@z]\}\}\big) \tag{5}$$

$$\forall l \in \mathtt{List}.\ (\mathit{vsubst}\{\mathit{vbnd}\{|l|; v.t[v]\}; l\} \equiv t[l]) \tag{6}$$

$$\forall l \in \mathtt{List}.\forall n \in \mathbb{N}.\big((n \geq |l|) \Rightarrow (\mathit{vsubst}\{\mathit{vbnd}\{n; v.t[v]\}; l\} \equiv \mathit{vbnd}\{n - |l|; v.bt[l@v]\})\big) \tag{7}$$

$$\forall n \in \mathbb{N}. \big(\mathit{vbnd}\{n; l.\mathit{vsubst}\{\mathit{vbnd}\{n; v.t[v]\}; l\}\} \equiv \mathit{vbnd}\{n; l.t[l]\}\big) \tag{8}$$

where "$\mathtt{hd}$" is the list "head" operation, "@" is the list append operation, "$\mathtt{List}$" is the type of arbitrary lists (the elements of a list do not have to belong to any particular type), $\mathbb{N}$ is the type of natural numbers, and all the variables that are not explicitly constrained to a specific type stand for arbitrary expressions.

[1] In NuPRL-style type theories the computational equality relation (which is also sometimes called "squiggle equality" and is sometimes denoted as "$\sim$" or "$\longleftrightarrow$") is the finest-grained equality relation in the theory. When $a \equiv b$ is true, a may be replaced with b in an arbitrary context. Examples of computational equality include beta-reduction $\lambda x.a[x]\,b \equiv a[b]$, arithmetical equalities $(1 + 2 \equiv 3)$, and definitional equality (an abstraction is considered to be computationally equal to its definition).

[2] Our presentation of the inductive definitions is slightly simplified by omitting some minor technical details. See [NKYH05, Appendix] for complete details.

Equivalence (5) allows the merging and splitting of vector *bnd* operations. Equivalence (6) is a vector variant of equivalence (3). Equivalence (8) is very similar to equivalence (6) applied in the $vbnd\{n; l.\cdots\}$ context, except that (8) does not require l to be a member of any special type.

4.4 De Bruijn-style Operations

Based on the HOAS constructors defined in the previous two sections, we define two de Bruijn-style constructors.

- $var\{i; j\}$ is defined as $vbnd\{i; bnd\{v.vbnd\{j; v\}\}\}$. It is easy to see that this definition indeed corresponds to the informal

  $$bterm\{x_1, \cdots, x_l, y, z_1, \cdots, z_r.y\}$$

  definition given in Section 3.4.

- $mk\_bterm\{n; op; ts\}$ is meant to compute a bterm of binding depth n, with operator op, and with ts as its subterms. This operation is defined by induction on natural number n as follows:

  $$\begin{array}{rcl} mk\_bterm\{0; op; ts\} & := & mk\_term\{op; ts\} \\ mk\_bterm\{n+1; op; ts\} & := & \\ \multicolumn{3}{r}{bnd\{v.mk\_bterm\{n; op; map\ \lambda t.subst\{t; v\}\ ts\}\}} \end{array}$$

  Note that, if *ts* is a list of *bnd* expressions (which is the intended usage of the *mk_bterm* operation), then the

  $$bnd\{v.\ \cdots\ map\ \lambda t.subst\{t; v\}\ ts \cdots\}$$

  has the effect of stripping the outer *bnd* from each of the members of the *ts* list and "moving" them into a single "merged" *bnd* on the outside.

We also define a number of de Bruijn-style destructors, *i.e.*, operations that compute various de Bruijn-style characteristics of a bterm. Since the *var* and *mk_bterm* constructors are defined in terms of the HOAS constructors, the destructors have to be defined in terms of HOAS operations as well. Because of this, these definitions are often far from straightforward.

It is important to emphasize that the tricky definitions that we use here are only needed to establish the basic properties of the operations we defined. Once the basic theory is complete, we can raise the level of abstraction and no usage of this theory will ever require using any of these definitions, being aware of these definitions, or performing similar tricks again.

- $bdepth\{t\}$ computes the binding depth of term t. It is defined recursively using the Y combinator as

  $$Y\left(\begin{array}{l}\lambda f.\lambda b.\mathtt{match}\,b\ \mathtt{with} \\ \qquad bnd\{\_\} \to 1 + f\big(subst\{b; mk\_term\{0; 0\}\}\big) \\ \qquad \mid mk\_term\{\_; \_\} \to 0\end{array}\right) t$$

  In effect, this recursive function strips the outer binders from a bterm one by one using substitution (note that here we can use an arbitrary *mk_bterm* expression as a second argument for the substitution function; the arguments to *mk_bterm* do not have to have the "correct" type) and counts the number of times it needs to do this before the outermost *mk_bterm* is exposed.

  We derive the following properties of *bdepth*:

  $$\begin{array}{c}\forall l, r \in \mathbb{N}.\big(bdepth\{var\{l; r\}\} \equiv (l + r + 1)\big); \\ \forall n \in \mathbb{N}.\big(bdepth\{mk\_bterm\{n; op; ts\}\} \equiv n\big).\end{array}$$

  Note that the latter equivalence only requires n to have the "correct" type, while *op* and *ts* may be arbitrary. Since the *bdepth* operator is needed for defining the type of Term of well-formed bterms, at this point we would not have been able to express what the "correct" type for *ts* would be.

- $left\{t\}$ is designed to compute the "left index" of a *var* expression. It is defined as

  $$Y\left(\begin{array}{l}\lambda f.\lambda b.\lambda l. \\ \mathtt{match}\,b\ \mathtt{with} \\ \quad bnd\{\_\} \to \\ \quad\ 1 + f\big(subst\{b; mk\_term\{l; 0\}\}\big)(l+1) \\ \quad \mid mk\_term\big\{l'; \_\big\} \to l'\end{array}\right) t\ 0$$

  In effect, this recursive function substitutes $mk\_term\{0; 0\}$ for the first binding of t, $mk\_term\{1; 0\}$ for the second one, $mk\_term\{2; 0\}$ for the next one and so forth. Once all the binders are stripped and a $mk\_term\{l; 0\}$ is exposed, l is the index we were looking for. Note that here we intentionally supply *mk_term* with an argument of a "wrong" type ($\mathbb{N}$ instead of Op); we could have avoided this, but then the definition would have been significantly more complicated.

  As expected, we derive that

  $$\forall l, r \in \mathbb{N}.(left\{var\{l; r\}\} \equiv l).$$

- $right\{t\}$ computes the "right index" of a *var* expression. It is trivial to define in terms of the previous two operators: $right\{t\} := bdepth\{t\} - left\{t\} - 1$.

- $get\_op\{t; op\}$ is an operation such that

  $$\begin{array}{c}\forall n \in \mathbb{N}.\big(get\_op\big\{mk\_bterm\{n; op; ts\}; op'\big\} \equiv op\big), \\ \forall l, r \in \mathbb{N}.\big((get\_op\{var\{i; j\}; op\} \equiv op\big).\end{array}$$

  Its definition is similar to that of $left\{\}$.

- $subterms\{t\}$ is designed to recover the last argument of a *mk_bterm* expression. The definition is rather technical and complicated, so we omit it; see [NKYH05, Appendix C] for details. The main property of the *subterms* operation that we derive is

  $$\begin{array}{l}\forall n \in \mathbb{N}.\forall btl \in \mathtt{List}.\big(subterms\{mk\_bterm\{n; op; btl\}\} \equiv \\ \qquad\qquad map\ \lambda b.vbnd\{n; v.vsubst\{b; v\}\}\ btl\big)\end{array}$$

  The right-hand side of this equivalence is not quite the plain "*btl*" that one might have hoped to see here. However, when *btl* is a list of bterms with binding depths at least n, which is necessarily the case for any well-formed $mk\_bterm\{n; op; btl\}$, equivalence (8) would allow simplifying this right-hand side to the desired *btl*.

4.5 Operators

For this basic theory the exact representation details for operators are not essential and we define the type of operators Op abstractly. We only require that operators have decidable equality and that there exist a function of the type Op $\to$ $\mathbb{N}\,\mathtt{List}$ that computes operators' shapes.

Using this shape function and the *bdepth* function from Section 4.4, it is trivial to formalize the "*ts is compatible with op at depth n*" predicate of Definition 4. We denote this predicate as $shape\_compat\{n; op; ts\}$ and define it as

$$\begin{array}{l}|shape\{op\}| = |btl| \wedge \\ \quad \forall i \in 1..|btl|.bdepth\{nth\{btl; i\}\} = n + nth\{shape\{op\}; i\}\end{array}$$

4.6 The Type of Terms

In this section we will define the type of terms (*i.e.* well-formed bterms), Term, as the type of all terms that can be constructed by the de Bruijn constructors from Section 4.4. That is, the Term type contains all expressions of the forms:

- $var\{i; j\}$ for all natural numbers i, j; or

- $mk\_bterm\{n; op; ts\}$ for any natural number n, operator op, and list of terms ts that is compatible with op at depth n.

The Term type is defined as a fixpoint of the following function from types to types:

$$Iter(X) := \texttt{Image}(dom(X); x.mk(x)),$$

where

- $\texttt{Image}$ is a type constructor such that $\texttt{Image}(T; x.f[x])$ is the type of all the $f[t]$ for $t \in T$ (for it to be well-formed, T must be a well-formed type and f must not have any free variables except for x);
- $dom(X)$ is a type defined as

  $$(\mathbb{N}\times\mathbb{N})+\big(n{:}\mathbb{N}\times op{:}\mathrm{Op}\times\{ts{:}X\ \texttt{List} \mid shape\_compat\{n; op; ts\}\}\big);$$

- and $mk(x)$ (where x is presumably a member of the type $dom(X)$) is defined as

  $$\begin{array}{l} \texttt{match}\,x\ \texttt{with} \\ \quad \texttt{inl}\,(i, j) \to var\{i; j\} \\ \ \mid \texttt{inr}\,(n, op, ts) \to mk\_bterm\{n; op; ts\}\,. \end{array}$$

The fixpoint of $Iter$ is reached by defining

- $\mathrm{Term}_0 := Void$ (an empty type)
- $\mathrm{Term}_{n+1} := Iter(\mathrm{Term}_n)$
- $\mathrm{Term} := \bigcup\limits_{n\in\mathbb{N}} \mathrm{Term}_n$

We derive the intended introduction rules for the Term type:

$$\frac{i \in \mathbb{N} \qquad j \in \mathbb{N}}{var\{i; j\} \in \mathrm{Term}}$$

and

$$\frac{n \in \mathbb{N} \quad op \in \mathrm{Op} \quad ts \in \mathrm{Term}\,\texttt{List} \quad shape\_compat\{n; op; ts\}}{mk\_bterm\{n; op; ts\} \in \mathrm{Term}}\,.$$

Also, the structural induction principle is derived for the Term type. Namely, we show that to prove that some property $P[t]$ holds for any term t, it is sufficient to prove

- (Base case) P holds for all variables, that is, $P[var\{i; j\}]$ holds for all natural numbers i and j;
- (Induction step) $P[mk\_bterm\{n; op; ts\}]$ is true for any natural number n, any operator op, and any list of terms ts that is compatible with op at depth n, provided $P[t]$ is true for any element t of the list ts.

Note that the type of "terms over n variables" (where $n = 0$ corresponds to closed terms) may be trivially defined using the Term type and the "subset" type constructor — $\{t : \mathrm{Term} \mid bdepth\{t\} = n\}$.

5. Conclusions and Future Work

In Sections 3 and 4 we have presented a basic theory of syntax that is fully implemented in a theorem prover. As we mentioned in the introduction, the approach is both natural and expressive, and provides a foundation for reflective reasoning about classes of languages and logics. However, we consider this theory to be only the first step towards building a user-accessible uniform reflection framework and a user-accessible uniform framework for programming language reasoning and experimentation, where tasks similar to the ones presented in the POPLMARK challenge [ABF+05] can be performed easily and naturally. In this section we provide an outline of our plans for building such frameworks on top of the basic syntactic theory.

5.1 Higher-Level User Interface

One obvious shortcoming of the theory presented in Sections 3 and 4 is that it provides only the basic low-level operations such as *bnd*, *var*, *subterms*, *etc*. It presents a very low-level account of syntax in a way that would often fail to abstract away the details irrelevant to the user.

To address this problem we are planning to provide user interface functionality capable of mapping the high-level concepts to the low-level ones. In particular, we are going to provide an interface that would allow instantiating general theorems to specific collections of operators and specific languages. Thus, the user will be able to write something like "$\texttt{reflect language [}\lambda x.\cdot\texttt{;}\ apply\{\cdot; \cdot\}\texttt{]}$" and the system will create all the components outlined in Example 1:

- It will create a definition for the type

  $$\mathrm{Language}[\underline{\lambda} x.\cdot; \underline{apply}\{\cdot; \cdot\}]$$

  of reflected lambda-terms (where $\mathrm{Language}[l]$ is a general definition of a language over a list of operators l);
- It will state and derive the introduction rules for this type;
- It will state and derive the elimination rule for this type (the induction principle).

Moreover, we are planning to support even more complicated language declarations, such as

$$t \ := \ \texttt{int} \mid t \to t; \quad e \ := \ v \mid \lambda x : t.e[x] \mid apply\{e; e\}$$

that would cause the system to create mutually recursive type definitions and appropriate rules.

Finally, we are also planning to support "pattern bindings" that are needed for a natural encoding of ML-like pattern matching (such as the one sketched in the POPLMARK challenge [ABF+05]). As far as the underlying theory goes, we believe that the mechanisms very similar to the "vector bindings" presented in Section 4.3 will be sufficient here.

5.2 "Dereferencing" Quoted Terms

As in Barzilay's work, the quoted operator approach makes it easy to define the "unquoting" (or "dereferencing") operator $[\![]\!]_{\mathbf{unq}}$. If t is a syntactic term, then $[\![t]\!]_{\mathbf{unq}}$ is the value represented by t. By definition,

$$[\![\underline{op}\{t_1; \ldots; t_n\}]\!]_{\mathbf{unq}} = op\{[\![t_1]\!]_{\mathbf{unq}}; \ldots; [\![t_n]\!]_{\mathbf{unq}}\}.$$

For instance, $[\![\underline{2} \mathbin{\underline{*}} \underline{3}]\!]_{\mathbf{unq}}$ is $2 * 3$ (*i.e.* 6).

In order to define unquoting on terms with bindings, we need to introduce the "guard" operation $\langle\!|\ |\!\rangle$ such that $[\![\langle\!| t |\!\rangle]\!]_{\mathbf{unq}}$ is t for an arbitrary expression t. Then $[\![]\!]_{\mathbf{unq}}$ can be defined as follows:

$$\begin{array}{l} [\![\underline{op}\{x_1, \cdots, x_k.t[x_1; \ldots; x_k]; \cdots; z_1, \cdots, z_l.s[z_1; \ldots; z_l]\}]\!]_{\mathbf{unq}} = \\ \quad op\{x_1, \cdots, x_k.[\![t[\langle\!| x_1 |\!\rangle; \ldots; \langle\!| x_k |\!\rangle]]\!]_{\mathbf{unq}}; \\ \qquad \cdots; \\ \qquad z_1, \cdots, z_l.[\![s[\langle\!| z_1 |\!\rangle; \ldots; \langle\!| z_l |\!\rangle]]\!]_{\mathbf{unq}}\}. \end{array}$$

For example, $[\![\underline{\lambda} x.\underline{2} \mathbin{\underline{*}} x]\!]_{\mathbf{unq}} = \lambda x.[\![\underline{2} \mathbin{\underline{*}} \langle\!| x |\!\rangle]\!]_{\mathbf{unq}} = \lambda x.[\![\underline{2}]\!]_{\mathbf{unq}} * [\![\langle\!| x |\!\rangle]\!]_{\mathbf{unq}} = \lambda x.2 * x$.

The unquote operation establishes the identity between the original syntax and the reflected syntax, making it a "true" reflection.

Note that the type theory (which ensures, in particular, that only terminating functions may be shown to belong to a function type) would keep the $[\![\ ]\!]_{\mathbf{unq}}$ operation from introducing logical paradoxes.[3]

[3] This is, obviously, not a proper argument. While a proper argument can be made here, it is outside of the scope of this particular paper.

Also, since the notion of the quoted operators is fully open-ended, each new language added to the system will automatically get to use the $[\![\ ]\!]_{\mathbf{unq}}$ operation for all its newly introduced operators.

5.3 Logical Reflection

After defining syntactic reflection, it is easy to define *logical reflection*. If we consider the proof system open-ended, then the logical reflection is trivial — when P is a quotation of a proposition, we can regard "$[\![P]\!]_{\mathbf{unq}}$" as meaning "P is true". The normal modal rules for the $[\![]\!]_{\mathbf{unq}}$ modality are trivially derivable. For example *modus ponens*

$$[\![P \underline{\Rightarrow} Q]\!]_{\mathbf{unq}} \Rightarrow [\![P]\!]_{\mathbf{unq}} \Rightarrow [\![Q]\!]_{\mathbf{unq}}$$

is trivially true because if we evaluate the first $[\![]\!]_{\mathbf{unq}}$ (remember,

$$[\![P \underline{\Rightarrow} Q]\!]_{\mathbf{unq}} = \left([\![P]\!]_{\mathbf{unq}} \Rightarrow [\![Q]\!]_{\mathbf{unq}}\right)$$

by definition of $[\![]\!]_{\mathbf{unq}}$), we get an obvious tautology

$$([\![P]\!]_{\mathbf{unq}} \Rightarrow [\![Q]\!]_{\mathbf{unq}}) \Rightarrow [\![P]\!]_{\mathbf{unq}} \Rightarrow [\![Q]\!]_{\mathbf{unq}}.$$

In order to consider a closed proof system (in other words, if we want to be able to do induction over derivations), we would need to define a provability predicate for that system. We are planning to provide user interface functionality that would allow users to describe a set of proof rules and the system would generate appropriate proof predicate definitions and derive appropriate rules (in a style similar to the one outlined in Section 5.1 for the case of language descriptions).

6. Related Work

In Section 2 we have already discussed a number of approaches that we consider ourselves inheriting from. Here we would like to revisit some of them and mention a few other related efforts.

Our work has a lot in common with the HOAS implemented in Coq by Despeyroux and Hirschowitz [DH94]. In both cases, the more general space of terms (that include the exotic ones) is later restricted in a recursive manner. In both cases, the higher-order analogs of first-order de Bruijn operators are defined and used as a part of the "well-formedness" specification for the terms. Despeyroux and Hirschowitz use functions over infinite lists of variables to define open terms, which is similar to our vector bindings.

There are a number of significant differences as well. Our approach is sufficiently syntactical, which allows eliminating all exotic terms, even those that are extensionally equal to the well-formed ones, while the more semantic approach of [DH94, DFH95] has to accept such exotic terms (their solution to this problem is to consider an object term to be represented by the whole *equivalence class* of extensionally equal terms); more generally while [DH94] states that "this problem of extensionality is recurrent all over our work", most of our lemmas establish identity and not just equality, thus avoiding most of the issues of extensional equality. In our implementation, the substitution on object terms is mapped directly to β-reduction, while Despeyroux *et al*. [DFH95] have to define it recursively. In addition, we provide a *uniform* approach to both free and bound variables that naturally extends to variable-length "vector" bindings.

While our approach is quite different from the modal λ-calculus one [DPS97, DL99, DL01], there are some similarities in the intuition behind it. Despeyroux *et al*. [DPS97] says "Intuitively, we interpret $\Box B$ as the type of *closed* objects of type B. We can iterate or distinguish cases over closed objects, since all constructors are statically known and can be provided for." The intuition behind our approach is in part based on the canonical model of the NuPRL type theory [All87a, All87b], where *each* type is mapped to an equivalence relations over the closed terms of that type.

Gordon and Melham [GM96] define the type of λ-terms as a quotient of the type of terms with concrete binding variables over α-equivalence. Michael Norrish [Nor04] builds upon this work by replacing certain variable "freshness" requirements with variable "swapping". This approach has a number of attractive properties; however, we believe that the level of abstraction provided by the HOAS-style approaches makes the HOAS style more convenient and accessible.

Ambler, Crole, and Momigliano [ACM02] have combined the HOAS with the induction principle using an approach which in some sense is opposite to ours. Namely, they define the HOAS operators on top of the de Bruijn definition of terms using *higher order pattern matching*. In a later work [ACM03] they have described the notion of "*terms-in-infinite-context*" which is quite similar to our approach to vector binding. While our vector bindings presented in Section 4.3 are finite length, the exact same approach would work for the infinite-length "vectors" as well.

Acknowledgments

The authors are grateful to Eli Barzilay whose ideas were an inspiration for some of the work that lead to this paper. We are also grateful for his comments on an early draft of this paper.

We are grateful to the anonymous reviewers for their very thorough and fair feedback and many helpful suggestions.

References

[AA99] Eric Aaron and Stuart Allen. Justifying calculational logic by a conventional metalinguistic semantics. Technical Report TR99-1771, Cornell University, Ithaca, New York, September 1999.

[ABF+05] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. Available from `http://www.cis.upenn.edu/group/proj/plclub/mmm/`, 2005.

[AC92] William Aitken and Robert L. Constable. Reflecting on NuPRL : Lessons 1–4. Technical report, Cornell University, Computer Science Department, Ithaca, NY, 1992.

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The NuPRL open logical environment. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 95–197. IEEE Computer Society Press, June 1990.

[ACM02] Simon Ambler, Roy L. Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, pages 13–30, London, UK, 2002. Springer-Verlag.

[ACM03] S. J. Ambler, R. L. Crole, and Alberto Momigliano. A definitional approach to primitive recursion over higher order abstract syntax. In *Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pages 1–11. ACM Press, 2003.

[ACU93] William Aitken, Robert L. Constable, and Judith Underwood. Metalogical Frameworks II: Using reflected decision procedures. *Journal of Automated Reasoning*, 22(2):171–221, 1993.

[All87a] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, June 1987.

[All87b] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[Art99] Sergei Artemov. On explicit reflection in theorem proving and formal verification. In Ganzinger [Gan99], pages 267–281.

[Art04] Sergei Artemov. Evidence-based common knowledge. Technical Report TR-2004018, CUNY Ph.D. Program in Computer Science Technical Reports, November 2004.

[BA02] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in NuPRL. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32. National Aeronautics and Space Administration, 2002.

[BAC03] Eli Barzilay, Stuart Allen, and Robert Constable. Practical reflection in NuPRL. Short paper presented at 18th Annual IEEE Symposium on Logic in Computer Science, June 22–25, Ottawa, Canada, 2003.

[Bar01] Eli Barzilay. Quotation and reflection in NuPRL and Scheme. Technical Report TR2001-1832, Cornell University, Ithaca, New York, January 2001.

[Bar05] Eli Barzilay. *Implementing Reflection in NuPRL*. PhD thesis, Cornell University, 2005. In preparation.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.

[CFW04] Luís Crus-Filipe and Freek Weidijk. Hierarchical reflection. In Slind et al. [SBG04], pages 66–81.

[Con94] Robert L. Constable. Using reflection to explain and enhance type theory. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, NATO Series F*, pages 65–100. Springer, Berlin, 1994.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagaciones Mathematische*, 34:381–392, 1972. This also appeared in the Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam, series A, 75, No. 5.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculus and its Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, April 1995. Also appears as INRIA research report RR-2556.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In *LPAR '94: Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, 1994. Also appears as INRIA research report RR-2292.

[DH95] James Davis and Daniel Huttenlocher. Shared annotations for cooperative learning. In *Proceedings of the ACM Conference on Computer Supported Cooperative Learning*, September 1995.

[DL99] Joëlle Despeyroux and Pierre Leleu. A modal lambda calculus with iteration and case constructs. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs: International Workshop, TYPES '98, Kloster Irsee, Germany, March 1998*, volume 1657 of *Lecture Notes in Computer Science*, pages 47–61, 1999.

[DL01] Joëlle Despeyroux and Pierre Leleu. Recursion over objects of functional type. *Mathematical Structures in Computer Science*, 11(4):555–572, 2001.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher–order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163. Springer-Verlag, April 1997. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[EM71] Andrzej Ehrenfeucht and Jan Mycielski. Abbreviating proofs by adding new axioms. *Bulletin of the American Mathematical Society*, 77:366–367, 1971.

[F+86] Solomon Feferman et al., editors. *Kurt Gödel Collected Works*, volume 1. Oxford University Press, Oxford, Clarendon Press, New York, 1986.

[FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of 14^{th} IEEE Symposium on Logic in Computer Science*, pages 193+. IEEE Computer Society Press, 1999.

[Gan99] Harald Ganzinger, editor. *Proceedings of the 16^{th} International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, Berlin, July 7–10 1999. Trento, Italy.

[GM96] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, Turku, Finland, August 1996: Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.

[GMO03] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. Technical Report PRG-RR-03-16, Oxford Univerity, Computing Laboratory, 2003.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English version in [vH67].

[Göd36] K. Gödel. Über die Länge von beweisen. *Ergebnisse eines mathematischen Kolloquiums*, 7:23–24, 1936. English translation in [F+86], pages 397–399.

[GS89] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, Cambridge, Mass., 1989.

[GWZ00] H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13^{th} International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 162–178. Springer-Verlag, 2000.

[HAB+] Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.

[Har95] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-53, SRI International, Cambridge Computer Science Research Centre, Millers Yard, Cambridge, UK, February 1995.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. A revised and expanded verion of '87 paper.

[Hic97] Jason J. Hickey. NuPRL-Light: An implementation framework for higher-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 395–399. Springer, July 13–17 1997. An extended version of the paper can be found at `http://www.cs.caltech.edu/~jyh/papers/cade14_nl/default.html`.

[Hic99] Jason J. Hickey. Fault-tolerant distributed theorem proving. In Ganzinger [Gan99], pages 227–231.

[Hic01] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[HL78] Gérard P. Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HNC$^+$03] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[HNK$^+$] Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

[Mos52] Andrzej Mostowski. *Sentences undecidable in formalized arithmetic: an exposition of the theory of Kurt Gödel*. Amsterdam: North-Holland, 1952.

[NH02] Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.

[NKYH05] Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. Technical Report CaltechCSTR:2005.003, California Institure of Technology, 2005. Available at `http://resolver.caltech.edu/CaltechCSTR:2005.003`.

[Nor04] Michael Norrish. Recursive function definition for types with binders. In Slind et al. [SBG04], pages 241–256.

[Par71] R. Parikh. Existence and feasibility in arithmetic. *The Journal of Symbolic Logic*, 36:494–508, 1971.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[Pfe89] Frank Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 313–322, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Plo90] Gordon Plotkin. An illative theory of relations. In R. Cooper, K. Mukai, and J. Perry, editors, *Situation Theory and Its Applications, Volume 1*, number 22 in CSLI Lecture Notes, pages 133–146. Centre for the Study of Language and Information, 1990.

[PN90] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.

[SBG04] Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors. *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[Sch01] Carsten Schürmann. Recursion for higher-order encodings. In L. Fribourg, editor, *Computer Science Logic, Proceedings of the 10th Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 585–599. Springer-Verlag, 2001.

[Smi84] B.C. Smith. Reflection and semantics in Lisp. *Principles of Programming Languages*, pages 23–35, 1984.

[vH67] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.