Reflecting Higher-Order Abstract Syntax in Nuprl*

Eli Barzilay and Stuart Allen

Cornell University
{eli,sfa}@cs.cornell.edu

Abstract. This document describes part of an effort to achieve in Nuprl a practical reflection of its expression syntax. This reflection is done at the granularity of the operators; in particular, each operator of the syntax is *denoted* by another operator of the same syntax. Further, the syntax has binding operators, and we organize reflection not around the concrete binding syntax, but instead, around the abstract higher-order syntax. We formulate and prove the correctness of a core rule for inferring well-formedness of instances of operator-denoting operators.

1 Introduction

This work is part of an overall effort to get a practical reflection of syntax, computation and proof in Nuprl [4, 1, 3]. Reflecting syntax in a logical system entails writing proof rules that express that reflection, i.e, establishing an inferential connection between the actual syntax used and the meta-terms supposedly referring to it.

Operator-denoting operators are called *shifted* operators: if an operator x denotes operator y, then x is called a shifted y, and will be typeset as $\underline{y}$. For example, $a \underline{+} b$ denotes $c + d$ if a denotes c and b denotes d. The plus operator denotes a function that takes two integers and returns an integer, and its shifted version denotes a function that takes two terms and returns a term. The problem is what do we do with an operator that has a bound subterm: for example, $\forall x.\, P(x)$ is an operator that denotes a function taking a boolean or propositional function and returning a boolean or a proposition (its syntactic form is, of course, binding).

The obvious choice for the semantics of the shifted version would be a function, $\underline{\forall}(x, P)$ that takes two expressions as input values: one for the bound name, and one for the body, and constructs the concrete $\forall$ term. We will not pursue this direction. Instead, we shall adopt a higher-order abstract syntax [7]. Going in this direction, we get the usual benefits of this approach over concrete syntax (or alternatives like de-Bruijn indexes), such as specified in [8]. But we get a further bonus: it allows us to retain the same binding structure as the operator being denoted. In particular, the single input argument for $\underline{\forall}$ has the same binding as $\forall$: it takes in a term-valued function as an argument.

Implementing reflection in a programming language is usually done in a straightforward way: simply expose the implementation's evaluation function so it is available to programs written in the language. However, in a logical setting this is usually not the chosen approach, and the result is usually limited in its usability to theoretical or toy examples. The best example is Gödel numbers [5] which are good as a theoretical tool but not fit for an actual running system. Our goal is an eventual implementation that follows the same principle of exposing internal functionality: this is the outcome of operators being denoted by operators. The result is expected to be a system that has practical reflection implemented as is the situation in programming languages.

This construction is intended for the Nuprl system, but we avoid relying on a specific substitution function, which makes this approach applicable in the general case. Relevant information about Nuprl terms is limited to their content: a Nuprl term contains an operator id, and a list of bound subterms, each containing a list of bound variables and a term. Throughout this text we use a more conventional notation, with the extension of using underlines for shifted operators.

Returning to the question above: we begin by asking what is the semantics of $\underline{\forall}$? The semantics of a *concrete* shifted $\forall$ is the trivial one given above, but the semantics for $\underline{\forall}$ is more subtle.

* This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under Grant #N00014-01-1-0765

2 Semantics of Shifted Operators

Since $\underline{\forall}$ is a binding operator, it takes a function as an argument. Our basic requirement is that $F(t)$ be the result of the 'All-Instantiation' rule applied to $\underline{\forall}x.\, F(x)$ and t. This means that F needs to be a *substitution function*. So the semantics we adopt for $\underline{\forall}x.\, F(x)$ is that it denotes the $\forall$ formula whose predicate part is $F(u)$ and whose binder is u for some u — almost.

But which u? As usual we can avoid this question by using a higher-order abstract syntax, and say that what is denoted is actually the α-equivalence class of all such formulas where some appropriate u could be found. From this point forth, we use 'Term' to refer to these α-equivalence classes rather than the concrete terms.

Before going to the technical parts, lets consider how we might reason about this in the reflective logic. The first intuition is that proving that something is a Term depends only on having a quoted operator $\underline{\mathsf{opid}}$ and on its subparts in a simple compositional way:

$$\begin{array}{l}\vdash \underline{\mathsf{opid}}(\overline{v_1}.b_1;\overline{v_2}.b_2;\ldots;\overline{v_n}.b_n) \in \mathsf{Term}\\ \text{if}\ \ \overline{v_1} : \mathsf{Term} \vdash b_1 \in \mathsf{Term}\\ \quad \overline{v_2} : \mathsf{Term} \vdash b_2 \in \mathsf{Term}\\ \quad \vdots\\ \quad \overline{v_n} : \mathsf{Term} \vdash b_n \in \mathsf{Term}\end{array}$$

This seems fine, but it fails with bound variables. For example, the following can be proved:

$$\begin{array}{l}\vdash \underline{\lambda}(x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2}) \in \mathsf{Term}\\ \text{because}\ \ x : \mathsf{Term} \vdash \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2} \in \mathsf{Term}\end{array}$$

The premise line is trivial, but the original statement is false, because the quoted $\underline{\lambda}$-term contains a function which is not a substitution function — it is not a "template" function. In other words, there is no literally quoted term that this value stands for.

When inspecting this term, we can compare it to similar but valid terms to see what went wrong with this rule:

1. $\underline{\lambda}(x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2})$
2. $\underline{\lambda}(x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{1})$
3. $\lambda x.\ \mathsf{if}\ x = \underline{0}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ \underline{2}$

The two $\underline{\lambda}$-terms are fine, because they're built from substitution functions, and the last one is a simple $\mathsf{Term} \to \mathsf{Term}$ function. The difference between these terms and the previous one indicates what is wrong with the above rule: the bound variable should not be used as a value. It is a binding that should only be used in template holes, as there is no real value that this variable is ever bound to that can be used. In the valid examples, the first one did not *use* the bound value except for sticking it in its place. The second one almost used the value, but since the two branches are identical it is possible to avoid evaluating the test term; therefore it can be evaluated without using it, and the last one is not a Term but a function on Terms, so it can use that value as usual.

The conclusion is that a bound variable can be used only as an argument of a quoted term constructor. In other words, it can serve only as a value that is "computationally inert", much like universe expressions in [2]. This is also similar to variables that are bound by Scheme's syntax rules [6] — they are template variables that can be used in syntactic structures to build new structures[1]. When put in this light, it seems that any attempt to get this property in a proof fails. The lesson from this is: variables bound by quoted operators do not behave like normal bindings in the sense that they do not provide any values usable on the normal Nuprl level — and this is also true regarding universe expressions.

[1] For example, in the template `((foo x) (bar x))`, the identifier '`x`' is just a place holder that can be used to stick a value in a template; it is not possible to inspect its value.

3 Term Definition

We take CTerms as concrete terms: the type of objects intended to be ordinary syntax objects with binding operators. A more precise definition is given later, in Section 5. To define the Term type, we also need to introduce a predicate, `is_subst`, which is used to distinguish proper substitution functions. This predicate is defined in Section 6, and it has specific rules which are introduced in Section 6.1.

As said above, Terms are defined over these CTerms:

$$\mathsf{Term} \equiv \mathsf{CTerm}//\alpha$$

Terms are constructed by shifted operators, which have the semantics of functions that create Terms from Term substitution functions. For example:

$$\underline{\lambda} : \{f : \mathsf{Term} \to \mathsf{Term} \mid \mathtt{is\_subst}_1(f)\} \to \mathsf{Term}$$

using a version of `is_subst` that works with one argument functions. Generally, $\mathtt{is\_subst}_n$ is a predicate over $\mathsf{Term}^n \to \mathsf{Term}$. To simplify things, we drop the n when the context is clear.

CVar is a subsets of CTerm, which contains only atomic variable terms. Correspondingly, $\mathsf{Var} = \{\{x\} \mid x \in \mathsf{CVar}\}$, therefore $\mathsf{Var} \subseteq \mathsf{Term}$, since variables are α-equivalent only to themselves. Two assumptions that will be used in the following are that we have an infinite supply of distinct variables in CVar (and therefore in Var) and that there is at least one closed CTerm we can use.

4 Operations, Assumptions, and Facts

These are the operations that will be needed in the following text:

- $\cdot\uparrow$ taking the α-equivalence class of an object.
- $\cdot\downarrow$ choosing an element of an α-equivalence class. This is some function, such as one that chooses the first available variable names using lexicographic order.
- $\cdot[\cdot/\cdot]$ standard capture-avoiding substitution on CTerms. It can be used to substitute for multiple variables at one shot, provided that the number of supplied terms matches the number of variables, which are all distinct.
- $\cdot[\![\cdot/\cdot]\!]$ substitution for Terms, which is defined using the above operations as: $b[\![\overline{x}/\overline{v}]\!] = b\downarrow[\overline{x}\downarrow/\overline{v}\downarrow]\uparrow$.

$\mathrm{newcvar}(\cdot)$ returns a new CVar, i.e., $\mathrm{newcvar}(t)$ is neither free nor bound in $t \in \mathsf{CTerm}$.
$\mathrm{newvar}(\cdot)$ is similar to $\mathrm{newcvar}(\cdot)$ but for Terms, defined as: $\mathrm{newvar}(x) = \mathrm{newcvar}(x\downarrow)\uparrow$.
$\mathrm{newcvar}_n(\cdot)$ returns n new CVars, defined as:

$\mathrm{newcvar}_1(x) = \mathrm{newcvar}(x)$,
$\mathrm{newcvar}_{n+1}(x) = (\text{let } v = \mathrm{newcvar}_n(x) \text{ in } v, \mathrm{newcvar}(v, x))$.

$\mathrm{newvar}_n(\cdot)$ returns n new Vars, defined in the same way as $\mathrm{newcvar}_n(\cdot)$.

We use versions of these operations that are generalized to any lists and tuples of input arguments in an obvious way. The $\mathrm{newcvar}(\cdot)$ and $\mathrm{newvar}(\cdot)$ operations are further extended to functions by plugging in some closed dummy term argument (that we name '0') and using the result:

$$\forall f : \mathsf{Term}^n \to \mathsf{Term}.\ \mathrm{newvar}_m(f) = \mathrm{newvar}_m(f(0^n))$$

Below we will often justify things of the form $a\uparrow = b\uparrow$, by mentioning lemmas of the form $a =_\alpha b$, without emphasizing this transition.

Note: an overline indicates the value is a tuple and a way to index its elements. For example, $\overline{x} : \mathsf{Var}^n$ means that $\overline{x}$ is a list of n Vars, and that x_i is the ith element of this list; that is, x is a function from $i : 1 \ldots \mathrm{len}(\overline{x})$ to the ith element of $\overline{x}$.

4.1 Important Assumptions and Facts

In this section we state several assumptions and derived facts about substitution — the assumptions are not argued for, but we think that it is clear they are all true for any reasonable definition of substitution (one that respects the usual term binding structure). This allows us to take substitution as given and avoid getting into a specific implementation. These will be used in the following text.

$\star 1$ $\forall x : \mathsf{Term}.\ x = x\lfloor\uparrow$

$\star 2$ $\forall x : \mathsf{CTerm}.\ x =_\alpha x\uparrow\lfloor$
This fact is mostly used when nested in a bigger term, see $\star 4$ below.

$\star 3$ $\forall x : \mathsf{CVar}.\ x = x\uparrow\lfloor$
because $x \in \mathsf{CVar} \ \Rightarrow\ x\uparrow\lfloor = \{x\}\lfloor = x$.

$\star 4$ $\forall \overline{x_1}, \overline{x_2} : \mathsf{CTerm}^n, \overline{v} : \mathsf{CVar}^n,\ b : \mathsf{CTerm}.\ \overline{x_1} =_\alpha \overline{x_2} \ \Rightarrow\ b[\overline{x_1}/\overline{v}] =_\alpha b[\overline{x_2}/\overline{v}]$
Note that using this fact, $\star 2$ can be used in a subterm of an α-equality, since: $\forall t, x : \mathsf{CTerm}.\ t =_\alpha t[x\uparrow\lfloor/x]$

$\star 5$ $\forall b_1, b_2 : \mathsf{CTerm},\ \overline{t} : \mathsf{CTerm}^n,\ \overline{v} : \mathsf{CVar}^n.\ b_1 =_\alpha b_2 \ \Rightarrow\ b_1[\overline{t}/\overline{v}] =_\alpha b_2[\overline{t}/\overline{v}]$
note that $\overline{v}$ is the same on both sides (free variables in the body are not changed).

$\star 6$ $\forall t : \mathsf{CTerm},\ \overline{x_1} : \mathsf{CTerm}^{n_1},\ \overline{x_2} : \mathsf{CTerm}^{n_2},\ \overline{v_1}, \overline{u} : \mathsf{CVar}^{n_1},\ \overline{v_2} : \mathsf{CVar}^{n_2}$.
the sequence $\overline{v_1}, \overline{v_2}$ are distinct & $\overline{u}$ are distinct, not free in $t, \overline{x_2}$
$\Rightarrow\ t[\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}] =_\alpha t[\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}][\overline{x_1}/\overline{u}]$
This is simple to verify:

$$\begin{aligned} & t[\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}][\overline{x_1}/\overline{u}] \\ (\text{any of } \overline{u} \text{ do not occur free in } t) \quad & =_\alpha t[\overline{u}[\overline{x_1}/\overline{u}], \overline{x_2}[\overline{x_1}/\overline{u}]/\overline{v_1}, \overline{v_2}] \\ (\overline{u} \text{ are distinct}) \quad & =_\alpha t[\overline{x_1}, \overline{x_2}[\overline{x_1}/\overline{u}]/\overline{v_1}, \overline{v_2}] \\ (\text{any of } \overline{u} \text{ do not appear in } \overline{x_2}) \quad & =_\alpha t[\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}] \end{aligned}$$

Note that it is easy to show that such a $\overline{u}$ exists by choosing it as:
let $\overline{u} = \mathrm{newcvar}_{n_1}(t, \overline{x_2}, \ldots)$

$\star 7$ $\forall t : \mathsf{Term},\ \overline{x_1} : \mathsf{Term}^{n_1},\ \overline{x_2} : \mathsf{Term}^{n_2},\ \overline{v_1}, \overline{u} : \mathsf{Var}^{n_1},\ \overline{v_2} : \mathsf{Var}^{n_2}$.
the sequence $\overline{v_1}, \overline{v_2}$ are distinct & $\overline{u}$ are distinct, not free in $t, \overline{x_2}$
$\Rightarrow\ t[\![\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!] = t[\![\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!][\![\overline{x_1}/\overline{u}]\!]$
Again, verifying this is simple: from $\star 6$ we know that
$t\lfloor[\overline{u}\rfloor, \overline{x_2}\rfloor/\overline{v_1}\rfloor, \overline{v_2}\rfloor][\overline{x_1}\rfloor/\overline{u}\rfloor]\uparrow = t\lfloor[\overline{x_1}\rfloor, \overline{x_2}\rfloor/\overline{v_1}\rfloor, \overline{v_2}\rfloor]\uparrow$,
so:

$$\begin{aligned} t[\![\overline{u}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!][\![\overline{x_1}/\overline{u}]\!] &= t\lfloor[\overline{u}\rfloor, \overline{x_2}\rfloor/\overline{v_1}\rfloor, \overline{v_2}\rfloor]\uparrow\lfloor[\overline{x_1}\rfloor/\overline{u}\rfloor]\uparrow \\ (\star 2,\ \star 5) \quad &= t\lfloor[\overline{u}\rfloor, \overline{x_2}\rfloor/\overline{v_1}\rfloor, \overline{v_2}\rfloor][\overline{x_1}\rfloor/\overline{u}\rfloor]\uparrow \\ (\text{by the use of } \star 6 \text{ above}) \quad &= t\lfloor[\overline{x_1}\rfloor, \overline{x_2}\rfloor/\overline{v_1}\rfloor, \overline{v_2}\rfloor]\uparrow \\ &= t[\![\overline{x_1}, \overline{x_2}/\overline{v_1}, \overline{v_2}]\!] \end{aligned}$$

A similar note holds here: it is easy to show that such a $\overline{u}$ exists if it is chosen as:
let $\overline{u} = \mathrm{newcvar}_{n_1}(t\lfloor, \overline{x_2}\rfloor, \ldots\rfloor)\uparrow = \mathrm{newvar}_{n_1}(t, \overline{x_2}, \ldots)$

$\star 8$ $\forall c : \mathsf{CTerm},\ \overline{v}, \overline{u} : \mathsf{CVar}^n,\ \overline{s}, \overline{t} : \mathsf{CTerm}^n$.
$\overline{u}$ are not free in c except for $\overline{v}\ \Rightarrow\ c[\overline{s}/\overline{v}][\overline{t}/\overline{u}] =_\alpha c[\overline{s}[\overline{t}/\overline{u}]/\overline{v}]$
Note that the $\overline{v}$ exception is usually not needed.

$\star 9$ $\forall c : \mathsf{Term},\ \overline{v}, \overline{u} : \mathsf{Var}^n,\ \overline{s}, \overline{t} : \mathsf{Term}^n$.
$\overline{u}$ are not free in c except for $\overline{v}\ \Rightarrow\ c[\![\overline{s}/\overline{v}]\!][\![\overline{t}/\overline{u}]\!] = c[\![\overline{s}[\![\overline{t}/\overline{u}]\!]/\overline{v}]\!]$
This is easily shown by $\star 2$ and the definition of $\cdot[\![\cdot/\cdot]\!]$, using the previous fact.

A general intuition that arises from these facts and others, is that Term values are indeed isomorphic to CTerms: as long as there are no "dirty" concrete tricks played by *using* names of bound variables, facts that hold for CTerms will have corresponding versions for Terms.

5 Definitions of Shifted Operators

In the general case, a *shifted* operator id, $\underline{\mathsf{opid}}$, is defined as a function that takes in some substitution functions (determined by `is_subst`) of some arities, and returns a Term value. This is done in the obvious way: each of the substitution functions is used to plug in new variables; then the results, with the chosen variables, are all packaged into a CTerm; and finally, the α-equivalence class of this result produces the resulting Term. The actual representation is not too important — we could go with pairs and lists. For example:

$$\underline{\lambda}(f) = \langle \text{`}\lambda\text{'}, [\langle [\text{newvar}(f)\rfloor], f(\text{newvar}(f))\lfloor \rangle] \rangle \restriction$$

but this gets too complex in the general case (and it makes analysis hard, since we should know if a pair stands for a bound term, a term, or a pair of terms).

Instead, we use some types and abstract operations, which avoids committing us to some representation. The additional types we need are:

- OpId will be used for term name labels;
- $\mathsf{BndCTerm}_a$ is a *bound CTerm* (where $a : \mathbb{N}$) — packaging a CTerm with a distinct CVars.

BndCTerms are created with a `mkBndCTerm` constructor[2]:

$$\texttt{mkBndCTerm} \quad \in \quad a : \mathbb{N} \to (1 \ldots a \to \mathsf{CVar}) \to \mathsf{CTerm} \to \mathsf{BndCTerm}_a$$

An alternate syntax for `mkBndCTerm` can be more natural when a is known:

$$\texttt{mkBndCTerm}(\overline{x}, t) \quad \text{stands for} \quad \texttt{mkBndCTerm}(\text{len}(\overline{x}), (\lambda i.\ x_i), t)$$

CTerms are created with `mkCTerm`:

$$\begin{aligned} \texttt{mkCTerm} \quad \in \quad & \mathsf{OpId} \to n : \mathbb{N} \\ & \to a : (1 \ldots n \to \mathbb{N}) \\ & \to (i : 1 \ldots n \to \mathsf{BndCTerm}_{a_i}) \\ & \to \mathsf{CTerm} \end{aligned}$$

An alternate syntax for this which can be more natural when n, a are known is:

$$\texttt{mkCTerm}(o,\ [\texttt{mkBndCTerm}(\overline{x_1}, t_1), \ldots, \texttt{mkBndCTerm}(\overline{x_n}, t_n)])$$

which can be used instead of

$$\texttt{mkCTerm}(o,\ n,\ (\lambda i.\ \text{len}(\overline{x_i})),\ (\lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i)))$$

The next thing we need is a type which is the subset of $\mathsf{Term}^n \to \mathsf{Term}$ functions that are substitution functions (using the `is_subst` predicate):

$$\mathsf{SubstFunc}_n = \{f : \mathsf{Term}^n \to \mathsf{Term} \mid \texttt{is\_subst}_n(f)\}$$

Now we have reached the point where we can finally define a `mkTerm` constructor for Terms which uses `mkCTerm`:

$$\begin{aligned} \texttt{mkTerm} \quad \in \quad & \mathsf{OpId} \to n : \mathbb{N} \\ & \to a : (1 \ldots n \to \mathbb{N}) \\ & \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}) \\ & \to \mathsf{Term} \end{aligned}$$

[2] We use the notation $x : A \to B_x$ to denote functions on A such that $\forall x : A.\ f(x) \in B_x$, a type which is more conventionally denoted by $\Pi x : A.\ B_x$.

This function is defined as:

$$\texttt{mkTerm}(o, n, a, f) = \\ \quad \texttt{mkCTerm}(o, n, a, \lambda i.\ \text{let } \overline{x} = \text{newvar}_{a_i}(f_i) \text{ in } \texttt{mkBndCTerm}(\overline{x}\lfloor, f_i(\overline{x})\lfloor))\uparrow$$

and the alternate syntax for this is:

$$\texttt{mkTerm}(o, [\langle a_1, f_1\rangle, \ldots, \langle a_n, f_n\rangle])$$

which stands for

$$\texttt{mkTerm}(o, n, \lambda i.\ a_i, \lambda i.\ f_i) = \texttt{mkTerm}(o, n, a, f)$$

A shifted operator is a Term constructor which uses mkTerm with some fixed operator name and arity list. For example, '$\underline{\lambda}$' and '$\underline{\Sigma}$' are defined as:

$$\underline{\lambda}(f) = \texttt{mkTerm}(\text{`}\lambda\text{'}, [\langle 1, f\rangle]), \quad \underline{\Sigma}(f, g) = \texttt{mkTerm}(\text{`}\Sigma\text{'}, [\langle 0, f\rangle, \langle 1, g\rangle])$$

Note that since mkTerm is curried, a shifted operator is made by specifying the first three inputs: $\texttt{mkTerm}(o, n, a)$.

In addition to the assumptions and facts introduced in Section 4.1, we further assume the following:

⋆10 We specify one way that substitution interacts with CTerms — for all i, k, if it is true that

$$\text{if } v_k \text{ is free in } t_i \text{ then none of } \overline{x_i} \text{ are free in either } r_k \text{ or } v_k$$

then[3],

$$\texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i))[\overline{r}/\overline{v}] =_\alpha \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, t_i[\overline{r}/\overline{v}]))$$

To see why it is true using any reasonable definition of substitution, it is simpler to first see that a precondition that could be used is that none of $\overline{x_i}$ occur free in $\overline{r}, \overline{v}$; this is too restrictive for our future needs but the explanation is somewhat similar.

First of all, if v_k is not free in t_i, then there is no need for any restriction, since it does not have any effect on the result. Now, if it does appear in t_i, then it is enough to have two guarantees for the above to remain an α-equality: (a) if none of $\overline{x_i}$ are free in r_k then capture by $\overline{x_i}$ is impossible; (b) if v_k is not in $\overline{x_i}$, then none of the v_k will not get "screened out" in the body.

- A fact similar to this assumption also holds for Terms — if none of $\overline{x_i}$ occur free in $\overline{r}, \overline{v}, \overline{f_i(\underline{0}^{a_i})}$ then:

$$\texttt{mkTerm}(o, n, a, \lambda i.\ f_i)[\![\overline{r}/\overline{v}]\!] = \texttt{mkTerm}(o, n, a, \lambda i.\ \lambda\overline{z}.\ f_i(\overline{z})[\![\overline{r}/\overline{v}]\!])$$

However, it turns out that this fact is not needed, so no proof is given.

⋆11 A simple fact about renaming bound variables:

$$\forall \overline{x_i}, \overline{z_i} : \mathsf{CVar}^n.\ \overline{x_i} \text{ are distinct } \&\ \overline{z_i} \text{ are distinct } \&\ \overline{z_i} \text{ are not free in } b_i \\ \Rightarrow\ \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x_i}, b_i)) \\ =_\alpha \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}, b_i[\overline{z_i}/\overline{x_i}]))$$

6 Defining is_subst

A function is a *substitution function* iff there exists an appropriate substitution that it is equivalent to. First, we describe this using CTerms, since we know how substitutions work on them:

$$\texttt{is\_subst}_n(f) \equiv \exists b : \mathsf{CTerm}.\ \exists \overline{v} : \mathsf{CVar}^n.\ \forall \overline{t} : \mathsf{CTerm}^n.\ f(\overline{t}\lfloor) = b[\overline{t}/\overline{v}]\uparrow \tag{1}$$

Note that f returns a Term which is an α-equivalence class, so we have an equality rather than an α-equality. This should be equivalent to directly using a Term argument for f:

$$\texttt{is\_subst}_n(f) \equiv \exists b : \mathsf{CTerm}.\ \exists \overline{v} : \mathsf{CVar}^n.\ \forall \overline{r} : \mathsf{Term}^n.\ f(\overline{r}) = b[\overline{r}\lfloor/\overline{v}]\uparrow \tag{2}$$

We should show that $\forall b : \mathsf{CTerm},\ \forall \overline{v} : \mathsf{CTerm}^n$, the two sub-expressions are equivalent.

[3] Note that the α-equality is needed only because the substitution definition might introduce arbitrary renamings.

(1) ⇒ (2) Instantiate $\overline{t}$ with the chosen $\overline{r}\downarrow$:

$$f(\overline{r}) \overset{\star 1}{=} f(\overline{r}\downarrow\uparrow) \overset{(1)}{=} b[\overline{r}\downarrow/\overline{v}]\uparrow$$

(2) ⇒ (1) Instantiate $\overline{r}$ with $\overline{t}\uparrow$ and we get:

$$f(\overline{t}\uparrow) \overset{(2)}{=} b[\overline{t}\uparrow\downarrow/\overline{v}]\uparrow \overset{\star 2 \star 4}{=} b[\overline{t}/\overline{v}]\uparrow$$

We can now try to use a version that has all Term types and no CTerm types, using the α-terms substitution, $\cdot[\![\cdot/\cdot]\!]$:

$$\texttt{is\_subst}_n(f) \equiv \exists b_a : \mathsf{Term}.\ \exists \overline{v_a} : \mathsf{Var}^n.\ \forall \overline{t_a} : \mathsf{Term}^n.\ f(\overline{t_a}) = b_a[\![\overline{t_a}/\overline{v_a}]\!] \tag{3}$$

Now, verify that this is indeed equivalent to the other two definitions:

(2) ⇒ (3) Let $b_a = b\uparrow$, $\overline{v_a} = \overline{v}\uparrow$, pick some $\overline{t_a}$, and instantiate $\overline{r}$ with it:

$$f(\overline{t_a}) \overset{(2)}{=} b[\overline{t_a}\downarrow/\overline{v}]\uparrow \overset{\star}{=} b\uparrow\downarrow[\overline{t_a}\downarrow/\overline{v}\uparrow\downarrow]\uparrow = b\uparrow[\![\overline{t_a}/\overline{v}\uparrow]\!] = b_a[\![\overline{t_a}/\overline{v_a}]\!]$$

($\star$) is true because of $\star 2$ (with b), $\star 3$ (with $\overline{v}$), and $\star 5$ (with $x_1, x_2, \overline{t}, \overline{v}$).

(3) ⇒ (2) Let $b = b_a\downarrow$, $\overline{v} = \overline{v_a}\downarrow$, pick some $\overline{r}$, and instantiate $\overline{t_a}$ with it:

$$f(\overline{r}) \overset{(3)}{=} b_a[\![\overline{r}/\overline{v_a}]\!] = b_a\downarrow[\overline{r}\downarrow/\overline{v_a}\downarrow]\uparrow = b[\overline{r}\downarrow/\overline{v}]\uparrow$$

6.1 The `is_subst` Rules

Now that we have a reasonable definition of `is_subst`, we define key rules which use `is_subst` to prove that something is a proper Term. These rules turn out to be quite simple — there are only two cases:

- $H \vdash \texttt{is\_subst}(x_1, x_2, \ldots, x_n.\ x_i)$
- $H \vdash \texttt{is\_subst}(\overline{x}.\ \underline{\text{opid}}(\overline{y_1}.\ b_1; \ldots; \overline{y_n}.\ b_n))$ where $\underline{\text{opid}}$ is some quoted opid
  $H \vdash \texttt{is\_subst}(\overline{x}, \overline{y_1}.\ b_1)$
  $H \vdash \texttt{is\_subst}(\overline{x}, \overline{y_2}.\ b_2)$
  $\vdots$
  $H \vdash \texttt{is\_subst}(\overline{x}, \overline{y_n}.\ b_n)$

Note that this is enough for proving the validity of any Term value; for example, quoted constants succeeds immediately since their opid is quoted and they have no subterms. Proving $t \in \mathsf{Term}$ is achieved by showing $\texttt{is\_subst}(.\ t)$. (Of course, this is not a complete set of rules, since there are more cases where we have general Term expressions that are not constants.)

6.2 Justifying the `is_subst` Rules

The validity of the first rule amounts to this:

$$\forall n, i : \mathbb{N}^+.\ i \le n\ \Rightarrow\ \texttt{is\_subst}_n(\pi_n^i)$$

which is easily verified. Choose distinct $\overline{v} = v_1, \ldots, v_n$ variables, and let $b = \pi_n^i(\overline{v}) = v_i$. Then, $\forall \overline{t} : \mathsf{Term}^n.\ \pi_n^i(\overline{t}) = v_i[\![\overline{t}/\overline{v}]\!]$ is true by the definition of π_n^i, of $\cdot[\![\cdot/\cdot]\!]$, and the distinctness of $\overline{v}$.

Our main result will be formulating and proving the validity of the second rule, but this formulation requires some preparation. First, recall that the type of `mkTerm` is:

$$\mathsf{OpId} \to n : \mathbb{N} \to a : (1 \ldots n \to \mathbb{N}) \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}) \to \mathsf{Term}$$

Note that, as said earlier, a shifted operator is the result of applying `mkTerm` on the first three arguments, since they define the operator symbol and the list of arities it expects. For example:

$$\underline{\lambda} = \texttt{mkTerm}(\text{`}\lambda\text{'}, 1, \langle 1 \rangle) \qquad \underline{\Sigma} = \texttt{mkTerm}(\text{`}\Sigma\text{'}, 2, \langle 0, 1 \rangle)$$

So, a shifted operator has the following type, for some given o, n, and a:

$$\texttt{mkTerm}(o, n, a) : \quad (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}) \to \mathsf{Term}$$

Remember that the current goal is to conclude that for some shifted operator, $\underline{\mathsf{opid}}$:

$$\texttt{is\_subst}(\overline{x}.\ \underline{\mathsf{opid}}(\overline{v_1}.\ b_1, \ldots, \overline{v_n}.\ b_n))$$
$$\text{if}$$
$$\texttt{is\_subst}(\overline{x}, \overline{v_1}.\ b_1)\ \&\ \ldots\ \&\ \texttt{is\_subst}(\overline{x}, \overline{v_n}.\ b_n)$$

We need to compose the $\underline{\mathsf{opid}}$ function with an object that will make the result a $\mathsf{Term}^k \to \mathsf{Term}$ function (consuming the $x_1, \ldots, x_k$ variables) which we then show is a substitution function. This means the function that is composed with $\underline{\mathsf{opid}}$ should get a tuple of Term^k as input and return the vector of n substitution functions, built by consuming $\overline{x}$. In short, we package all the necessary information in F:

$$F : \mathsf{Term}^k \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i})$$

so we get the expected:

$$\texttt{mkTerm}(o, n, a) \circ F : \mathsf{Term}^k \to \mathsf{Term}$$

Now for the main result — the validity of the second rule may be formulated thus:

$$\begin{array}{l} \forall\ o : \mathsf{OpId},\ n : \mathbb{N},\ a : (1 \ldots n \to \mathbb{N}),\ k : \mathbb{N}, \\ \quad F : \mathsf{Term}^k \to (i : 1 \ldots n \to \mathsf{SubstFunc}_{a_i}). \\ \qquad \forall i : 1 \ldots n.\ \texttt{is\_subst}_{k+a_i}(\lambda_{(k)}\, ts, xs.\ F(ts)(i)(xs)) \\ \qquad\quad \Rightarrow\ \texttt{is\_subst}_k(\texttt{mkTerm}(o, n, a) \circ F) \end{array}$$

where $(\lambda_{(k)} x, y.\ B(x, y))(u_1, \ldots, u_{k+n}) \equiv B((u_1, \ldots, u_k), (u_{k+1}, \ldots, u_{k+n}))$.[4]

Proof. Assume o, n, a, k, and F are given as specified. We also assume that the constructed functions are substitution functions; therefore, for every $1 \le i \le n$ we get $c_i : \mathsf{Term},\ \overline{u_i} : \mathsf{Var}^k,\ \overline{v_i} : \mathsf{Var}^{a_i}$ such that:

$$\forall r_1^1, \ldots, r_k^1, r_1^2, \ldots, r_{a_i}^2 : \mathit{Term}.\ F(r_1^1, \ldots, r_k^1)(i)(r_1^2, \ldots, r_{a_i}^2) = c_i[\![\overline{r^1}, \overline{r^2}/\overline{u_i}, \overline{v_i}]\!]$$

- Let $\overline{t} : \mathsf{Term}^k$ be some k Terms,
- let $\overline{x_i} = \mathrm{newvar}_{a_i}(F(\overline{t})(i))$,
- and let $\overline{s} = \mathrm{newvar}_k(\overline{c}, \overline{x_1}, \ldots, \overline{x_n})$.

Now we can proceed: our goal due to the definition of `is_subst`, is to derive an equality of the form

$$(\texttt{mkTerm}(o, n, a) \circ F)(\overline{t}) = B[\![\overline{t}/\overline{X}]\!]$$

where, and this will be the tricky part, B and $\overline{X}$ are *independent* of the input, $\overline{t}$. So:

[4] Note that this special form of λ could be avoided if the fourth input type to `mkTerm` would take the terms first and then the index (instead of the $\mathsf{SubstFunc}_{a_i}$), but that would require a special composition operation instead.

$$
\begin{aligned}
(\mathtt{mkTerm}(o,n,a)\circ F)(\overline{t}) &= \\
&= \mathtt{mkTerm}(o,n,a,F(\overline{t})) \\
(\mathtt{mkTerm}\text{ def.}) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, F(\overline{t})(i)(\overline{x_i})\downarrow))\uparrow \\
(F\text{'s fact}) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{t}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow \\
(\star 7) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!][\![\overline{t}/\overline{s}]\!]\downarrow))\uparrow \\
(\cdot[\![\cdot/\cdot]\!]\text{ def.}) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow[\overline{t}\downarrow/\overline{s}\downarrow]\uparrow\downarrow))\uparrow \\
(\star 2) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow[\overline{t}\downarrow/\overline{s}\downarrow]))\uparrow \\
(\star 10,\text{ see below}) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))[\overline{t}\downarrow/\overline{s}\downarrow]\uparrow \\
(\star 2) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow\downarrow[\overline{t}\downarrow/\overline{s}\downarrow]\uparrow \\
(\cdot[\![\cdot/\cdot]\!]\text{ def.}) &= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]
\end{aligned}
$$

In the above, making sure $\star 10$ applies needs some care. Assume that for some j, l, the variable $s_j\downarrow$ is free in the lth body, which is $c_l[\![\overline{s}, \overline{x_l}/\overline{u_l}, \overline{v_l}]\!]\downarrow$. We need to make sure in this case that $\overline{x_l}\downarrow$ is not free in either $t_j\downarrow$ or $s_j\downarrow$. The latter is trivial by the choice of $\overline{s}$ (and holds for all indexes), but the former is not obvious. What we do know about $\overline{x_l}\downarrow$ is its definition:

$$\overline{x_l}\downarrow = \mathrm{newvar}_{a_l}(F(\overline{t})(l))\downarrow = \mathrm{newvar}_{a_l}(c_l[\![\overline{t}, 0^{a_l}/\overline{u_l}, \overline{v_l}]\!])\downarrow$$

but since $s_j\downarrow$ is free in $c_l[\![\overline{s}, \overline{x_l}/\overline{u_l}, \overline{v_l}]\!]\downarrow$, then $u_{l,j}\downarrow$ must appear in $c_l\downarrow$; therefore, the choice of $\overline{x_l}\downarrow$ above must pick variables that do not appear in $\overline{t_j}\downarrow$ so we're safe.

Going back to the main proof, the last term of the equality chain built so far was:

$$\mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

which has the $B[\![\overline{t}/\overline{X}]\!]$ structure that we're looking for, but we're not finished because both the B and the $\overline{X}$ parts depend on $\overline{t}$ — $\overline{x_i}$ is defined in terms of $\overline{t}$, $\overline{s}$ is defined in terms of $\overline{x_i}$, and both B and $\overline{X}$ parts contain instances of $\overline{s}$ (and B actually contains $\overline{x_i}$ as well).

So we choose $\overline{t}$-independent values now: let $\overline{x'_i} = \mathrm{newvar}_{a_i}(c_i)$ and let $\overline{s'} = \mathrm{newvar}_k(\overline{c}, \overline{x'_1}, \ldots, \overline{x'_k})$, we also need to show that in the above, using $\overline{x'_i}, \overline{s'}$ instead of $\overline{x_i}, \overline{s}$ is still the same value. In an attempt to simplify this we now choose n sets of variables $\overline{z_1} \in \mathsf{Term}^{a_1}, \ldots, \overline{z_n} \in \mathsf{Term}^{a_n}$, which are completely fresh: they do not appear in anything mentioned so far, including $\overline{t}$.

Now, back to our equality chain which left off at:

$$= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{x_i}\downarrow, c_i[\![\overline{s}, \overline{x_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

By $\star 11$:

$$= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

Next, we use substitution to get $\overline{s'}$ inside — $\overline{s}$ are distinct, $\overline{s'}$ are distinct, and $\overline{s'}$ does not occur in $\overline{z_i}$:

$$= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}[\![\overline{s}/\overline{s'}]\!], \overline{z_i}[\![\overline{s}/\overline{s'}]\!]/\overline{u_i}, \overline{v_i}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

Because $\overline{s'}$ does not occur free in c_i, this would be the expansion of the following substitution by $\star 9$:

$$= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!][\![\overline{s}/\overline{s'}]\!]\downarrow))\uparrow[\![\overline{t}/\overline{s}]\!]$$

Combining $\cdot[\![\cdot/\cdot]\!]$ and $\star 2$ we get:

$$= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow[\overline{s}\downarrow/\overline{s'}\downarrow]))\uparrow[\![\overline{t}/\overline{s}]\!]$$

$\overline{z_i}$ do not occur in either $\overline{s}$ or $\overline{s'}$ so we can use $\star 10$:

$$= \mathtt{mkCTerm}(o,n,a,\lambda i.\ \mathtt{mkBndCTerm}(\overline{z_i}\downarrow, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\downarrow))[\overline{s}\downarrow/\overline{s'}\downarrow]\uparrow[\![\overline{t}/\overline{s}]\!]$$

Again, using $\cdot[\![\cdot/\cdot]\!]$ and $\star 2$:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\rfloor, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\rfloor))\restriction[\![\overline{s}/\overline{s'}]\!][\![\overline{t}/\overline{s}]\!]$$

Now, $\overline{s}$ does not appear in the `mkCTerm` except possibly for $\overline{s'}$ (because we know it is not in c_i or $\overline{z_i}$), so using $\star 9$ we get:

$$\begin{aligned} &= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\rfloor, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\rfloor))\restriction[\![\overline{s}[\![\overline{t}/\overline{s}]\!]/\overline{s'}]\!] \\ &= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{z_i}\rfloor, c_i[\![\overline{s'}, \overline{z_i}/\overline{u_i}, \overline{v_i}]\!]\rfloor))\restriction[\![\overline{t}/\overline{s'}]\!] \end{aligned}$$

Finally, using $\star 11$ we get:

$$= \texttt{mkCTerm}(o, n, a, \lambda i.\ \texttt{mkBndCTerm}(\overline{x'_i}\rfloor, c_i[\![\overline{s'}, \overline{x'_i}/\overline{u_i}, \overline{v_i}]\!]\rfloor))\restriction[\![\overline{t}/\overline{s'}]\!]$$

Our final term has the desired $B[\![\overline{t}/\overline{X}]\!]$ form, and now the B and the $\overline{X}$ parts are independent of $\overline{t}$. This is because:

- $\overline{x'_i}$ depends only on c_i;
- $\overline{s'}$ depends only on $\overline{x'_i}$ and $\overline{c}$, and therefore only on $\overline{c}$;
- and $\overline{u_i}$ and $\overline{v_i}$, just like $\overline{c}$, were derived from the assumption that the inputs are substitution functions.

QED.

7 Conclusions

The construction of the Term type was done to facilitate exposing internal Nuprl functionality to Nuprl users, which, it is hoped, will lead to a lightweight reflection implementation. We have shown the plausibility of basing logical reflection on higher-order abstract syntax, where each syntactic operator is denoted directly by another operator.

We are continuing the implementation of reflection in the Nuprl system along these lines, and hope to soon test this conjecture. The core rules reflecting syntax that we showed correct here, are already implemented, reusing existing internal functionality, without involving concrete syntax. Initial examples have indicated that it is, in fact, useful.

8 Acknowledgments

Thanks to Robert Constable for numerous discussions on these and closely related topics.

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia, Pennsylvania*, pages 95–105, Los Alamitos, California, June 1990. IEEE Computer Press Society.
2. Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.
3. Eli Barzilay. *Implementing Reflection in Nuprl*. PhD thesis, Cornell University, to appear in 2002.
4. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
5. Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, New York, 1992.
6. Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised 5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.
7. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
8. Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, pages 230–255, 2000.