Developing Tools for Reflection

Eli Barzilay
eli@cs.cornell.edu

Cornell University
General

Present current (ongoing) work on the reflection implementation in Nuprl.

- This work still follows the principles I talked about before.
- The main point: use strong reflection rather than the more conventional weak approach.
- We finally got to the stage of enjoying results of this work.
- The development demonstrates the benefits and the implications that were discussed previously.
General

- Will be included in the Formal Digital Library pages.
- These principles are general — they were borrowed from PL, and could apply to other provers as well (e.g., MetaPRL, PVS, Coq).
- No dependency on specific system features such (e.g., type system, computational system), but for systems closer to Nuprl have more directly applicable material.
- MetaPRL is a specific case where such an implementation might be easier due to better term management functionality (for example, pattern matching and rewrites).
The example that was used to motivate the current developments is a formalization of Tarski’s theorem (undefinability of truth).

As long as we don’t have an actual reflection rule, there is no more implications of this except for a more rigorous proof which is easier to maintain.

When we add a reflection rule, this same Nuprl proof can demonstrate Nuprl’s limitation as a logical system.

The presentation is somewhat random in its nature: I try to follow the development chain as it happened.
Tarski: Paper Proof

Start with a “paper proof” that Stuart wrote. Continue by going over the different bits and pieces, sort of a guided tour.

The following is Stuart’s text, slightly edited for clarity.

(Only relevant parts will be discussed, see the postscript file for the full version.)

Here’s my sketch of a Tarski result about truth not being reflected. We’re assuming we have the type of terms and a "reps" relation between terms.
Tarski: Paper Proof

a. We assume that if t reps s then t is closed.

Notation and some simple corrolaries (indicated by "Thus"): There are also assumptions about substitution into \( \text{SUBX}(?,?) \) and \( Q(?) \).

a1. \(-x-\) is a variable
a2. \(t/e\) is substitution of term e for variable \(-x-\) in t
a3. \(\text{SUBX}(t,r)\) reps \(t'/r'\) if t reps \(t'\), and r reps \(r'\)
a4. \(\text{SUBX}(t,r)/e = \text{SUBX}(t/e,r/e)\)
a5. \(q(t)\) reps t
a6. \(Q(t)\) reps q(r) if t reps r.

b. Thus, \(Q(q(t))\) reps q(t).

b1. \(Q(t)/e = Q(t/e)\)
c1. \( f(t) \) is \( \text{SUBX}(q(t), \text{SUBX}(-x-, \text{Q}(-x-))) \)

\[ c2. \ s(t) \text{ is } f(t)/q(f(t)) \]

\[ c3. \text{ Thus, } s(t) \text{ is } \text{SUBX}( q(t), \text{SUBX}( q(f(t)), \text{Q}(q(f(t)))) ) \]
   \hspace{1cm} \text{by (a) on } q(t)

\[ c4. \text{ Thus, } s(t) \text{ reps } t/(f(t)/q(f(t))) \text{ by (b)} \]

\[ c. \text{ Thus, } s(t) \text{ reps } t/s(t) \]

\[ d1. \text{ NOT}(t) \text{ is the term built from term } t \text{ by the negation-denoting operator} \]

\[ d. \text{ Thus } \text{NOT}(t)/e = \text{NOT}(t/e). \]
The Tarskian Argument:

Let \( \text{TrRep}(L, \text{Tr}, tr) \) where \( L \) and \( \text{Tr} \) are properties of terms and \( tr \) is a term, mean:

1. \( \forall S: \text{term. } L(tr/S) \) if \( S \) reps some term
2. \( \& \forall t: \text{term. } \text{Tr}(\text{NOT}(t)) \) iff \( L(t) \) and not \( \text{Tr}(t) \)
3. \( \& \forall S, t: \text{term. } \) if \( S \) reps \( t \) then ( \( \text{Tr}(tr/S) \) iff \( \text{Tr}(t) \) )

This is meant to be part of the criterion for \( \text{Tr} \) being truth on \( L \), and for \( tr \) to denote \( \text{Tr} \) (in -x-).
Then there are no Tr, tr such that TrRep(Tr, tr) thus:

4. Assume TrRep(L, Tr, tr)

5. let S = s(NOT(tr))

6. S reps NOT(tr/S) by (5,c,d)

7. L(tr/S) by (4,1,6)

8. Tr(tr/S) iff Tr(NOT(tr/S)) by (4,3,6)

9. Tr(NOT(tr/S)) iff L(tr/S) & not Tr(tr/S) by (4,2)

10. Tr(NOT(tr/S)) iff not Tr(tr/S) by (9,7)

*. Tr(tr/S) iff not Tr(tr/S) by (8,10)

... which is false so (4) is false.
The top-level theorem is easy to write:

\[ \neg(\exists \text{Tr}: \text{Term} \rightarrow \mathbb{P}. \exists \text{tr}: \text{Term}. \exists \text{L}: \text{Term} \rightarrow \mathbb{P}. \text{TrRep(L; Tr; tr)}) \]

But any attempt to continue this requires reflection tools. We will now continue by an overview of existing functionality, followed by new material that was implemented.
The important part in this theorem is the definition of the \texttt{TrRep}, the predicate for truth-representating terms.

\[
\text{TrRep}(L; \ Tr; \ tr) == \\
(\forall S:\text{Term Term.} \\
(\exists t:\text{Term.} \ S \downarrow= t) \implies L \ subx(tr; \ S)) \\
\land (\forall t:\text{Term.} \ Tr \ not(t) \iff L t \land \neg(Tr t)) \\
\land (\forall S:\text{Term Term.} \ \forall t:\text{Term.} \ S \downarrow= t \implies \\
(Tr \ subx(tr; \ S) \iff Tr t))
\]
Background Functionality

There is a lot to consider before we can begin stating the main facts:

- Terms are the major objects used, so term representation is the first thing we need.
  (in the proof, term representations are indicated by capitalized names.)

- We also need tools to “shift” representation levels up.
  (the \( \alpha \) operator.)

- Term substitution is also needed.
  (the \(?/?\) and \texttt{SUBX} (its representation) operators.)

- More bits are not not mentioned literally: a ‘Term’ type, various well-formedness facts, a representation relation, etc.
**Existing Functionality: Term type**

We already have a ‘Term’ type, with most of the baggage required for a Nuprl type.

- Based on *operator shifting*, keeping the original binding structure, resulting in a HOAS representation.
- Enable *descriptions* of terms, where some parts are not literal quotations.
- Implemented using a new parameter type (‘rquote’), and *system-level functions*: ‘quote’ and ‘unquote’.
- These functions are accessible from the editor, to generate literal quotations.
- New visualization code, using colors.
Some reminders:

- Formulated by the simple idea of a substitution function:
  \[
  \text{is\_subst}_n(f) \equiv \\
  \exists b : \text{Term}. \exists \overline{v} : \text{Var}^n. \forall \overline{t} : \text{Term}^n. f(\overline{t}) = b[\overline{t}/\overline{v}]
  \]
  also,
  \[
  \equiv \exists b : \text{CTerm}. \exists \overline{v} : \text{CVar}^n. \forall \overline{t} : \text{CTerm}^n. f(\overline{t}) = b[\overline{t}/\overline{v}]
  \]

- \( \lambda(x. \text{if } x = 0 \text{ then } 1 \text{ else } 2) \) is not a substitution function — this is an example of an exotic term.
Existing Functionality: Term type

- Membership in Term is defined by the `is_subst` rule:
  - $H \vdash \text{is}_\text{subst}(x_1, x_2, \ldots, x_n. x_i)$
  - $H \vdash \text{is}_\text{subst}(\overline{x. \text{opid}(y_1. b_1; \ldots; y_n. b_n)})$ where `opid` is some quoted opid
    - $H \vdash \text{is}_\text{subst}(\overline{x, y_1. b_1})$
    - $H \vdash \text{is}_\text{subst}(\overline{x, y_2. b_2})$
    - \vdots
    - $H \vdash \text{is}_\text{subst}(\overline{x, y_n. b_n})$

- Main idea: bindings that belong to shifted operators can only be used as “template holes” — similar to the idea of level expressions being “inert values” that cannot be used, so there is no way to have a simple decomposition rule (leads to exotic terms).
Existing Functionality: Up/Down

As seen by the paper proof, user-level operations that shift values up and down are needed — the ‘$q$’ in the proof.

- Without these operations, all we have is various compositions of already-quoted literals.

- The ‘quote’ and ‘unquote’ functions cannot be accessible at the user-level: they work on concrete representations available from within the implementation but transparent to users.

  (\texttt{natnum\{3:n\}()} is not a term at the user level.)

- Implemented a while ago, before the formalization of the Term type.
Existing Functionality: Up/Down

The computation rule is very simple:

\[ \downarrow \uparrow x \rightarrow x \]

\[ \downarrow \text{foo}(x.b[x]) \rightarrow \text{foo}(x.b[\uparrow x/x]) \]

where \( \text{foo} \) is the shifted version of \( \text{foo} \)

and analogously:

\[ \uparrow \downarrow x \rightarrow x \]

\[ \uparrow \text{foo}(x.b[x]) \rightarrow \text{foo}(x.b[\downarrow x/x]) \]

where \( \text{foo} \) is the shifted version of \( \text{foo} \), and it is canonical

Follows the same intuition: bindings are kept at the same quoted level they originally had.

(In fact, \texttt{is\_subst} explains why this is required.)
New Abstraction: \texttt{reps} Relation

- Begin with this simple statement from Stuart’s proof:

\[ q(t) \texttt{ reps } t \]

- We already have ‘up’ for the ‘\( \downarrow q \)’ operator, what we need now is a representation relation for ‘\texttt{reps}’.

- It seems natural that if \( x \) represents \( y \), then \( \downarrow x = y \), so it is used as the definition for the representation relation, which is displayed as \( x \downarrow\downarrow y \).

- This decision leads to making ‘\( \downarrow\downarrow \)’, like ‘\( \downarrow \)’, have another argument for the type.

\[ \star \texttt{reps: } x \downarrow\downarrow y \in t \Rightarrow \downarrow x = y \in t \]
New Rule: down_up

- Another necessary bit is nailing the relation between the level shifting operators.

- Add a rule that states that \( \downarrow \uparrow x \) is \( x \).

- There is no way to prove it in Nuprl, since we cannot evaluate \( \uparrow x \) when \( x \) is not a quoted literal.

  (but we could have proved it for any specific literal.)

- \* down_up: \( \forall t: \text{Term}. \ \downarrow \uparrow t = t \in \text{Term} \)

- Question: wouldn’t this be true for any type?
New Theorem: up_reps

- We now have enough functionality to prove a theorem relating ‘\(\downarrow=\)’ to the ‘up’ operator.

- \textbf{up_reps}: \(\forall t: \text{Term. } \uparrow t \downarrow= t \in \text{Term}\)

- This is proved using the ‘\texttt{reps}’ definition and the ‘\texttt{down_up}’ rule.

- This is the translation of ‘\(q(t) \texttt{ reps } t\)’, but it is included in the ‘reflection’ theory.
The next thing to translate is

‘Q(t) reps q(r) if t reps r’.

This is a good example for using the system to clarify confusions: it is not clear which of these should be the translation:

\[ \forall t,r: \text{Term. } t \Downarrow r \Rightarrow \uparrow \uparrow t \Downarrow \uparrow r \]

\[ \forall t,r: \text{Term. } t \Downarrow r \Rightarrow \uparrow t \Downarrow \uparrow r \]

Stuart used ‘Q’, which indicates ‘\(\uparrow\)’ is the right choice, so try the first one to see if we fail.
Verify Failure: upup_reps

Indeed, we get the following chain:

\[ t \downarrow = r \quad \Rightarrow \quad \Uparrow \Uparrow t \downarrow = \Uparrow r \]

(unfold ‘reps’)

\[ \downarrow t = r \quad \Rightarrow \quad \downarrow \Uparrow \Uparrow t = \Uparrow r \]

(by ‘down_up’)

\[ \downarrow t = r \quad \Rightarrow \quad \Uparrow t = \Uparrow r \]

The clarification is that the difference between ‘\( \Uparrow \Uparrow x \)’ and ‘\( \Uparrow x \)’ is that the former will shift ‘x’ up too.
New Theorem: \texttt{qup\_reps}

- So continue with the correct version:

  \[
  \forall t, r: \text{Term. } t \downarrow = r \Rightarrow \uparrow t \downarrow = \uparrow r
  \]

- Something new that is needed to prove this: we need to do some computation steps to show that `\downarrow \uparrow x` evaluates to `\uparrow x`.

- This almost succeeds, but there are two more well-formedness facts needed.
New Rule: \textit{up\_wf}

- We need to know when an ‘up’ term is well formed.
- This question opened up a old can of poisonous worms: on which types should ‘up’ be allowed to work?
- Finally, the safe version which was chosen is:
  \begin{itemize}
  \item \texttt{up\_wf: } \forall t : \texttt{Term}. \uparrow t \in \texttt{Term}
  \end{itemize}
- More in this will follow shortly.
New Functionality: `termof`

- The second well-formedness bit that is required for `qup_reps` is to know what condition should hold for \( t \) so we can safely know that \( \downarrow t \in \text{Term} \).

- For this, we define a new type constructor: for a type \( A \), the new type \( \text{Term}_A \) is defined as a set type:

  \[
  \text{termof}: \ A \text{ Term} \equiv \{ t : \text{Term} | t \downarrow \in A \}
  \]

- This is defined using a new built-in predicate `'termin'` (displayed as `\( \downarrow \in \)`).

- This must be a predicate since testing for membership is useless (always true when well-formed).
New Rule: \textit{termin\_member}

The new $\text{Term}_A$ type is given meaning using the following rule:

\begin{align*}
\forall a : \text{Term}. \quad \forall A : \mathbb{U}. \quad a \downarrow \in A \Rightarrow \downarrow a \in A
\end{align*}
Addition to Term

Another missing piece pops when we try using ‘up_wf’.

Using the previous definitions for Term meant crawling over a quoted term, but what if some of its parts are some descriptions with known facts? Examples:

\[
\begin{align*}
  t & : \text{Term} \vdash \uparrow t \in \text{Term} \\
  x & : \text{Term} \vdash x + 2 \in \text{Term}
\end{align*}
\]

We need to add another rule:

* isSubstTerm:
  \[
  t \in \text{Term} \Rightarrow \text{is}\_\text{subst0}(.t)
  \]
Addition to Term

The new rule is used in combination with the previous is_subst rule by a tactic — ‘TermAuto’ which invokes the appropriate rule.

Still incomplete: we also need to be able to prove:

\[ x : \text{Term} \vdash \lambda y. x + 2 \in \text{Term} \]

Requires another rule that needs to be implemented: an ‘isSubstThinVars’ that will eliminate vars that do not appear free in the body of the is_subst goal.

‘TermAuto’ will be extended to use it too.

Note that is_subst is still not necessary.
Typing ‘up’

Back to the type of ‘up’ issue.

- The version we have: $\forall t:\text{Term}. \uparrow t \in \text{Term}$
- But originally, I was hoping for a stronger version:
  $\forall u:U. \forall t:u. \uparrow t \in \text{Term}$
- Two related problems with this: what about diverging terms and function bodies.
- Previously, I thought I could get away with no type, some weird term that we only know how to compute (similar to ‘$\text{ycomb}$’, no wf).
- Unfortunately, we can’t sweep it under the rug this way.
Typing ‘up’

There are some additional ‘concrete’ types that we can ‘up’: integers, booleans, tokens, and pairs/lists/etc of concrete types.

If we had some type predicate that would determine when a type is ‘concrete’, we could use it.

This is related to bar types which are related back to the basic problem of diverging terms.

So it seems that it is an unrelated problem that needs to be silved separately, and maybe such an addition can be exploited with the reflection implementation.
Typing ‘up’

Obviously, this whole thing depends on how ‘up’ is computed, if we remove the restriction that it works only on canonical terms, we get a quotation operator which should not be user-visible. (e.g., we cannot have \( \uparrow 1 + 2 \neq \uparrow 3 \))

Another approach that I considered: saying that \( \uparrow x \) is defined when we know that there exists some \( y \) such that \( \downarrow y = x \). But the intended usage for ‘up’ on these cases was a “future promise” to cancel with a ‘down’ — so if we know \( y \), there is no point in such a promise. . .

Last intuition: restricting ‘up’ on Term inputs is reflects the fact that the internal quotation function can work only on terms.
Quickref

* reps:  \( x \downarrow = y \in t == \downarrow x = y \in t \)
* down_up:  \( \forall t: \text{Term. } \downarrow \uparrow t = t \in \text{Term} \)
* up_reps:  \( \forall t: \text{Term. } \uparrow t \downarrow = t \in \text{Term} \)
* qup_reps:  \( \forall t, r: \text{Term. } t \downarrow = r \Rightarrow \uparrow t \downarrow = \uparrow r \)
* up_wf:  \( \forall t: \text{Term. } \uparrow t \in \text{Term} \)
* termof:  \( A \text{ Term} == \{ t: \text{Term} | t \downarrow \in A \} \)
* termin_member:
  \( \forall a: \text{Term. } \forall A: \mathbb{U}. \ a \downarrow \in A \Rightarrow \downarrow a \in A \)
* isSubstTerm:
  \( t \in \text{Term} \Rightarrow \text{is}_\text{subst}0(.t) \)
To be continued...

- term_downeq
- termsubst
- the proof...