Implementing Reflection in Nuprl

Eli Barzilay

eli@barzilay.org

Cornell University
Importance of thesis topic: Reflection

Reflection is a deep idea in Logic, Computer Science, Natural Language and more

(Gödel incompleteness, paradoxes, key idea in Lisp/Scheme)

Reflection is currently interesting (PoplMark, PL)

Specifically, reflection is practical in a theorem prover:

- allows formalizing meta-mathematical proofs
- can be used to speed proofs, extend the system with verified tactics, meta-proof tools (ACHA90, Knoblock)
- Reasoning about syntax, including programming languages (as a by-product of syntax representation) (PoplMark)

Has been trying for years to achieve reflection in Nuprl.
Requirements for Practical Reflection

We want to implement reflection in Nuprl in a way that is:

- efficient
- elegant
- robust
- practical
Thesis results and Impact

- Direct reflection of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms (BTW: HOAS)
- A practical quoting scheme: operator shifting
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Reflection...

Always involves an object and a meta language
In the context of formal languages we need to extend the semantic domain to include syntactic objects:

- ...must allow syntax that denotes (by its semantics) its own syntactic constructs (quotations)
- ...must have functionality for these constructs (PL, TP, are dynamic systems)

To make this a reflection:

must connect meta level and the object level
Thesis results and Impact

- *Direct reflection* of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms (BTW: HOAS)
- A practical quoting scheme: operator shifting
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
My Approach

Direct reflection vs. Indirect reflection

- Direct reflection: expose the actual implementation to the user-level.
  (e.g., Scheme/Lisp)

- Indirect reflection: re-implement functionality in your system.
  (e.g., a meta-circular evaluator)

Direct reflection: the obvious choice in PL, not so in logic (various reasons)
Thesis results and Impact

- **Direct reflection** of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms (BTW: HOAS)
- **A practical quoting scheme: operator shifting**
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Our working environment: Nuprl Terms

Nuprl terms have uniform structure:

- Operator
- Opid
- Parameters
- Bound subterms
- Value: tag
- Bindings: term

\[ \text{foo}\{\text{bar}: t, 3:n\} (x, y, z. t_1; x. t_2) \]

- Or \( \text{opid}\{v:f, \ldots k\} (x, \ldots m. t; \ldots n) \)

- Operators are a discrete, open-ended set
- Explicit binding positions
- Exist only at the meta-level
Bindings

- Bindings are strings, a variable reference is a 'variable' term with a string parameter value.
- Example: ‘λx. v + 1’ is

  \[
  \text{lambda}(x. \text{add(variable\{x:v\};
  \text{natural_number\{1:n\}}))
  \]

- Related through the family of variable name parameters (‘v’ tag)
- Sophisticated substitution (consistent with user names)
Interaction: Editing & Display Forms

- Work through a structure editor (complex, expansible)
- Display forms (important for Nuprl work)

\[ \lambda x, y. y \] vs. \[ \lambda{}(x.\lambda{}(y.\text{variable}{y:v}())) \]

\[ \text{ALL}(\mathbb{N}; \ x. x < x + 1) \] vs.

\[ \text{all}()(.\text{nat}()(); \ x.\text{less}\_\text{than}()(.\text{variable}{x:v}(); \ .\text{add}(.\text{variable}{x:v}(); \ .\text{natural}\_\text{number}{1:n}()))) \]

- Different renderings for different users
  (compare Eli's style with Stuart's)
Some Key Features of Nuprl

- **Substitution**
  implementation, no hand waving

- **Evaluation** (Crary and Constable)
  primitive, abstract, & canonical terms
  (untyped lambda calculus)

- **Library**
  abstractions, display forms, rules, theorems, ML
  (another large part of the system)

- **Interactive proof editor, tactics**

⇒ Eventually, want to reflect all (precondition: syntax)
Thesis results and Impact

- **Direct reflection** of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms (BTW: HOAS)
- **A practical quoting scheme: operator shifting**
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Advantages of Direct Syntactic Reflection

- Flat representations (strings, Gödel numbers) are bad: no structure
- Quote-like contexts fail (destroy congruence)
- Black-box representations cannot mix descriptions with literals

Standard Type Definitions (re-implementation)
- Naive approach: imitate the system in itself:
  ```
  make_lambda("x"; make_var("x"))
  ```
- Indirect reflection ⇒
  - Easy to fall in an exponential trap
  - Incomprehensible without preprocessing
  - Requires intense work, equivalence proof, type theory
Shifted Operators: Quasi-Quoting

Quasi-quotes are important (a modern version of Quine quotes)

- Cannot use contexts ⇒ our approach: shifted operators (comes from Allen and Aaron)
- To shift an operator we need to recursively add quotedness tags
- Things that are not shifted in this process retain their meaning

Scheme analogy: (’ + ’ x (’ * y ’ 2 ) )
Thesis results and Impact

- Direct reflection of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms (BTW: HOAS)
- A practical quoting scheme: operator shifting
- Implemented on top of CTT, no changes to the system, in
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Implementing Shifted Operators

- A new integer parameter value for quotedness,
- No parameter \(\equiv\) zero quotedness
- The big question:
  What do we do with bindings of shifted operators?
- The Lisp way: quoted variables are symbols
- In Scheme: problems with hygiene macros
- In Nuprl: the only way we could achieve true direct reflection is to keep variables in quoted (shifted) terms as variables — keep the same binding structure
  \(\Rightarrow\) we get a HOAS representation
Thesis results and Impact

- **Direct reflection** of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms *(BTW: HOAS)*
- A practical quoting scheme: operator shifting
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Implementing Shifted Operators

Did we just lose all explicit names?

From: Stuart Allen <sfa@CS.Cornell.EDU>
To: eli@CS.Cornell.EDU, rc@CS.Cornell.EDU
Subject: Tarski sketch
Date: Wed, 28 Feb 2001 15:44:59 -0500 (EST)

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

Notation:
-x- is a variable
Not(t) is the term built from term t by the negation-denoting operator
NOT(t) reps Not(r) if t reps r
sub(v,t,e) is substitution of e for variable v in t
subx(t,e) is sub(-x-,t,e)
SUBX(t,s) reps subx(r,p) if t reps r, and s reps p
q(t) reps t
Q(t) reps q(r) if t reps r. Thus, Q(q(t)) reps q(t).

f(t) is NOT( SUBX(q(t),SUBX(-x-,Q(-x-))) )
s(t) is subx(f(t),q(f(t)))

Thus, s(t) is NOT( SUBX( q(t), SUBX(q(f(t)),Q(q(f(t)))) ) )
Thus, s(t) reps Not( subx(t, subx( f(t),q(f(t)) ) ) )
0) Thus, s(t) reps Not(subx(t,s(t)))

Assume L is a language closed under Not(?).

Let FU(T,tr) where T is a property of terms and tr is a term, mean
1) forall s,r:term. L(subx(tr,s)) if s reps r
   & forall t:term.
   2) T(Not(t)) iff L(t) and not T(t)
   3) & forall s:term. if s reps t then ( T(subx(tr,s)) iff T(t))

Then there is not T,tr such that FU(T,tr) thus:

4) Assume FU(T,tr)
   5) s(tr) reps Not(subx(tr,s(tr))) by (0)
   6) L(subx(tr,s(tr))) by (4,1,5)
   7) T(subx(tr,s(tr))) iff T(Not(subx(tr,s(tr)))) by (4,3,5)
   8) T(Not(subx(tr,s(tr)))) iff
      L(subx(tr,s(tr))) & not T(subx(tr,s(tr))) by (4,2)
   9) T(Not(subx(tr,s(tr)))) iff not T(subx(tr,s(tr))) by (8,6)
      T(subx(tr,s(tr))) iff not T(subx(tr,s(tr))) by (7,9)
      which is false so (4) is false.

s
Implementing Shifted Operators

A few technicalities:

- ‘rquote’ parameter, ‘rquote_term’, ‘runquote_term’
- Editor level operations, work on the concrete level
- Heavy hacking for interface issues
Thesis results and Impact

- **Direct reflection** of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms *(BTW: HOAS)*
- A practical quoting scheme: operator shifting
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Implementing Shifted Operators

Visualizing quotations — display forms:

```
M> "\x.x + 1" ;;
```
Implementing Shifted Operators

Visualizing quotations — display forms:

```
M> \x.x + 1
```

Implementing Syntactic Reflection in Nuprl – p.22
Implementing Shifted Operators

Visualizing quotations — display forms:

![Image of a code output showing a lambda expression and a semicolon.](image-url)
Implementing Shifted Operators

Visualizing quotations — display forms:

```
M> \(\lambda x. x + 1\) ;;
```
Implementing Shifted Operators

Visualizing quotations — display forms:
Implementing Shifted Operators

Visualizing quotations — display forms:

```
top
\[ \forall t, r, e : \text{Term.} \]
\[ \text{subx(subx}(t; r); e) \]
\[ = \text{subx(subx}(t; e); \text{subx}(r; e)) \]
\[ \in \text{Term} \]
```
Thesis results and Impact

- Direct reflection of Nuprl’s syntax, follows requirements
- Setting a clear approach for direct reflection
- Carried common PL methodology to a theorem prover
- Efficient H.O. representation of Nuprl terms (BTW: HOAS)
- A practical quoting scheme: operator shifting
- Implemented on top of CTT, no changes to the system, including CTT’s open-endedness syntax
- Independent of the type theory (mostly)
- Provides a convenient user-interface
Semantics of Shifted Terms

Concrete representation would make ‘\(\lambda\)’ a function of two syntax objects (name+body) which constructs a ‘\(\lambda\)’ term.

The semantics of our representation uses functions, making it a \textit{higher-order abstract syntax}.

‘\(\lambda\)’ is a binding operator \(\Rightarrow\) takes a function as input.

‘\(\lambda x. F(x)\)’ instantiates \(\lambda\) expressions.
\(\Rightarrow\) \(F\) needs to be a \textit{substitution function}.

\(\Rightarrow\) We get the usual benefits
...and the usual headaches.
Semantics of Shifted Terms

The semantics we adopt for ‘\( \lambda x. F(x) \)’ is that it denotes the ‘\( \lambda \)’ term whose body is ‘\( F(u) \)’ and whose binder is \( u \) for some \( u \)

...the question is which \( u \)?

Using a HOAS: we avoid an answer, denote \( \alpha \)-equivalence classes

Use \( \text{Term} \) for the type that corresponds to \( \alpha \)-equivalence classes (\( \text{CTerm} // \alpha \))

(This work could not be done without Stuart.)
Semantics of Shifted Terms

First intuition: to verify Term-ness you need a shifted operator and check the subterms in a compositional way:

\[ \vdash \text{opid}(v, \ldots, b, \ldots) \in \text{Term} \]
\[ \text{if} \ v : \text{Term}, \ldots \vdash b \in \text{Term} \]
\[ \vdash \ldots \]

Fails with bound variables:

\[ \vdash \lambda x . \text{if } x = 0 \text{ then } 1 \text{ else } 2 \in \text{Term} \]

because \[ x : \text{Term} \vdash \text{if } x = 0 \text{ then } 1 \text{ else } 2 \in \text{Term} \]

The premise is trivial, but the statement is false
Semantics of Shifted Terms

The problem is *exotic terms*, compare:

- \( \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } 2 \) (bogus term)
- \( \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } 2 \) (valid term)
- \( \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } 1 \) (valid term)
- \( \lambda x. \text{if } x = 0 \text{ then } 1 \text{ else } 2 \) (valid function)

Conclusion:

a bound variable can be used only as an argument of a quoted term constructor (syntactic property)

- Addresses the known problem of *exotic terms*
- Difficult to find a formal semantic solution
Semantics of Shifted Terms

- **CTerm**: given concrete type
- **Term** $\equiv$ **CTerm**$/\alpha$ constructed by shifted operators, having the semantics of **Term** $\rightarrow$ **Term** substitution functions *e.g.*
  \[ \lambda : \{ f : \text{Term} \rightarrow \text{Term} \mid \text{is\_subt}_1(f) \} \rightarrow \text{Term} \]
- **CVar** is a subset of **CTerm**,
  \[ \text{Var} = \{ \{ x \} \mid x \in \text{CVar} \} \subseteq \text{Term} \]
  $\Rightarrow$ *so we do have quoted names*

\[\begin{align*}
\text{Examples:} \\
\lambda(f) &= \text{mkTerm}(\lambda', [\langle 1, f \rangle]), \\
\Sigma(f, g) &= \text{mkTerm}(\Sigma', [\langle 0, f \rangle, \langle 1, g \rangle])
\end{align*}\]
Semantics of Shifted Terms

Defining ‘is_subst’:

\[ is\_subst_n(f) \equiv \exists b : CTerm. \exists v : CVar^n. \forall t : CTerm^n. f(t) = b[t/v] \]

\[ is\_subst_n(f) \equiv \exists b : CTerm. \exists v : CVar^n. \forall r : Term^n. f(r) = b[r/v] \]

\[ is\_subst_n(f) \equiv \exists b_a : Term. \exists v_a : Var^n. \forall t_a : Term^n. f(t_a) = b_a[t_a/v_a] \]

⇒ all equivalent
(Note: based on known concept of concrete substitution)
Semantics of Shifted Terms

...Finally, the ‘is_subst’ rule:

1. \( H \vdash \text{is_subst}(\bar{x}. x_i) \text{ where } 1 \leq i \leq \text{len}(\bar{x}) \)

2. \( H \vdash \text{is_subst}(\bar{x}. \text{opid}(\bar{y}. b; \ldots )) \)
   where \( \text{opid} \) is some shifted opid
   if \( H \vdash \text{is_subst}(\bar{x}, \bar{y}. b) \)

Using it:

1. \( H \vdash t \in \text{Term} \)
   if \( H \vdash \text{is_subst}(. t) \)

\( \Rightarrow \) HOAS without heavy type definitions
\( \Rightarrow \) Convenient rule, easy to mix in descriptions
Using Shifted Terms

- Representation relation, \texttt{reps}
- Term induction (actually \texttt{SubstFunc} induction)
- Quotation ('\texttt{q}') and unquotation ('\texttt{unq}')
- '↑.' and '↓.'

\[
\begin{align*}
\downarrow^\uparrow t & \quad \text{evalsto} \quad \downarrow t \\
\downarrow \texttt{op}(x. b) & \quad \text{evalsto} \quad \texttt{op}(x. \downarrow (b[↑x/x])) \\
\uparrow\downarrow t & \quad \text{evalsto} \quad \uparrow t \\
\uparrow \texttt{op}(x. b) & \quad \text{evalsto} \quad \texttt{op}(x. \uparrow (b[\downarrow x/x])) \\
\end{align*}
\]

(canonical '\texttt{op}')
Induction Rule

We prove an induction rule on substitution functions:

\[ \forall P : (n : \mathbb{N} \rightarrow \text{SubstFunc}_n \rightarrow \text{Prop}). \]

\[ (\forall n : \mathbb{N}. \]

\[ (\forall i : 1\ldots n. \ P_n(\pi^n_i)) \]

\[ \& (\forall o : \text{Opld}, \ k : \mathbb{N}, \ a : (1\ldots k \rightarrow \mathbb{N}), \ f : ((i : 1\ldots k) \rightarrow \text{SubstFunc}_{n+a_i}). \]

\[ (\forall i : 1\ldots k. \ P_{n+a_i}(f_i)) \Rightarrow P_n(\text{mkSubstFunc}_n(o, k, a, f))) \]

\[ \Rightarrow (\forall n : \mathbb{N}, \ f : \text{SubstFunc}_n. \ P_n(f)) \]

The proof is very verbose, and requires a long setup.
Related work: Early Nuprl

Knoblock’s thesis: sketch for a reflected Nuprl

- Outline of reflecting Nuprl, focus on the meta-logic (‘Metaprl’) for user-created tactics
- Naive approach to reflecting syntax (indirect, not open-ended), but identifies many issues that need addressing (antiquotation)


- Design for a reflected logic
- Re-implementation leads to exponential blowup
Related work: HOAS

- Lots of HOAS-related work, became more popular recently (PoplMark, Merlin)
- All face the same fundamental problems: representation, induction, exotic terms, etc
- Usually solved through types, involves heavy math (modal logic, functor categories, set theory)
- Survey in my text
Related work: MetaPRL

- Based on this work
- Main change: use ‘bterm’ as the primitive type
- A (top-level) bterm is viewed as a term with free-variable context

Very recent work, continues to change... (Merlin’05)
Related work: MetaPRL

- Able to express terms in de-Bruijn style: ‘var’, ‘mk_bterm’
- These result from exposing more: the opid and arity
- Induction is tricky to define, easy to use
- Aims at abstracting over whole languages and classes of languages

⇒ lost some directedness of the reflected system, got some exponential blowup issues.
Related work: Nominal Logic

Use permutations as the main tool.

- Permutations avoid capture
  example: \((x \ y) \bullet (\lambda y \cdot y \ (x))\)

- Easy to deal with (reversible, list concatenation and reverse)

- Works on a very low level (binding positions are treated like the body)

- Variable names are used — but may lose their meaning when permuted
Related work: Nominal Logic

Very recent work by Urban et al builds on nominal logic (Urban-Tasson CADE’05, Urban-Norrish Merlin’05)

- Starts from the same low-level (the ‘support’ set consists of all atoms in a term)
- Introduces nominal abstractions that behave like \( \alpha \)-equivalence classes —but in definition:
  
  \[
  a \neq b \land x_1 = (a \ b) \cdot x_2 \land a \# x_2
  \]

  and in the construction of \( \Lambda \rightarrow \Phi \) functions for \( \Lambda_\alpha \):
  
  ‘if \( b \# t \) then \( a \ b \cdot t \) else er’

  —the freshness constraint essentially turns permutation into plain renaming

- The resulting \( \Lambda_\alpha \) setting is similar to our Term (fresh = free-in)
Related work: Nominal Logic

\[ \Phi \supseteq \Lambda_{\alpha} \leftrightarrow \text{bijection} \rightarrow \Lambda_{/\approx} \leftrightarrow \text{coincide} \rightarrow \Lambda_{/\equiv_{\alpha}} \]

Comparison:

- Not derived from simple known syntax principles: elegant, but lots of specific terms (Disagreement Set, Permutation Equality, PSets, Support, Freshness, Fs-PSets, Nominal Abstractions)
- Capture-avoiding substitution does not come for free
- Free induction principle, but not structural
- Currently limited to $\lambda$-calculus (recent work on $\pi$-calculus)
  \[ \Rightarrow \text{ Might work better if used in Nuprl due to uniform binding treatment} \]
- Some HOL/Isabelle problems (functions indirectly via relations)
Related work: Nominal Logic

Major feature of Urban’s work: customized induction principle uses context variables for Barendregt-style proofs with the variable convention

- May be possible to slap this on other HOAS inductions in the same way
- If MetaPRL’s `bterm’s work for this, it will make a more elegant solution
Conclusions

- Syntax can be reflected efficiently and robustly
- Major achievement: direct reflection as a methodology
- Already used — as the foundation for MetaPRL’s reflection
Future Work

- Usable induction rule
- Generalized up/down operations
- Formalize syntax-oriented texts
- Enhance general language-oriented reasoning (PoplMark)
- Possible to implement a fully reflected Nuprl, based on Knoblock and ACHA90
Thanks

- Members of my committee: Robert Constable, Greg Morrisett, Barry Perlus
- Stuart Allen who made the formalization possible
- Aleksey Nogin (early ideas, continues in MetaPRL)

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, and NSF Innovative Programming technology for embedded systems #CCR-0208536