Practical Reflection in Nuprl

Eli Barzilay, Stuart Allen, Robert Constable

{eli,sfa,rc}@cs.cornell.edu

Cornell University

Introduction & Motivation

- Reflection: useful in practice and theory.
- Simple PL solution: expose underlying meta-level features to the object-level directly — as **primitives**.
- Call this "**strong reflection**", opposed to "weak reflection" solutions of (re-)implementing features.
- We aim at a strong reflection implementation for a the Nuprl theorem prover, using the PL intuition.
- Main idea: Avoid functionality duplication.
  In our case: avoid recoding object-level functionality that already exists in the meta-level.
- Puts practical usage as a main goal.

Operator Shifting

- Nuprl has a uniform term structure — a term has a name and subterms (bindings later).
- First problem: Nuprl uses normalization rather than evaluation — we need a way to achieve quasi-quote functionality without a quoting context.
- We solve this using **operator shifting**: every operator op has a "shifted" version $\underline{op}$ that denotes syntactic constructions of op.
- For example: $\mathrm{foo}(11;12)$ is represented by $\underline{\mathrm{foo}(11;12)}$; we can say: $\forall x : \mathsf{Term}.\mathrm{sizeof}(\underline{\mathrm{foo}(x;12)}) = \mathrm{sizeof}(x) + 2$

Quoting with Bindings

- Actually, Nuprl terms have **bound** subterms — a subterm has a list of variables bound in it. For example: $\mathrm{all}(\mathrm{nat}();\ x.\mathrm{let}(\mathrm{add}(\mathrm{var}{:}x();\mathrm{num}{:}1());\ y.\mathrm{gt}(\mathrm{var}{:}y();\mathrm{var}{:}x())))$ which is displayed as: $\forall x : \mathbb{N}.\ \mathrm{let}\ y = x + 1\ \mathrm{in}\ y > x$.
- How should such terms be represented?
  - Obvious choice: $\underline{\mathrm{all}(\mathrm{nat};\mathrm{var}{:}x;\mathrm{let}(\ldots))}$.
  - No binding structure, so: $\underline{\mathrm{all}([0;1];\mathrm{nat};\mathrm{var}{:}x;\mathrm{let}(\ldots))}$.
  - Big disadvantage — the quoted term has a different binding structure, quotation is a complex operation.
- Breaks structure sharing with the implementation!

Quoting with Bindings

- The solution is natural: leave **bindings as bindings**. For example, quoting the previous term yields: $\underline{\mathrm{all}(\mathrm{nat}();\ x.\mathrm{let}(\mathrm{add}(\mathrm{var}{:}x();\mathrm{num}{:}1());\ y.\mathrm{gt}(\mathrm{var}{:}y();\mathrm{var}{:}x())))}$ which is displayed as: $\underline{\forall x{:}\ \mathbb{N}.\ \mathrm{let}\ y = x + 1\ \mathrm{in}\ y > x}$.
- Retain the ordinary semantics of binding operators: subterms with bindings denote functions. In the syntactic case, these are **substitution functions**.
- This makes it a HOAS, a less familiar approach, than concrete syntax.
  - Theoretically: Standard HOAS problems —eliminating exotic terms, induction is problematic.
  - Practically: Give up free variables? What interface?

HOAS

- Substitution functions are the core concept of our HOAS:

  $\texttt{is\_subst}_n(f) \equiv \exists b : \mathsf{Term}.\ \exists \overline{v} : \mathsf{Var}^n.\ \forall \overline{t} : \mathsf{Term}^n.\ f(\overline{t}) = b[\![\overline{t}/\overline{v}]\!]$

- Characteristic theorem:

  $\forall t : \mathsf{Term}.\ \exists!\ o : \mathsf{OpId}, k : \mathbb{N}, a : 1..k \to \mathbb{N},$

  $f : i{:}1..k \to \{g{:}\mathsf{Term}^{a_i} \to \mathsf{Term} \mid \texttt{is\_subst}_{a_i}(g)\}$

  $t = \texttt{mkTerm}(o, k, a, f)$

- A usage example:

  $\forall a : \mathsf{Term}, b : \mathsf{SubstFunc}_1.$

  $\underline{\mathrm{ap}(\lambda(x.\ b(x));\ a)}\ \texttt{reduces\_to}\ b(a)$

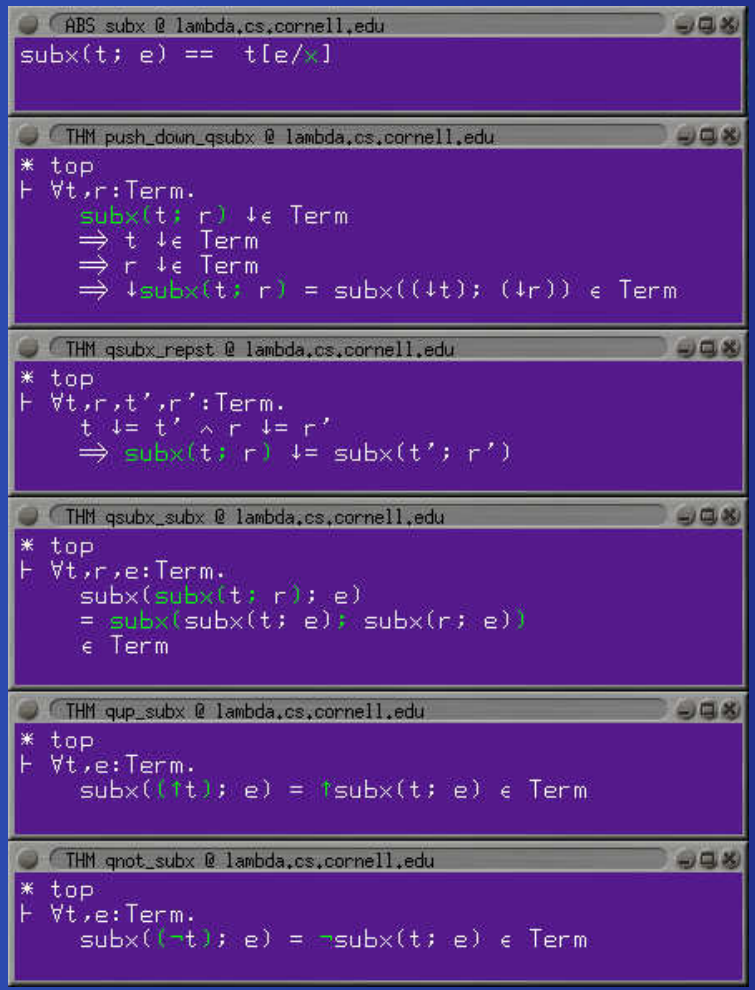
Tarski: Original Version

```
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
```

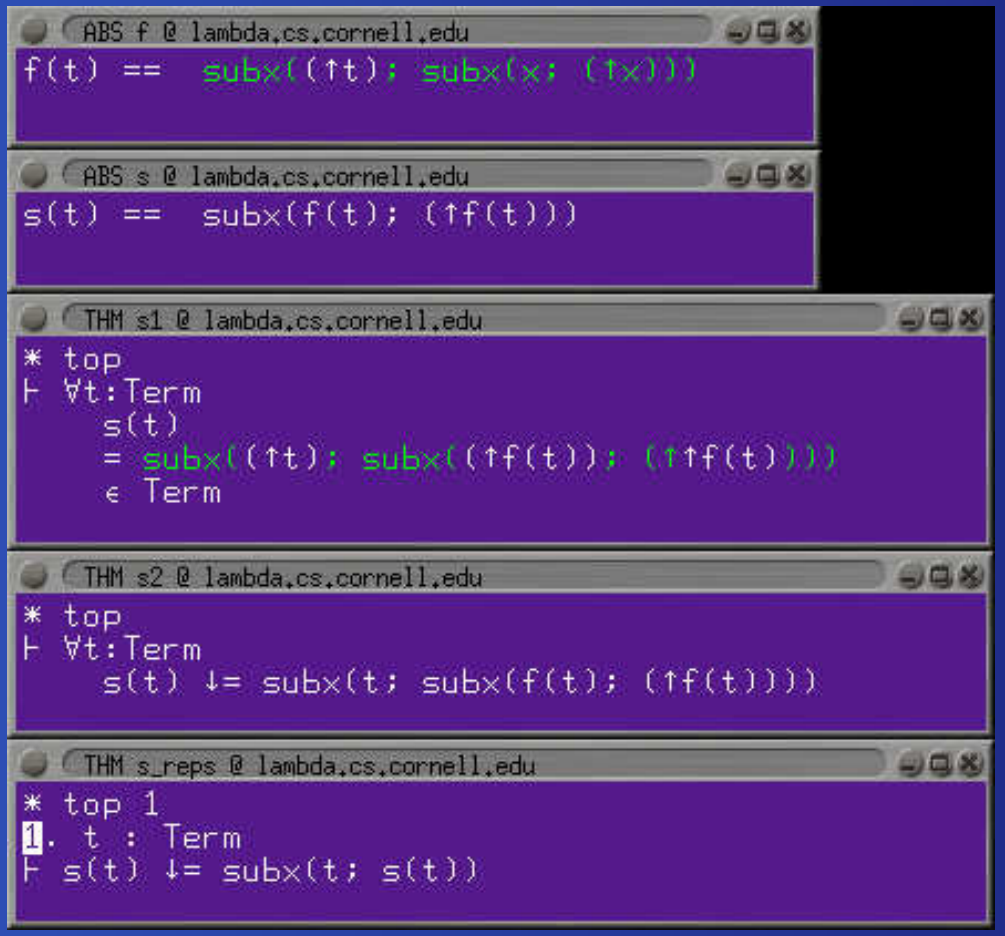
Nuprl Screen Shots

(Slides marked with a $\star$ are part of the presentation.)

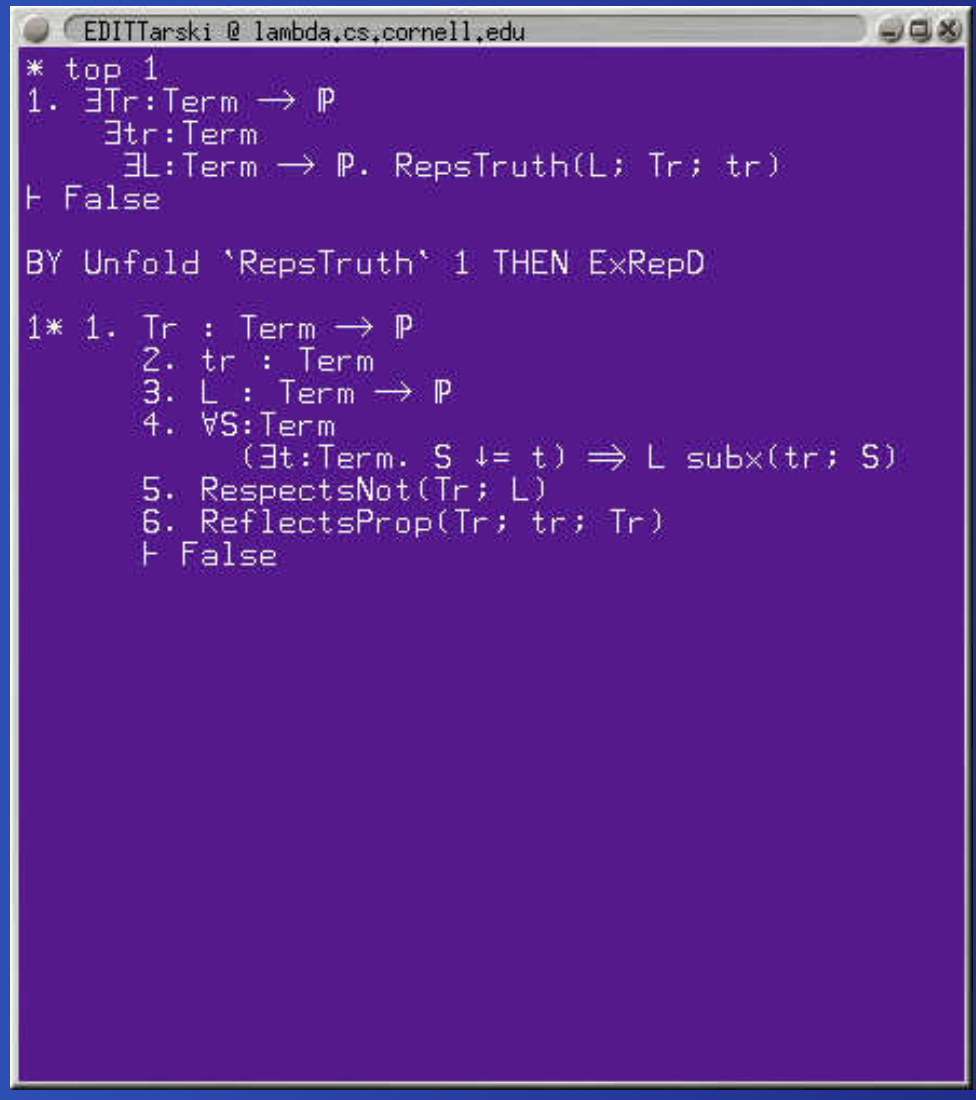
★**The definition of subx and some facts. Note the usage of a quoted free variable as a symbol. The last fact is a good example for intuitive usage that we will use shortly.**



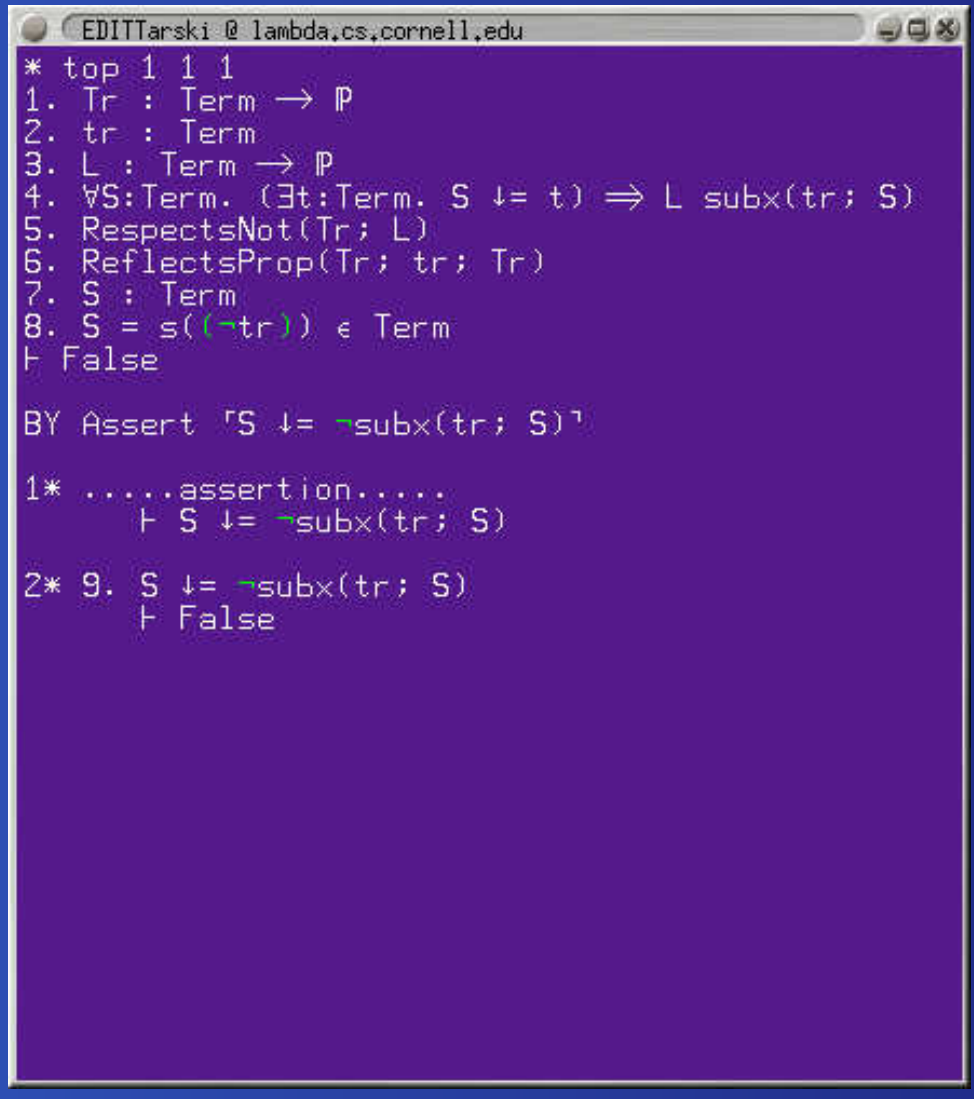
★A definition of an ‘s’ term which will be used to derive the contradiction by diagonalization. Note the clear similarity between the term in ‘s1’ and the one it represents in ‘s2’.



★The `Tarski` proof: unfolding the definition.



★The `Tarski` proof: asserting `S`'s property that will produce the contradiction.



Conclusion (& Motivation)

- **Cheap**: no new stuff — just expose existing functionality (only one implementation).
- **Robust**: the reflected system is inherently identical to the underlying one; changes propagate; no proof needed.
- Borrow ideas from PL: make object and the meta level share syntax objects and functionality.
- Still, extra facts and logical descriptions are needed — a lot of (unexpected) hard work.
- Result: elegant implementation.