April 3, 2000

Eli Barzilay

[part II]

Reflections & Visions
General ideology: duplication of functional information is bad.

- Clarity of the connection between the user's eval and the implementation's eval — trivial.
- The user shares the implementation's data structures.
- Also, the eval function is shared.

Consequent: the recursive type alone is not a good solution — it means that we duplicate the encoding of term structures that is already part of Nuprl itself.

The value Scheme's reflection is:

- Clarity of the connection between the user's eval and the implementation's eval — trivial.
When it comes to implementing Scheme, these shortcuts are natural:

- The implementation uses data structures and code that behaves like Scheme’s data structures: why use a different representation and translate back and forth?
- Even clearer when considering \( \text{eval} \): execution of quoted execution of code by \( \text{eval} \) is expected to be the same as execution of code, so using the same function is the obvious solution.
The Problem with Logic

The simplicity of Scheme is not seen in a logical system: 

- We must deal with truth and proofs as well as evaluation.
- This is hard so: we tend to avoid changing the logic — a strong point for using Gödel numbers is that the logic itself is not extended.
- My intuition: when it comes to an implemented, Scheme's insights are extremely useful because they will make reflection practical (as I will show).
Scheme's Approach

Avoid the evil of duplicating function in logic: code representation and evaluation are shared with users.

It is trivial to get convinced that eval's implementation is correct assuming that the language implementation is itself correct.

Avoid the [double] evil of duplicating function in logic: code representation and evaluation are shared with users.
Natural language is very different than formal language, but it is still a good source of intuition. There are several indications that we use the same parse mechanism for natural language:

- Parsing is done in "real-time" — the brain is doing the same mental activity (makes being a simultaneous translator difficult).
- If quotations were parsed in a different way we would take them as a different language.

Natural language is very different than formal language, but it is still a good source of intuition.
Try to understand things following some very general example.

Assume some system $\mathcal{S}$ (any kind of system).

In systems that we are interested in, the definition/design will have several parts with some dependencies.

Assume $\mathcal{S}$ is defined in three stages $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ and that each stage depends on the previous. (For example, defining Nuprl's syntax, semantics and rules).
Now we want to get $S'$ — a reflected version of $S$ within itself.\[\]

Finally, we have $S'$, and now we should show that it is exactly the same as $S$.\[\]

After all this, we get what we want: since $S'$ is exactly the same as $S$, we can do the same and get $S''$, etc.\[\]

And continue with $S^i_1$ in $S$ using $S^i_1$ and the same for $S^j$.\[\]
If we want to actually have an infinite chain of such embedded systems, we will have to use some circularity (or an infinite piece of paper). This can be achieved using the method described earlier. Specifically, a fixed-point construction or implementation shortcuts can do this.
The problem: we have both a complete implementation of \( S \) in \( S \)'s implementation of \( S \) and another complete implementation of \( S \) in \( S \)'s implementation of \( S \). This is the place that programmers should feel that something is duplicated. It doesn't feel good implementing something that we already did. An implementation of \( S \) in \( S \)'s implementation of \( S \) in \( S \)'s implementation of \( S \) doesn't necessarily go wrong and that logicians wouldn't necessarily be concerned about — it doesn't feel bad implementing something that we already did.
General Example

One way we can try to solve this — show that some is equivalent to another, and then use it to continue, for example:...

do the same thing until we have $\mathcal{S}'$

show that $\mathcal{S}'_1$ and $\mathcal{S}'_2$ are equivalent to $\mathcal{S}'_1$ and $\mathcal{S}'_2$

wrap these two in an interface that $\mathcal{S}'$ can use,

and finally use $\mathcal{S}'$ itself.

This is one thing that Aiken could have used to reduce some work.

One way that we can try to solve this — show that some is equivalent to another, and then use it to continue, for example:...

do the same thing until we have $\mathcal{S}'$

show that $\mathcal{S}'_1$ and $\mathcal{S}'_2$ are equivalent to $\mathcal{S}'_1$ and $\mathcal{S}'_2$

wrap these two in an interface that $\mathcal{S}'$ can use,

and finally use $\mathcal{S}'$ itself.

This is one thing that Aiken could have used to reduce some work.
Took some work off — no need for $S^1$, but still need to:

- Show that $S^1$ and $S^1$ are the same as those in $S$.
- Make sure that $S^2$ is accessible by itself.
- Wrap $S^2$ somehow.
- Makes this very unnatural.

We could do the same right after we get $S^1$. But it is better if don't do anything at all: just make shortcuts from $S^1$ into itself.

General Example
General Example: The Scheme Side

```scheme
(define user-eval (compose rep tmp-eval unrep))

- Wrap `eval` in these:
  - Internal `eval` structures,
  - Write translation functions to `(unrep)` and from `rep` the
    - Make sure they are equivalent to Scheme's,
    - Create some new data structures in Scheme,

Partial Shortcut:

Full Scenario: A complete Scheme parser/interpreter in Scheme.

52: `eval` function.
51: Syntax structures,

This is exactly what happens in Scheme:
```
Better stop and think before trying to implement a reflected system.

Exposing some internals of a system into itself is the best way to get reflection.

This might require some changes to the design of the system.

Since the exposed internals usually use mechanisms that have no equivalent in the world of its users (for example, terms in Nuprl),

General Example: Conclusion
The Nuprl Side

- Can split further to a more complex DAG.
  - Inference rules.
  - Semantics/Evaluation.
  - Syntax.

The definition stages we have are basically

My strong intuition here is that this applies perfectly to Nuprl.
Moreover, such shortcuts should also be useful for the "paper implementation." The "Nuprl Side"

- Coq? HOL?
- Classic tradeoff.
- Stuff we can do now.
  - If we'd use 0 and succ, we wouldn't be able to do a lot of
    - are OK.
  - Risk: Nuprl's number behave correctly only if Lisp's numbers
    - Goes down to Lisp numbers.
  - A similar case — Nuprl uses its implementation's numbers:

- A side-effect effect is that implementation details all pop up in
  - Implementation — the design.
  - The design stage (because we're mixing the two).

Eli Barzilay

Reflection & Nuprl
The suggested way of doing quotations was — add quotes parameter, with a value that indicates the quoted-ness level of the term. Choosing a quotation mechanism turns out to be a crucial decision — looks like this one is The Right Thing.
The Hard Way

Quotations

Term Quotations: The Hard Way

- Mentioned earlier: use a recursive type definition: an opid and a pair of lists.
- Relies on having an available type theory, bad for Meta-PRL.
- Reprsentation gets exponentially big when a term is quoted over and over, a Scheme example shows this easily:

Original expression: \((\text{foo} \ x)\)

Quoted once: \((\text{list} \ \text{quote} \ \text{foo} \ x)\)

Quoted twice: \((\text{list} \ \text{list} (\text{list} \ \text{quote} \ \text{foo}) (\text{list} \ \text{quote} \ x))\)
The single advantage of this is that the logic is not extended.

- Implementation down the wrong path — Aitken's case.
- The main damage this causes is that it leads the whole
  implementation down the wrong path — Aitken's case.
- However, it is still as efficient as using 0 and succ for integers.
- I/O mechanisms.
- The user-interface problem can be avoided using appropriate...

Term Quotations: The Hard Way
Term Quotations: Another Alternative

An example for some suggested quotations that didn’t “go all the way”.

This is a good example to get the feeling of implementation shortcuts being much better.

Make binding variables of subterms be quoted on their own as well.

But these are ignored now.

Easily accessible (actually, level-expression variables as well, more subterms — supposed to make binding variables more...

So the above term is quoted as:

\[
(\overline{y} \; \overline{z} \; \overline{x} \; \overline{y} \; x) \{ x : t \} : t \; t : g \; t : i \}
\]

Now the fun begins.

•
First of all, we have to be really careful about the representation:

integer quote parameter can prevent exponential blowup.

Also, pulls toward the recursive type solution (except that the

This, again, depends on type theory for the lists.

That describes the quoted term’s binding structure.

Solution: make the quote parameter contain more information

subterms are quoted subterms and which are quoted bindings.

in the above quotation there is no way to know which quoted

Alternative: Problem #1
Using this quotation method, it is possible to quote a term without any of its subterms. For example, quote only the "A" in \( \forall x \in \mathbb{R} \).
and this has no sense as a quoted term.

\((\overline{f} \, \overline{z} \, \overline{y} \, \overline{z} \, \overline{A} \, \overline{z} \, \overline{w} \, \overline{z} \, \overline{B}) \{ t : Z, \overline{t} : Z \} \) 00 \{ t : Z, \overline{t} : Z \} \)

So, it is possible to enter:

has

unrestricted — we can't control what subterms a quoted term has.

Another problem pops up because these quotes are too

Alternative: Problem #3
Advantages

- Almost completely independent of the underlying logic.
- Exposing internal functionality is very simple.
- It is also easy to get "reflected" functionality as well — use the same functionality wrapping it in quotation level shifters, so given some internal Term, create $f$ for every quoted level $i$ — $f_{i} m_{i} - f_{i}$. This is even simpler if the quote parameters have an integer level.
- Another point is that nonsensical quoted terms are harder to make.

$$(((\tau \tau) \text{unquoty}) f) = (\tau)_{1} f$$

Using:

- This is even simpler if the quote parameters have an integer level.
- Another point is that nonsensical quoted terms are harder to make.
The decision that binding variables are quoted with their surrounding term makes even more sense. Quoting them means that they have no semantic meaning of their own, but they are still binding occurrences in the quoted level.

This can be demonstrated in Scheme: `(lambda x (x x))`
Terms with Alpha-Equality

and

Moreover, we cannot talk about alpha-renaming or any related equivalence classes (higher-level abstraction): there is no alpha-renaming automatically (similar to representing sets using taking equivalence classes over alpha-renaming.

Moreover, we cannot distinguish because we cannot distinguish.

We can continue this: instead of using quoted terms we can.
The intuition here: think about an implementation with a similar auto-alpha-renaming package and see what you can and cannot do with such an interface (for example, display-forms).

Terms with Alpha-Equality

This means that different pattern variables are can only be instantiated with an that contains a free occurrence of x.

For example, the rewrite rule \[ \frac{f \to x f \cdot xx}{f} \] cannot be.”

This can be added to a “smart-substitution” mechanism such as the one used to specify rewrite rules in Meta-PRL.

The intuition here: think about an implementation with a similar...
Terms with Alpha-Equality

Using this frees us from making sure that no name-capture happens.
Terms with Alpha-Equality

This is, however, limited — need some primitive system for

\[ ((((((\lambda x (\lambda y (\lambda z (\lambda t (\lambda w (\lambda v (\lambda u \text{swap}) (\lambda x y z t w v u))))))) (\text{swap})))))))) \]

An attempt at a better solution: the hygiene macro system:

\[ (((((\lambda \text{swap} (\lambda x (\lambda y (\lambda z (\lambda t (\lambda w (\lambda v (\lambda u \text{swap}) (\lambda x y z t w v u))))))) (\text{swap})))))))) \]

Ad-hock solution:

\[ (((((\lambda \text{swap} (\lambda x (\lambda y (\lambda z (\lambda t (\lambda w (\lambda v (\lambda u \text{swap}) (\lambda x y z t w v u))))))) (\text{swap})))))))) \]

For example:

The usual problem is unsafe macros that capture names.

A similar idea appears in Scheme macros.

some things.
Terms with Alpha-Equality

- This idea can be carried over to a reflected Nuprl.

- We could say things like:

\[ \forall t \in \text{Term}_1. \text{SomeClaimAbout}(\forall x \in \mathbb{N}. P(t)) \]

when \( t \) does not have a free occurrence of \( x \),

- And when it does:

\[ \forall t \in \text{Term}_1. \text{SomeClaimAbout}(\forall x \in \mathbb{N}. P(t(x))) \]

except that \( \text{Term}_1 \) should be replaced with one-argument second-order pattern.

- This makes it even more obvious that \( x \) can still be regarded as a binding variable.
It makes a lot of sense for such ideas to come from an implementation.

Meta-PRL’s rule specifications are actually a meta-language that makes writing invalid rules more difficult, similar to the Scheme hygiene macros: impossible to write a macro that captures a name. Using this not only makes writing correct rules easier — it also specifies Meta-PRL.

There will probably not be any need for a lower-level term representation that will not use alpha-equivalence (hopefully).

Terms with Alpha-Equality
Final Thoughts...

Some more things that will take too much time:

Object (example: quoting Scheme procedures).

Impossible to know the intensional structure of an extensional.

A "quotify" operator must be a built-in operation since it is.

"quotify" using a \texttt{rep/unrep} wrapper.

About \texttt{N, E, eval, etc}.

Once we have such quoted objects any functionality can be.

Opaque.

Library hold quoted objects. (Not practical) Since it makes it too.

Page 245. An equivalent of this would be to make the Nuprl.

For example, logic books refer to objects like "Lemma 2.5 on.

In natural language, we can never use an object, just talk about.

Some more things that will take too much time:
Future work: definitely.

Conclusions: many.

Related work: ?