March 6, 2000

Eliezer Barzilay
eli@cs.cornell.edu

Reflections on Nuprl

[Part I]
Goals

Reflection and Nuprl

Implementation — in Nuprl for our case.

The best way to fully understand this is to have an implementation.

Yet still we are missing it in proof systems.

Reflection has proven itself valuable in programming languages.

Programming languages are known — many demonstrated.

Relationships between the concepts of proof systems and

Introduction: Language

- *Language*, in the way that we use this term, is some formal way of communication.

- Two basic elements of a language are its *syntax* and its *semantics*. 
Introduction: Reflection

Reflection in a language means that we can have syntactic constructs that have syntax as its semantics.

– Should have some syntax for these constructs.
   – Represented values domain.
   – Make the set of syntax constructs a subset of the way of representing syntax: quotation.

So, to have reflection, the first thing we need is to have some constructs that have syntax as its semantics.

Eli Barzilay Reflection & Nuprl
In natural language, we use quotes explicitly:

- "half-full" vs. "half-empty"
- "Eli's wife" vs. "Regina"

We can even use some form of evaluation:

- She said "be quiet" so please lower your voice.

And an operation to unquote text:

- The title of this slide.

But we can also mention normal text without quotes:

- "This sentence is longer than this sentence."

In natural language, we use quote symbols explicitly.
An example of how this can get complicated:

Liu said that he offered the pillowcase to Columbia officials.

"I don't want to ruin the pillow," they said.

"But you ruined my life," he said.

They refused to do this. They said, "I don't want to ruin the pillow case with the levels on the case on the pillow," but they refused to test the levels with the case on the pillow.
Reflection Rule

Being able to quote syntax is half the story; we need a rule that

relates quoted text to our language:

Programming lang: parse quoted text the same way as normal text;

Natural lang: parse quoted text is that same;

Logical system: a reflection inference rule takes quoted

inferences and conclude that they are true.

Without such a rule, we have no way to know that the quoted text is

in the same language we use.

Reflection Rule
In the language (Scheme lists, Caml4 ASTs).

Solution: use some recursive data constructs that are available.

The recursive nature of the syntax:
- It uses text strings as quoted code — they fail to represent the recursive nature of the syntax.
- Relies on external features (compiler, OS...).
- Relies on external features (compiler, OS...).
- Relies on external features (compiler, OS...).

This is almost always possible — write a source code file, run the compiler, execute the result — but this is bad:

Reflection: programs that write and evaluate other programs.

We get more precise: have formal semantics.

Programming Languages: Quotation
Programming Languages: Reflection

Then the next step would be an evaluator for quoted syntax.

Solution: Expose the implementation's evaluation to its users.

... programmers wishful thinking...

The only thing that guarantees that this is reflection is the programmer’s wishful thinking.

- Re-invent the wheel (same one you’re riding),
- Inefficient,

This could be implemented in the language itself, but:

- Inefficient.
An alternative is to use some fixed point — this would be an equivalent solution, except more complicated to implement.

The difference is similar to recursion using the Y combinator.
Implementing reflection is obvious in programming languages, but not in logic (e.g., Aitken's work). We wish to 'borrow' ideas from the PL world into the logical one.

Wishing to 'borrow' ideas from the PL world into the logical one.

implementing reflection is obvious in programming languages.

Scheme ↔ Nuprl

Programming Languages: Scheme
Number is its own syntax.

Some confusion might arise from this design, for example, a number is its own syntax.

- Generally, reflection is an integral part of Scheme.

- The result: Representing Scheme syntax in Scheme is trivial.

- Communicating with the user, so users never see anything else.

- Use a reader and a writer functions that are responsible for the same values are used by Scheme programs.

- Extremely simple syntax: made of atomic values and lists.

Scheme Syntax
Scheme, like most dialects of Lisp, employs a fully parenth­ized prefix notation for programs and (other) data. From the Scheme Revised Report:

Scheme syntax

Scheme programs and data to uniform treatment by other Scheme programs generates a sublanguage of the language used for data. An important consequence of this simplicity, uniform representation is the susceptibility of Scheme programs to uniform treatment by other Scheme programs. For example, the \texttt{eval} procedure evaluates a Scheme program expressed as data.
Scheme isn't concerned with denotation, just evaluation.

- \( \text{eval} \) is a function that takes some input source code and produces the results that this code evaluates to, if any.
- Possible because it evaluates, not normalizes (like Nuprl).
- There is no confusion: \( \text{eval} \)'s input is always code, and its output is always Scheme values.

\[ 3 \leftarrow \text{eval}(+12) \]
\[ 3 \leftarrow \text{eval}((+12)) \]

So, it's a function from Scheme values to Scheme values:

\[ \text{eval} : \text{Scheme values} \rightarrow \text{Scheme values} \]

The Scheme standard specifies an \( \text{eval} \) function.

\[ \text{eval} \]
Scheme: Quotations

Historically: S-exprs (data) and M-exprs (language), a Lisp evaluator in Lisp led to using only S-exprs.

Extension: make quote stop the evaluation of any expression.

Example: quote (+ 1 2) with (+ 1 2).

So the above is quoted by ' (+ 1 2).

- x evaluates to the symbol x.
- x evaluates to the value of the variable x.

Extension: make quote stop the evaluation of any expression.

Example: quote (+ 1 2) with (+ 1 2).

So, so:

We need a way to quote symbols: introduce a quote special form, so:

- To quote an expression, use <quote>.
- To quote an atomic value, use itself!

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:

Quote a piece of code is trivial: write an expression that evaluates to the code:
Reflection in Scheme is simple due to two important facts:

- Syntactic structures are part of the Scheme value domain.
- The `eval` function is available to user programs.

Reflection is required by the Revised Report — all that it requires is that it will implement Scheme correctly.

Actually, the fact that Scheme code uses the same data objects that are used to represent syntax is another broken abstraction barrier (and this is required by the standard).

- Exposing the internal evaluation to the user-level: breaking the abstraction barrier.
- Using a syntactic fixed point, or.
- As said, implementing an evaluation function can be done by:
  - The `eval` function is available to user programs.
  - Syntactic structures are part of the Scheme value domain.

This exposure is not forced by the Revised Report — all that it common.

The second way makes more sense, so it is (much) more.

The difference is the same as the one in the fact example.
This is called "procedural reflection", and is useful in the context of many other substrate systems besides a programming language.

All this is called "procedural reflection", and is useful in the context of many other substrate systems besides a programming language.

object systems (CLOS: a Meta Object Protocol),

operating systems (kernel modules) and file systems,

databases (trigger functions & meta data),

... and logical systems.

... and logical systems.

All this is called "procedural reflection", and is useful in the context of many other substrate systems besides a programming language.
Nuprl: Syntax and Values

Nuprl has a simple & uniform syntactic objects — terms.

Computation is defined on terms, nothing else

Uses display forms to hide the above full form.

This design has a lot in common with Scheme S-expressions.

Uses tables to define what terms are canonical (value-terms).

Computation is defined on terms, nothing else

The structure editor makes this obvious.

This design has a lot in common with Scheme S-expressions.

General form:

\[ (f\cdot x) \{ A : d \} \]

Tree structure and attached atomic values (parameters).

The information they hold comes from operator names, their

Used for input/output and processing.

Nuprl: Syntax and Values
Evaluation

So we have to find some way of representing terms using other terms.

2. There is no way it can stop with (+ 1 2).

There is no semantic distinction between two terms if both can be reduced to the same term — evaluation is idempotent.

There is no way (for now) to represent terms using terms.

This is the reason we cannot implement Scheme-style reflection: there is no way it can stop with (+ 1 2).

Eli Barzilay Reflection & Nuprl
So, how can we represent terms using other terms?

Nuprl: Quotations

- A solution to the substitution problem — keep the quoted term.
  - Makes substitution context-sensitive.
  - Reduces in them.
  - Modify the evaluator to treat these terms as data and disallow
    reductions in them.
  - Create a new quoted term that has a single sub-term, and
    modify the evaluator to a Scheme-like evaluator.
  - That will represent it (needs run-time support).

- For every possible term, have a corresponding new value term.
  - Change the term structure to have an additional "flag" in the
    term specifying whether a term is quoted or not.
  - The simple approach: Represent terms using a recursive type
    definition that will be an opid and a pair of lists (params, bound
    subterms).

Eli Barzilay Reflection & Nuprl

19/21
Nuprl: Quotations So Far, Aitken’s Approach

The approach used so far — explicitly do all the meta-match in

Nuprl with a recursive type.

Problems:
- No relation to the Nuprl implementation,
- (took Aitken about 500 pages and still not done),
- Basically, everything said before.
- Main reason for not having reflection so far.

Quotation example:

- Consider that pairs and lists are terms, and the quotation of
  \[ \langle \langle 3, \' t \rangle, \langle \text{'x'}, t \rangle \rangle \] → \text{'foo'} \text{'x'}, t \}

that becomes messy.

\[ \langle \langle 3, \' t \rangle, \langle \text{'x'}, t \rangle \rangle \] → \text{'foo'} \text{'x'}, t \}
Create a new kind of quote parameter, with a value that indicates the 'quoted-ness' level of the term.

- $\text{foo}\{1:q,3:l\}(x.t) \\
- $\text{foo}\{2:q,3:l\}(x.t) \\
- $\text{foo}\{3:q\}(x.t) \\
- $\text{foo}\{4\}(x.t) \\

This gives us terms like:

- $\text{foo}\{1:q,3:l\}(x.t) \\
- $\text{foo}\{2:q,3:l\}(x.t) \\
- $\text{foo}\{3:q\}(x.t) \\
- $\text{foo}\{4\}(x.t) \\

This is close to the second one:

- $\text{foo}\{1:q,3:l\}(x.t) \\
- $\text{foo}\{2:q,3:l\}(x.t) \\
- $\text{foo}\{3:q\}(x.t) \\
- $\text{foo}\{4\}(x.t) \\

Suggested Quotations

Eli Barzilay Reflection & Nuprl

Nuprl: Suggested Quotations