Quotation and Reflection in Nuprl and Scheme

Eli Barzilay
(eli@cs.cornell.edu)

Contents

1 Goals & Outline

Relationships between the concepts of proof systems and programming languages are known. Some are well demonstrated in systems like Coq and Nuprl, but other aspects have not been fully implemented, such as reflection. I believe that the true context in which such ideas are becoming useful is when they are implemented, this "implementation as understanding" principle is the reason some parts of the following text contain code pieces. This should take the form of a logical environment with reflection mechanisms, Nuprl is a good choice since it is used for connecting logic and programming languages. Therefore, the first step towards creating such an implementation should be taken: pinpointing what should be done, and how. This paper is an attempt to do this.

An implementation of such a reflective system merges two quite different environments, each with its own syntax, semantics, evaluation and quotations. The discussion, therefore, starts with these attributes in general languages in Section 2; programming languages are inspected generally in Section 3 and using a pure version of Scheme as a point of view due to the simplicity of its reflection mechanisms in Section 4; Nuprl is then discussed as our goal logical environment in Section 5; possible ways of implementing a quotation mechanism are discussed in Section 6. Finally, conclusions are presented in Section 7, this is the most important part.

2 Introduction

The term "language" as we use it, is a formal way of communicating concepts (objects in some domain). The language itself can come in several different ways such as vocal sounds, written text, or text encoded in computer files. Whatever form a language takes, there are rules to specify what constructs are valid — *syntax*, and how to associate syntactic constructs with the concepts they represent — *semantics* (or meaning). For example the syntactic construct of the Hebrew sound "shalosh", of the English letter sequence t-h-r-e-e, of the ASCII character "3" in some conventional programming language, and of the Nuprl term '`natnum{3:n}()`', all have the semantics of the number three. The semantic rules match syntactic structures in the language to objects in some domain that this language denotes.

Note that the place where syntax ends and semantics begins is not fixed

— we decide what syntax is valid, and then how to get its semantics, so filtering out some constructs can be done by declaring them as syntactically incorrect *or* by making their semantics void. For example, we can say that the expression '`1+"a"`' is syntactically incorrect, or that it is syntactically correct but raises an error when evaluated or compiled, making it meaningless. This is clear in a programming language implementation (deciding what component is responsible for detecting such errors — the evaluator or the parser), but it is also a question in natural languages (one option is that "books sky snail" is syntactically incorrect because it contains a sequence of three nouns, but another is that it is correct because all three words are spelled right).

The term "reflection" describes three properties of a language, the first two are the fact that it allows syntax that denotes (by its semantics) its own syntactic constructs, and that it can talk about such constructs. In written natural language, the double quote symbol is used to specify that a piece of text is not to be taken as representing concepts in the normal way but instead, as representing the actual text itself. For example, the English word "water" stands for water, but the text "the English word 'water'" uses the actual word "water" as a piece of syntax (and this sentence just mentioned a piece of text that contained quotes). It is obvious that quotations are a fundamental aspect of reflection.

So the first thing to have in the domain represented by a language that can reflect itself is objects that stand for syntax object of the language itself, in other words, make the set of syntax constructs a subset of the represented values domain. Then, we must have some syntax that specifies such quotations (the double quotes in the natural language case). When we have such a piece of syntax S_1 that denotes a data structure that represents a piece of syntax S_0, we say that S_1 is the *quotation* of S_0.

Many representations can be used to specify quotations. One obvious representation is taken from the informal usage of quotes and raw text in natural language, however, this is an extremely poor representation for programming languages and logical systems since it does not reflect the inherently recursive nature of syntactical constructs[1]. A representation that is natural in the context of formal languages is using the language capabilities for defined types

[1]Natural language syntax is structured as well, but this structuring can be ambiguous sometimes which means that a recursive tree structure can be insufficient. This is not a central issue since when such text is read, we automatically perceive its 'parsed' form.

(records in the case of programming languages, and tuples/sequences/defined types in the case of logical systems). Writinf such structures explicitly is also insufficient since it makes quotation cumbersome and inefficient, specifically, repeatedly quoting some syntactic object makes the result grow exponentially. Other mechanisms like a quotation context, operator shifting[2] and a general preprocessing mechanism can all help solve this problem, these are discussed in Section 6.

Most languages have an inherent evaluation process: we first get the syntax, then see what it denotes (if it makes sense) using the semantics of our language, and then we evaluate the result. This is a mental process that starts with a sentence as a piece of syntax, converts it to a piece of semantic information, and then forms a final mental piece of information in our mind using some form of evaluation.

Evaluation can take several forms, for example — we can identify and expand definitions such as "Eli's wife" or pronouns like "you" and identify them with other concepts such as "Regina Barzilay". We can also use some logical rules that are part of our language like eliminating double negations. More rules that we use to build such a 'mental image' can come from the process in which this image is built, for example, adjectives specify properties of objects, so they are order-independent (*e.g.*, "the big blue car" and "the blue big car"). Finally, some information is taken from rules of the physical world: we know that "mixing flour and eggs" is the same as "mixing eggs and flour", or that "a half-full glass" is the same as "a half-empty glass" — this, of course, can depend on the context in which it is used.

There are also rules that handle quotations: this is interesting since it is the way natural language implements self-reference. Quotations can be used as any other object, and they actually describe their contents: so the first thing that makes this similar to the world of programming is that evaluation does not occur inside quotes. As an example, the previous paragraphs mentioned several pieces of text that would evaluate to the same mental image *if* they were unquoted. More rules involve referencing pieces of text, as in "The third word of this sentence", or direct evaluation using terms like 'meaning' as in: "The word 'word' stands for the concept of a word."

This leads us to the third property of a reflected language: when we have the above two, then it is possible to talk about the language within

[2]This is the term we use for using an operator name to create a constructor that generates quoted code that mentions the original operator.

itself, but there is no real guarantee that the quoted language is *identical* the language itself. Therefore, the third property is the correspondence between this representation and the language. This can be regarded as the guard-dog that makes sure represented objects behave as we *expect* them to behave. The form of this correspondence depends on the nature of the language:

- in a natural language we want quoted text to be related to the actual meaning of that text;
- in a programming language we want evaluation of quoted source code to behave the same as the same unquoted source;
- and in a logical system, we want a reflection inference rule that can take a piece of quoted inference and concludes that the same fact is true (in other words, provability of some represented term implies that the term itself is true).

3 Reflection in Programming Languages

When talking about programming languages, we must be more precise. A programming language has some formal rules for constructing its syntax, and a function that evaluates such input, producing some result. An operational semantics is defined by an evaluation process that turns syntax into values. The evaluator can come in several forms such as an interpreter, or the composition of a compiler and machine execution. To reflect here means to be able to write a program that can itself generate pieces of code and execute them. Of course, this is almost always possible, since even a primitive language like assembly code can be used to write a text file containing some other text, use the operating system to invoke the assembler over this file and execute the result. This is, however, an extremely crude way of implementing and using reflection[3] because:

1. It relies on features that are external to the language itself (the OS and accessibility of a compiler in this case) which is inefficient, and might not be available at run-time.

[3] I have heard of some production engineers in Intel that used a similar technique to implement functions that get a variable number of arguments using batch files that wrote Basic programs.

2. Text file generation, or text strings in general, is a low-level representation that is difficult to manage and understand, mainly because they fail to represent the recursive nature of the syntax [3].

What is much preferred for this purpose are data structures within the language that can represent syntax, and, of course, some mechanism to specify quotations. *Quotation* of a piece of syntax S_0 in this context means: finding a piece of syntax S_1 that *evaluates* to an object which is a representation of S_0.

As mentioned earlier, the obvious way for representing syntax is to define recursive data structures (assuming the language has some way to define such structures). This can vary from complex representations like the Abstract Syntax Tree entries used by CamlP4 [4] to the simple lists of Scheme [6]. One thing to note is that these data structures define the line between syntax and representation — everything that can be parsed to such structures is considered valid syntax.

The next step on the way to reflection is to have a user-available evaluator in some reasonable way. One way of achieving this is to implement one — this has the advantage of requiring only user data structures and Turing-completeness. An obvious reason for rejecting this is that it is basically re-inventing the wheel that you already use, but an even stronger reason is that this is not true reflection in the sense that the implemented evaluator has no relation to the language used except for the programmer's wishful thinking. The consequent of this is that "true" reflection should be enabled using the actual evaluation function which executes the program itself — this is by means of exposing it to the language. This guarantees the third property mentioned above.

This can be achieved using a fixed-point principle: implement an evaluator that can evaluate itself and use the result. The question here is whether this result is equivalent to the original evaluator[4]. A safer way for this is exposing the language's built-in evaluator to the language itself; by this, the interface barrier between the language and its interpreter is broken, but this is the essence of reflection: using a language to talk about itself. Exposing internal parts of a substrate system such as a programming language is a general idea that can be used to achieve greater flexibility as demonstrated

[4]Some compilers such as OCaml are built by bootstrapping — making the compiler compile itself and iterate using the result, until the compiled result is identical to the compiler itself

in “The Art of the Metaobject Protocol” [7].

4 (Pure) Scheme

In the following two sections, the features of Scheme and Nuprl are compared. Nuprl is taken as a representative logical environment that contains an evaluator for a simple untyped term language. Scheme was chosen as a programming language representative due to its simple design compared to other languages, especially when it comes to its reflection capabilities. The discussion is limited to a pure subset of Scheme, side effects and other irrelevant concepts are ignored. The syntactic issues are the same as in standard Scheme.

4.1 Syntax

Scheme’s syntax is essentially the same as that of other languages in the Lisp family. It is extremely simple — everything is either an atom of some fundamental type (*e.g.*, numbers, symbols and strings), or a list of objects represented by some parenthesized whitespace-delimited sequence of objects. This is actually the syntax for general Scheme objects; the syntax for the language is a subset of these expressions (for example, lists represent applications, symbols represent variables, a list beginning with the symbol ‘`lambda`’ represent functions etc.). This is the first of several features that make reflection an integral part of the language. Quoting the Scheme Revised^5 Report [6, p. 3]:

> Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. For example, the ‘`eval`’ procedure evaluates a Scheme program expressed as data.

Scheme implementations have a *reader* function (‘`read`’) that parses input, and a *printer* function (‘`write`’) to display values. The philosophy behind this is that printed output always represents values equal (modulo pointer equality) to the result of feeding this output back to the reader[5].

4.2 Semantics (Values)

Values in Scheme are of two major kinds:

- atomic values such as symbols, numbers and strings[6],
- composite values — lists holding an ordered sequence of values[7].

Lists are implemented using the '`cons`' function that constructs a pair in memory (a *cons cell*) and the empty list ('`'()`'). Proper lists are either the empty list or a pair of any value (the head of the list) and a proper list (its tail). Also, the '`list`' function is a convenient shortcut for creating lists: '`(cons` x `(cons` y `'()))` = `(list` x y`)`'.

The way atomic values are represented in Scheme syntax structures raises a subtle point: the Scheme interpreter sees all input through the glasses of its reader function — so when Scheme source code contains, for example, a number, the reader will parse this and create the internal representation of that number, which becomes part of the [parsed] input source; therefore, the syntax for a number is itself. Other values, including lists are also represented by themselves using the same mechanism.

It was said previously that to enable reflection we must extend the domain of the language so it holds *syntactic structure* objects — in Scheme this is done by simply making objects be the syntax that *represent themselves*, so the domain of Scheme objects is a superset of the domain of Scheme syntax structures. This point is unique to Scheme (and other Lisp dialects) due to the combination of an interpreted environment with the way syntax is represented as values. More on this below.

---

[5] This is not possible with all objects, for example, functions usually cannot be printed. Also note that feeding such output back to the interpreter will get it re-evaluated unless quoted.

[6] Strings do have some internal structure, but here we make a distinction between atomic and composite objects in the specification of Scheme programs so this is irrelevant.

[7] Another thing that is ignored for this discussion is usage of "dotted-lists".

4.3 Evaluation

A Scheme interpreter is basically a `read-eval-print` loop ("REPL"). The '`read`' and '`print`' parts are responsible for user interaction (the mapping between internal objects and their textual representation) and '`eval`' is the actual Scheme evaluator function. Obviously, the evaluator is responsible for the actual behavior of Scheme programs.

'`eval`' is a [partial] function that takes some input source code (an internal representation built by '`read`') and produces the results that this code evaluates to, if any. It is an applicative-order evaluator that uses lexical environments.

The fact that '`eval`' is just a function from Scheme values to Scheme values might sound confusing at first: how can it distinguish values that represent code from other values? The solution is simple — *the input is always taken as a piece of code representation and the output is always a piece of data*. For example, if the code '`(list '+ 1 2)`' is evaluated, the return value is a list holding the symbol '`+`', and the numbers '`1`' and '`2`', and this is not evaluated further. In fact, if the '`eval`' function was not available to the user, then there was no way that it would ever get any input syntax other than user code — that is, it would never get applied on expressions that are results.

So when a Scheme interpreter is used, entering a result string back can result in an equal object in the case of a non-symbol atomic value, but a different value in case of a symbol or a list, in other words, Scheme's evaluation is not idempotent. For this reason, the DrScheme pedagogic environment [5] helps beginner-level students getting used to the language by "language level" settings, where the printer is modified so values are printed in a way that will evaluate to an equal object when re-entered — for example, the result of evaluating '`(list 1 2)`' is the list holding one and two which is printed as '`(list 1 2)`' or as '`'(1 2)`' by DrScheme. There are other implementations that choose to display values in a similar way.

What eval is doing on a given argument can be summarized as follows:

1. If the argument is a symbol, its binding in the current lexical environment is returned;

2. If it is any other atomic value, then this value is returned;

3. If it is a list and its first element is a special-form then the corresponding special evaluation rule is used;

4. If it is a list and its first element is a macro symbol, then the macro is applied to the unevaluated arguments (source code pieces) and the result is evaluated further;

5. Otherwise, it is a list and its elements are evaluated (in some unspecified order), and the first is applied on the rest.

Note that rule 2 is possible because values are actually part of the input source representation.

4.4 Quotations (Representations)

The way Scheme syntax is defined leads to the fact that Scheme source code is represented by data objects that are part of the language and accessible to user-programs; the syntax of Scheme programs is that of these objects. The same holds for other Lisp dialects. This, however, was not always the case: the Lisp 1.5 Programmers Manual [9] specifies two ways of expressing programs:

S-expressions These are *symbolic* expressions that are used for representing arbitrary data — this includes Lisp source represented in internal form.

M-expressions The actual source language that a Lisp programmer uses is named the "meta-language", since it specifies how S-expressions are processed. M-expressions can be represented in the form of S-expression for Lisp programs that use other Lisp programs as data.

The distinction between the two was supposed to be clear: programs in the form of M-expressions are what the compiler works with, while S-expressions are used for internal data — sometimes representing Lisp syntax. However, an evaluator function was written, leading to a Lisp interpreter for Lisp programs that are written in S-expression form. This led to the representation of Lisp code using S-expressions being the dominant programming language [8].

It might be possible to use a more 'standard' syntax in a smarter way than the one intended to be used in Lisp 1.5 — modify the reader and the writer functions so *both* use the same syntax — essentially modifying the way lists are represented as text. However, this would require extra information such as what symbols are infix operators, their precedences etc. This is further complicated by the fact that we might want to print some object as denoting data rather than code (for example, '`(foo '(a b c) '(if of uf))`'). It is therefore sensible to stick to the simple syntax.

As said above, Scheme values represent themselves, and composite pieces of syntax are represented by lists, so quotation becomes trivial: to quote a piece of input source you simply write an expression that will have it as its result. The only missing piece to complete this picture is the way symbols can be a result of a Scheme expression — the evaluator treats symbols as variable references, so a new special form named '`quote`' is added to the language, which stops evaluation of a symbol: the result of evaluating '`(quote a)`' is the symbol '`a`'.

Now we know that:

- to quote a symbol, we wrap it by a '`quote`' special form;
- to quote any other atomic value, we simply use it (as discussed above);
- to quote a composite syntax (a list) — use the '`list`' function to create the list.

For example, the quotation of '`(+ 1 2)`' is '`(list (quote +) 1 2)`'.

Finally, quoting is made easier by extending it so it stops evaluation of any Scheme syntax object including lists. For example, using the single quote character makes the quotation of the previous example as simple as '`'(+ 1 2)`'. This is generally discussed in Section 6.1 and the Scheme case is detailed in Section 6.3.

4.5 Reflection

Lisp was designed for easy symbolic computation, part of which was evaluating Lisp code using Lisp code. Scheme, as a modern and elegant dialect of Lisp, was designed to be as clean and as simple as possible. The lesson learned from Lisp led to making reflection usage as simple as[8]:

```
(eval '(+ 1 2))
```

This simplicity comes from two main factors:

[8] Actually, this is not the way '`eval`' is defined in the Scheme Report. In fact, '`eval`' was introduced only in the 5th Revised report in 1998.

- The syntactic structures are part of the domain of Scheme values: non-symbol atoms represent themselves, quotes can be used to get symbols as values and lists can be constructed by users.

- The evaluation function (‘`eval`’) is available to user programs.

There are few other supporting mechanisms such as quasi-quotes, macros etc that are discussed below.

As said in Section 3, we can get the evaluation function for our own language either by using a fixpoint or by exposing the actual evaluation function to the user level — breaking the abstraction barrier between the language and its implementation. The second way is what Scheme implementations usually do: the ‘`eval`’ function is the same as the one that the implementation uses for evaluating code. This it is simpler, safer and more efficient. It should be noted that such implementation is not required by R^5RS, the only thing that is required is the availability of an ‘`eval`’ function that evaluates Scheme expressions.

The fact that syntax structures are part of the domain of Scheme values is another such exposed internal mechanism: the Scheme implementation and user programs share *the same* data structures. This is required by the standard so Scheme code can always be used as data, for example, as input to ‘`eval`’.

This way of reflecting a system by exposing some of its internal functionality to its users is called *procedural reflection*, see Smith [10] for more details.

5 Nuprl

Nuprl is a logical environment implementation that connects constructive logic and programming. It is a candidate system for an implementation of reflection so the connection between the logical meaning and the programming meaning of reflection can be made explicit.

5.1 Syntax

Terms are the fundamental objects Nuprl manages: they are used for input, output and internal processing. The information that terms represent comes from their operator name, their tree structure, and from attached atomic values (parameters). The structure of a typical Nuprl term is a tree structure of terms with no parameters and terms with parameters and no sub-terms as leaves.

In short, terms and parameters provide a simple and uniform syntax for Nuprl, much like lists and atomic values in Scheme. There are, however, some differences as we will see.

The actual user-interaction uses a structure editor for entering terms and a display form mechanism for visualizing terms. This is a rather technical point that makes life a little easier for users[9]: the internal representation is the same *no matter how it is presented.* The analogy for this in Scheme would be a modification to the reader and printer functions as discussed in Section 4.4.

5.2 Semantics (Values)

In Nuprl there are no "atomic values" as in Scheme — there are only terms — attached parameters provide the actual content of term values. Since there are no atomic values, there is an additional mechanism to specify what terms stand for [canonical] values (named *value terms*) and what terms should be evaluated further to get a value. For example, numbers are represented in Nuprl by '`natnum`' terms with no subterms and with a parameter that specifies the actual value, for example: '`natnum{3}()`'. Value terms have no specified meaning — they stand for themselves. This is due to the fact that evaluation in Nuprl has the form of normalizing terms, substitutions can occur in any order.

5.3 Evaluation

Terms are used in Nuprl as the elementary data objects, representing logical sentences. In addition, the system contains an evaluator component that uses terms as an untyped lambda-calculus like language. Terms that are not declared to be values, get reduced by the evaluator. These terms have *evaluation fragments* which are small functions that define reductions that the evaluator use to handle them. This evaluator is different than standard programming language evaluators, it is a normalizing process: a term can be reduced until it is a value term.

[9]This is especially necessary since many Nuprl users are mathematicians.

Another difference between evaluation in Scheme and in Nuprl is that because a term and its normalized form are always equal, then the evaluation can be lazy where Scheme is eager. Moreover, there is another important point about the way Nuprl treats terms that should be made clear at this point: there is no distinction between two terms if one can be reduced to the other (or both to a the same). The evaluator gets terms as input, and reduces subterms lazily as necessary — this makes it a function that maps terms to terms. This is similar to Scheme, but the fundamental difference is that terms that can be reduced to the same (alpha-equal) term are considered indistinguishable, in other words, this eval function is idempotent: there is no difference between eval(x) and eval(eval(x)). If evaluation in Scheme was defined similarly, then evaluating the expression '`(car (list (list '+ 1 2) 1))`' would yield '`3`' because '`list`' creates an actual expression. The 3Lisp language [10] faces the same problem and the solution was to make it use normalization as well.

Nuprl's approach allows a lot of freedom in the sense that different evaluation techniques can be intermixed, it even allows a more complex system that specifies what parts should be reduced. In other words — the Scheme evaluator knows that anything that the evaluator gets is a syntax value and anything it returns is a simple value, while the Nuprl evaluator always returns a term and these terms are classified to data values and reducible terms: this allows it to do an incomplete job, deferring unnecessary reductions.

5.4 Quotations (Representations)

The restriction implied by the evaluation process implies that exposing the term constructor in Nuprl so terms represent themselves is impossible: as described above, Nuprl can reduce arbitrary subparts of some expression, which means that terms cannot represent themselves since there is no way to specify that they should be treated as values. There are several possible ways for making syntax representation possible:

1. Represent terms using a simple recursive type definition that will be composed of a pair of lists, one representing parameters and the other representing bound subterms. This is the simple/naive approach.

2. Change the term structure so that there will be an additional 'flag' parameter in terms, specifying whether a term is quoted or not, it

should be possible to accumulate these flags, which will denote "quoteness" levels. These flags only indicate the term as quoted, not its subterms: we shift the meaning of the operator with this flag from what it denotes to a representation of its own syntax; we name this *operator shifting*. This is an implementation change.

3. For every possible term, make a corresponding new value term that will represent it, this should be done carefully so all terms are representable, including these terms as well.

4. Modify the evaluator so it is more similar to the Scheme evaluator. This will, naturally, reduce its flexibility. Note that this is a suggestion on how to design the way evaluation will happen, it can still be a lazy evaluator.

5. Create a new term named '`quote`' that has a single sub-term, and modify the evaluator to treat these terms as data and disallow reductions in them. This violates the principle of uniform management of terms by making substitution context-sensitive (free variables in quoted terms should not be replaced). This has a drastic effect on the system since it changes the way equality behaves.

6. Do the same, but have no subterms, instead keep the quoted term as a parameter value. This means that no implementation change is needed. However, the structure of the quoted term is not easily accessible, specifically, we cannot have subterms that describe parts of the quoted term.

The standard approach in Nuprl, which is the one taken in [2] and in [1], is the first one above. A suggested improvement is the second one which is temporarily hacked similar to the third one: for every term we declare a matching value term that is used to represent the syntax of that term, But eventually we want this to be done automatically by such quote flags and selector functions that will be able to pull information out of quoted terms (see Section 6.4). There is a '`rep`' function from terms to terms, that produces the canonical representation of a term; the Scheme equivalent of this function, when the simple approach is used, would roughly look like this:

```
(define (rep expr)
  (if (list? expr)
```

```
      (cons 'list (map reps expr))
      (list 'quote expr)))
```

but this is, of course, simpler using the extension that allows quoting every object, specifying quoted contexts:

```
  (define (rep expr)
    (list 'quote expr))
```

Another function, '`unrep`' is the opposite operation, which in Scheme (given the above function definition) is simply '`eval`'. As with any other Nuprl term, terms are the same as ones that are the result of their evaluation, so, for example, the '`unrep`' function must be partial since some terms represent infinite computations. Note that the only way to speak about these functions is to lift the discussion to the quoted level: for example, it is impossible to write the above Scheme function so it will get a piece of unquoted syntax since that syntax will get evaluated (but it is possible to write it as a macro).

5.5 Reflection

The above quotation mechanism allows reflection to be implemented. This however was never done in practice: a theoretical discussion appeared in [2], and an attempt to get a practical implementation was done in [1]. This paper is an attempt to be the first step on the road to a practical working reflection implemented in Nuprl of a partial subset of the system exposed internals, unlike the ambitious attempt of [1] for the reflection of the full Nuprl in terms of itself. The plan is to use operator shifting to expose a representation of term structures, together with internal functionality to manipulate these representation.

6 Quotations

So far, we have examined Scheme's reflection capabilities and compared them to Nuprl's. The problem with what was done in Nuprl is that it never reached a stage of practical usage. To get to such a working system in a new way, some variation is needed: here we inspect possibilities for quotation mechanisms, from a general point of view and possible implementations.

6.1 Quotation by Preprocessing

Now that we know how to use quotations, we encounter another problem: using quoted data structures in the obvious way is very cumbersome. This is true even for the simple list structures in Scheme — a simple expression such as:

```
(if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))
```

is quoted (using the quote character shortcut) as:

```
(list 'if (list '<= 'n 1) 'n
  (list '+ (list 'fib (list '- 'n 1)))
  (list '+ (list 'fib (list '- 'n 2))))
```

As said in Section 2, some quote notations are much simpler to use, for example, if we would have chosen to represent Scheme code as strings, then the quoted expression becomes as simple as:

```
"(if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))"
```

but, as already mentioned, strings are a very poor tool for syntax representation for lack of recursive structure. The reason that makes strings a poor representation is exactly what makes 'good' representations cumbersome: we *want* the recursive nature of the syntax to be represented in our data structure, so we always get these constructors stuck in the middle of the represented text such as the extra '`quote`'s and '`list`'s in the example above. A property of a quotation mechanism that is easy to use is that original syntax appears literally in the expression that is its quotation, such as the string example (as discussed in Smullyan [11, Chapter 1]).

The way Scheme solves this problem is by introducing the '`quote`' special form. This allows us to write:

```
(quote (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
```

or even simpler:

```
'(if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))
```

When considering this, it looks like a neat solution that requires a minor addition to the language of the '`quote`' special form. However, we can observe that if we know how to translate a piece of syntax to another which is its representation, then it is possible to write a source code transformation function that will do this for us so we are not even aware of the actual way these objects are implemented. All we need is some form of quotation to be added to the *input syntax*, then it is easy to use such a transformation function as a preprocessor that will effectively eliminate these constructs and substitute the actual full syntax constructors. If this function is applied inside-out, we get it to handle multiple levels of quotations for free. For example, transforming this code:

```
(f '(g '(+ 1 2)))
```

starts at the internal quote, getting:

```
(f '(g (make-expr '+ '1 '2)))
```

then the second quote is expanded and the final result is:

```
(f (make-expr 'g (make-expr 'make-expr ''+ ''1 ''2)))
```

This example is using a '`make-expr`' function that constructs expressions, which avoids specifying how atomic constants are encoded. An actual code in Scheme is simplified by the way quote characters are handled by the reader — all we handle are expressions with the symbol '`quote`' in their first position. An example for such a preprocessing function is shown in Figure 1.

```
(define (preprocess expr)
  (define (quotify expr)
    (cond ((pair? expr) (cons 'list (map quotify expr)))
          (else         (list 'quote expr))))
  (cond ((not (pair? expr)) expr)
        ((eq? (car expr) 'quote)
         (quotify (preprocess (cadr expr))))
        (else (map preprocess expr))))
```

Figure 1: Preprocessing quotes in Scheme.

This code is generic: it can be used in any language as a preprocessor for these simple quotes — it only relies on basic mechanisms of the underlying language: being able to construct and deconstruct expressions that represent syntax, quoting symbols and managing lists in the Scheme case. Actually, it is very different from the way Scheme handles quotes which is described below. This code can also demonstrate the fact that writing naive expressions to generate structure that represent some syntax ends up in an exponential blowup of expression sizes, see page 22 for an example.

The CamlP4 package for OCaml is an example for this approach: it extends the input syntax with quotation marks, uses the parser on the string contents of these quotations and returns some transformation of the resulting abstract syntax tree object; this is further complicated by typed abstract syntax structures — a quotation must specify the type of syntax piece to parse.

6.2 A Quasi-Quotation Preprocessor

Quoting pieces of syntax using such a mechanism as described above is very convenient, but we can get still greater convenience. Quotes are used for specifying fixed syntax pieces, but when writing programs that manipulate syntactic structures, it is desirable to mix quoted syntax objects with 'normal' code. For example, here is a function that manipulates input values that are quoted objects themselves:

```
(define (build-plus exp1 exp2)
  (list '+ exp1 exp2))
```

Using simple quotations is certainly not enough since the body of

```
(define (build-plus exp1 exp2)
  '(+ exp1 exp2))
```

quotes the two variables instead of using their values. The solution is to use quasi-quotation: this is taken exactly as the normal quotation above, except that we allow another input construct for 'unquoting' some parts of the quasi-quoted expression. For example, in Scheme, the above code becomes:

```
(define (build-plus exp1 exp2)
  `(+ ,exp1 ,exp2))
```

where '`` `x ``' is a shortcut for '`(quasiquote x)`' and '`,x`' for '`(unquote x)`'.

The way to preprocess a quasi-quoted construct is to turn it into an expression that generates the templates — using '`quote`'s and '`list`'s in the Scheme case, but leaving unquoted values as they are. This means that '`` `(+ ,exp1 ,exp2) ``' is transformed into '`(list '+ exp1 exp2)`'. This is achieved

```
(define (preprocess expr)
  (define (quotify expr)
    (cond
     ((and (pair? expr) (eq? (car expr) 'unquote)) (cadr expr))
     ((pair? expr)      (cons 'list (map quotify expr)))
     (else              (list 'quote expr))))
  (cond ((not (pair? expr)) expr)
        ((eq? (car expr) 'quasiquote)
         (quotify (preprocess (cadr expr))))
        (else (map preprocess expr))))
```

Figure 2: Preprocessing quasi quotes in Scheme.

by the simple code in Figure 2, which is the same as the code in Figure 1 with a different treatment for unquoted forms.

This is also different than the way Scheme handles '`quasiquote`'s, but using it gives the same convenience. As is the case with the code from Figure 1, this code is generic in that it requires minimal support from the underlying language, yet it provides the full convenience of using quasi-quotations as template specifications. The surprising fact here is that the single addition to this code is enough to handle nested quasiquotes when it is wrapped in the recursive '`preprocess`' function. Again, an example for this is the CamlP4 quotation mechanism: it extends the way that quotation strings are parsed by allowing 'anti-quotation' constructs providing the same functionality.

The simplicity of using quasi-quotes with unquotes comes from their natural view as templates with holes to be filed. A very brief experience with these is enough to get convinced by their usefulness. The [planned] way of implementing a quotation user-interface in Nuprl with display forms and a modified input method which both use the represented terms with different colors is another form of achieving this goal: no matter how the quotation mechanism actually is implemented, it is hidden behind an abstraction interface. The input method will be very similar to the above code, using the representation of terms discussed in Section 5.4.

6.3 Quotations and Quasi-quotation in Scheme

As discussed above, quotations and quasiquotations can be handled in an "evaluator-transparent" way by using a preprocessor. However, the way that

Scheme implements these is more sophisticated.

Since the '`quote`' symbol is already treated as a special form that stops evaluation, it is natural to extend its behavior to any expression. Using it with atomic values is not useful since they evaluate to themselves anyway, but with lists it becomes very handy. One thing to note here is that since quotes are being treated as evaluation stoppers, then the quoted value appears as a literal constant in the code, so when semantics of references are being considered, it is actually different than using the equivalent '`list`' form[10].

Quasiquotes are also implemented as special forms in Scheme. This is a subtle point that might not be obvious when reading the Scheme Report. First, it appears as a derived expression (one that can be expressed using primitive syntax), second, the way it specified is, indeed, by translation to primitive syntax[11]:

> If a comma appears within the ⟨*qq-template*⟩, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression.
>
> ...
>
> The external syntax generated by write for [quasiquoted expressions] may vary between implementations.

So, it looks like all usages of quasiquotes are eliminated when it is being read in — however it is possible to quote (or quasiquote) quasiquoted expressions, as the Report states:

> Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

As demonstrated above, having quasiquotes as a special form in the language is not necessary — the way it is added to Scheme makes it more efficient. An additional note about this feature in Scheme (or any other Lisp dialect), which is clear by declaring it as derived syntax, is that quasi-quotes can be (and sometimes are) implemented as macros that preprocess the code

[10] This is the why the Scheme Report restricts such values as immutable values.

[11] This discussion ignores splicing.

so this essentially makes quasi-quotes work exactly as described above. This is also the reason why it is simpler to add quasiquotes to the language as a macro than eliminating any mention of it with a seperate preprocessor.

6.4 Suggested Quotations in Nuprl

When considering a quotation mechanism for reflection in Nuprl, the way Scheme defines them is one option. However, taking into account the way Nuprl terms are evaluated, problems are encountered. The major problem is that such '`quote`'s or '`quasiquote`'s constructs (terms, in this case), work by creating a context that changes the meaning of expressions — the way they evaluate. This is perfectly fine for Scheme evaluation since macros get expanded before code is executed, and even if treated as special forms, things are still okay, since expressions cannot be evaluated at arbitrary places but only outside-in. In Nuprl, however, reductions can occur at arbitrary places, so having these quotations means that any substitution of a term must consider its context. This is a critical point in the system which defines how equality behaves. Another point where things get unnecessarily complicated is the fact that terms that are subparts of other terms become ambiguous: we need extra information to specify whether they are quoted or not.

The desired solution should have the property of using normal terms that are not treated specially by reductions and in general, modifying as little as possible existing functionality. All this should be accomplished while the result is still convenient enough to use. One such solution is presented in `http://some-url-in-stuart's-home-page`, which is similar to option #2 on page 13. The idea is that we add quote-tags to operators, and these will be treated as if they were defined as canonical terms representing the corresponding operators. Operators with these tags are called *shifted operators*. Once the Nuprl implementation is modified, then the full representation using the recursive type definition is not needed.

This can be demonstrated by a simple Scheme program: simulate shifted operators by application of quoted symbols. The result program is in Figure 3. Using this code as a preprocessor, the result of entering[12]:

```
('+ ('* '1 '2) (* 1 2))
```

is the list `(+ (* 1 2) 2)`.

[12]Note that this uses Scheme's special treatment of atomic values, the actual multiplication should actually be quotified.

```
(define (preprocess expr)
  (cond ((and (pair? expr)
              (list? (car expr))
              (= (length (car expr)) 2)
              (eq? (caar expr) 'quote))
         (preprocess (cons 'list expr)))
        ((list? expr) (map preprocess expr))
        (else         expr)))
```

Figure 3: Preprocessing shifted operators in Scheme.

This is, of course, only a simulation that uses actual Scheme quotes for creating syntax representations, but it is a good demonstration of this idea when it is implemented for Nuprl terms: ‘`'+`’ is the shifted version of the ‘`+`’ operator. One additional fact that can be observed here is that this quotation style is also efficient: it prevents the exponential growth of expression sizes when it is quotified multiple times. For example, if we use the preprocessor of Figure 2, then quoting the simple expression `(+ 1 2)` three times yields:

```
(list 'list
      (list 'quote 'list)
      (list 'list (list 'quote 'quote) (list 'quote '+))
      (list 'list (list 'quote 'quote) (list 'quote '1))
      (list 'list (list 'quote 'quote) (list 'quote '2)))
```

whereas the new style of quotation using operator shifting yields a simple: `('''+ '''1 '''2)`. Note also that the Scheme-style version of this will be even simpler: `'''(+ 1 2)`, but this is due to the simplicity of using contexts.

As demonstrated, the context of a logical system such as Nuprl makes Scheme-style quotes too complex and the above quotation mechanism helps in that. However, contextual quotes are still useful, as can be seen by their usage in informal language. This makes a good justification for providing a Scheme-like quasiquote mechanism that will be translated by a preprocessor to actual terms, similar to the idea of color-coding quotations.

7 Conclusions

- The provability/programming relation expresses itself as proofs that talk about other proofs which translate to programs that write programs — this is close to macros and staged evaluation/compilation. Working with macros is an old subject that is very well-known and I believe that these techniques can help formalizing provability up to a point where it can be used in a computer-aided logic environment such as Nuprl. One example for a possible contribution of this might be an implementation of tactics as meta-proofs. An implementation is therefore needed to fully understand this relation.

- A small, fully-reflected implementation of a "Mini-PRL" system will be a good starting point for playing with these ideas. This should be a very small system that can do simple refinements without the major complexity of Nuprl (such as tactics, display forms, interactive editor, sophisticated rewrites etc.). There is enough to learn from such an experience, and it can then be extended.

- The simplicity of Scheme, which is achieved by exposing the evaluation function of the language to its programs makes reflection simple and 'neat' — it is very small, very simple and very elegant. I believe that similar techniques can be useful in the Nuprl case as well.

- The same also holds for the syntactic data structures: exposing the internal structure constructor (of terms in the Nuprl case) to the user level will make reflection much simpler. This can be done as discussed in Section 6.

- The benefits of having a reflection mechanism was shown as an extremely useful tool in numerous domains, not only programming languages, but other substrate systems as well — operating systems, object systems, data bases etc. Nuprl is a substrate system of yet another kind, and as such, it will probably benefit as well from a reflection mechanism. One such benefit, getting tactics as results of reflected proofs is mentioned above, but again: an implementation is necessary to fully explore the possibilities.

- Another question that needs an answer is whether it is possible to achieve reflection by proving a Mini-PRL system inside itself and have it be the extraction of this process. This will be the first logical system which can "verify itself" in some interesting sense. We know that there must be some external mechanism to make a reflective system work, it will be interesting to locate this minimal mechanism when a reflective logical system is implemented and to contrast it with the programming language world.

References

[1] W. Aitken. *Metaprogramming in Nuprl Using Reflection*. PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 1994.

[2] S. F. Allen, R. L. Constable, D. J. Howe, and W. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.

[3] A. Bawden. Quasiquotation in lisp.

[4] D. de Rauglaudre. *CamlP4*, April 2000.

[5] F. Felleisen, F. R. B., F. M., and K. S. The drscheme project: An overview. *SIGPLAN Notices: Functional Programming Column*, 1998.

[6] R. Kelsey, C. W., J. Rees, et al. Revised^5 report on the algorithmic language scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7–105, 1998.

[7] G. Kiczales, desRivieres J., and B. D. G. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[8] J. McCarthy. History of lisp. February 1979.

[9] J. McCarthy et al. *Lisp 1.5 Users Manual*. MIT Press, Cambridge, MA, 1962.

[10] B. C. Smith. Reflection and semantics in Lisp. *Principles of Programming Languages*, pages 23–35, 1984.

[11] R. M. Smullyan. *Diagonalization and Self-Reference*, volume 27 of *Oxford Logic Guides*. Oxford Sciences Publication, Oxford, 1994.