Practical Reflection in Nuprl

Eli Barzilay Stuart Allen Robert Constable

March, 2003

Abstract

We present an implementation of reflection for the Nuprl theorem prover, based on combining intuitions from programming languages and logical languages. We reflect a logical language, adapting the common practice from Lisp dialects of avoiding redundant coding whenever it is possible to expose internal functionality. In particular, concepts of the logical language such as term syntax and substitution are directly reflected as primitives. The resulting notation has both the expressiveness and the simplicity needed for ordinary syntactic explanations and arguments. The system is demonstrated by formalizing Tarski's result regarding the internal undefinability of a Truth predicate, closely following a standard "paper proof". We believe this shows to good effect our rather transparent quote-like notation, especially by exploiting colors.

1 Introduction

Reflection is known as a useful in both theory in practice, yet, a practical implementation of reflection, similar to the one used in programming languages, in a logical setting is difficult. We present our work on such an implementation, based on the simple intuition in computer science and logic that redundancy is to be avoided: if some information is encoded in some form, duplicating it is a potential sink of unnecessary human resources, and worse — a possible source of mistakes. Following this principle, the well established practice of computational reflection in Lisp dialects is both cheap and robust — instead of implementing an evaluator on top of the exiting one which imitates it, the actual underlying interpreter which is being used at the system (meta) level is made available to the user (object) level. The result is cheap as there is only one implementation, and robust since there is no need in proving that the result has the same operational semantics as the original. We name this *strong reflection* as the reflected system is inherently identical to the system it reflects by the fact that the same functionality is being used at both levels.

Logical languages are more expressive than programming languages, as they deal with truth semantics rather than being limited to an operational one. Since computer implementations of such systems are computational in nature, implementing reflection is usually achieved by a re-implementation which includes facts about needed functionality, as well as its operational equivalence. In other words, the context of a theorem-prover naturally leads to the re-implementation which is easily avoided in the case of programming languages. A classic example of this approach is Gödel numbers which are a good theoretical tool, yet unreasonable for practical usage (i.e., as an actual code representation in a compiler).

We demonstrate an implementation of reflection in the Nuprl theorem prover by borrowing the common programming languages technique of exposing existing system functionality to the user-level. The initial idea is to use the very same underlying term representation for user-level term representation. Using strong reflection for this implementation, the embedded system is inherently identical to the original, and the implementation is simple. Still, facts and logical descriptions of such functionality do not exist in a computer program but are necessary to reasoning about it.

2 New Techniques

Term Representation

A fundamental issue for any syntactically-reflected system is the representation of syntactic objects. In Lisp dialects the same data structure is used by both the system and the user, and proper quotations are achieved via the `quote` special form which treats any piece of syntax as accessible data. This approach cannot be used with Nuprl as it uses normalization rather than evaluation: a basic property is that subterms can be replaced by computationally equal ones, and introducing a quotation context term will break this. Our solution for this problem is by *operator shifting* — a shifted operator is one which constructs a representation of the original one, and every operator can be shifted, including already-shifted operators.

Similar to Lisp, Nuprl uses a uniform term structure for syntax: terms have an operator name and subterms. But unlike S-expression, Nuprl terms have explicit binding positions for any binding operator. This introduces a dilemma: one option is making shifted operators represent bindings as subterms, resulting in a familiar concrete syntax, but the result deviates from the original term by changing the binding structure. This option was used in previous work on Reflection in Nuprl [1]. Another option is to keep the same binding structure when shifting an operator — this option is theoretically riskier as it requires redesigning a non-concrete syntax: in Nuprl, a binding operator always denotes a higher order function, meaning that such a representation is a higher order abstract syntax (HOAS, [3]) as explained in [2], where the arguments to shifted operators are *substitution functions*. Considering these options, we have chosen the second based on the availability of a neat implementation which uses the existing functionality for shifted operators. This poses some well known HOAS problems, mainly avoiding *exotic terms* and devising an induction principle; the first is addressed in [2], and the second problem is part of current work.

For example, indicating operator shifting by underlining their name, the Nuprl lambda term that adds one to its input might look like this: `lambda(x.add(x,1))`, and it is denoted by: `lambda(x.add(x,1))`, which is simply gotten by shifting all operators other than bound variable. The same term is also denoted by `lambda(y.lambda(z.add(y,z))(1))`, because the `lambda` operator contains the *same* substitution function. Notice that this last example mixes shifted operators with unshifted ones in a way which is as clear as using quasi-quotations in Lisp.

We visualize shifted operators using the Nuprl display-form engine, indicating different quotation levels with different colors (using underlines in this text), which is as close to true quotations as we can get in this logic.

Term Functionality

By aiming for a practical demonstration of the effectiveness of our methods, we identified necessary functionality. Initially, a 'quote' and an 'unquote' operations are needed system-level functions, which operate on the visible concrete representation — these are "translation" functions which translate external user syntax to the HOAS internal representation. These functions are not visible at the object level, where a 'representation' and a 'reference' operations are used instead to shift levels of user data — these are implemented with an 'up' and 'down' computation rules, and defined to handle bindings properly.

Another fundamental functionality is substitution: obviously, the Nuprl system has substitution functions as part of the implementation, but like 'quote' and 'unquote', these work internally on concrete, fully specified terms. In contrast, user-level term representations might contain "incomplete" terms which contain descriptive parts. For example, `lambda(x.add(x,t))` contains an unknown part, so we cannot do any full substitution over it. The solution for this is a user-level substitution that we know how to execute in single steps: `lambda(x.add(x,t))[e/v]` is reduced to `lambda(x'.add(x',t)[e/v])` where `x'` is a renaming of `x`. This renaming is done with the system-level substitution, resulting in a consistent behavior, while keeping the principle that user-level statements have no access to concrete binding names. (The relation between 'up'/'down' and 'quote'/'unquote' is similar to the relation between the internal substitution operation and the single step version.)

This leads to a related issue: in Nuprl, variables are simple terms, which means that it is now possible to use quotations of such terms: a representation for free variables with user-accessible names. This can be confusing: terms are represented by HOAS object so there is no access to bound variable names, so how can we mix this with quoted free variables? One way to settle this issue is that no user-level 'up' operation can be used to "materialize" a hidden binding name to a visible quotation, and similarly, no user-level 'down' operation can be used on such objects to "capture" them back to a binding. For another viewpoint on this issue, the discussion in [2] of the equivalence between HOAS and concrete terms used coercions from concrete terms to their α-equivalence classes and back — and clearly, the α-equivalence class of the free variable '`x`' is the singleton set '`{x}`'.

Finally, we take a different approach from the usual verbose arguments on the issues of closed terms, free/bound variables and substitution explained using them. We view substitution as the fundamental concept and use it to explain other concepts. For example, free variables are exactly the set of objects that can be substituted, and `closed_term(t)` is defined as the set of terms such that `t[e/v] = t`.

3 Practical Demonstration

Our formalization of reflection is demonstrated by producing a formal proof of the Tarskian argument regarding the internal undefinability of a Truth predicate. The purpose of this was twofold: the first was to show that one can carry out conventional styles of syntactic arguments using the unconventional HOAS. The second purpose was to find out what functionality needs to be exposed for practical purposes. The proof followed a previous paper proof that was written for the sake of clarity rather than for an easy implementation and which was about *concrete syntax* — it preceded the implementation of the necessary tools by almost two years, when it was not at all clear whether we could cope with such problems as quoted free variables, substitution and proper semantics. The surprising result is that not only were we able to complete this a formal proof, but we were able to get the formal version to follow the exact steps of the paper version.

This process demonstrated more than just being able to complete the proof: a computer-aided theorem prover equipped with reflection is a good educational tool, allowing us to inspect different variations as well as clarifying the nature of the new domain of syntactic values. For example, using the system we can get a clear explanation of tricky questions like how '$\uparrow\uparrow t$' differs from '$\underline{\uparrow}t$'. Using existing Nuprl functionality, the text proof itself was converted to Nuprl 'hypertext' containing term objects, making it possible to use Nuprl's display form mechanism for visualizing quotations and for alternative operator syntax.

References

[1] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.

[2] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32. National Aeronautics and Space Administration, 2002.

[3] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

- Additional material extracted from the reflection implementation and the Tarski argument are available on-line at: `http://www.cs.cornell.edu/eli/reflection.html`