First-class Interpreters: Illustrating the Limits Imposed by Representation in a Reflective Language

John Wiseman Simmons II
Daniel P. Friedman
Indiana University

August 8, 1993

1 Introduction

A reflective system allows the user's code to reify and manipulate parts of the system's computational state. The modified state can then be reinstalled, changing the course of the computation. Traditional reflective systems, such as 3-Lisp [5, 16], Brown [7, 18], and Blond [3, 4, 14, 15], provide *reifying procedures*, or *reifiers*, which allow the reification of the current expression, environment, and continuation. This facility allows the addition of new special forms to the language, but does not allow the introduction of such debugging tools as stepping, tracing, or monitoring variable accesses. Nor does it allow introduction of new types of user-defined applicable objects. The reflective object system 3-KRS [13] allows deviating interpreters for individual objects to be installed, but it does not allow reification of the entire interpreter. Part of its interpreter remains in the background.

We extend traditional reflective systems by making the entire interpreter a first-class, reifiable, object. The user can then modify the interpreter to add new forms, as with reifiers, and many other features that reifiers cannot support. Our language, called Refci (reflective extension by first-class interpreters) is implemented in Scheme and may be regarded as a reflective extension of Scheme.

Investigation of the applications of the extensible, first-class interpreter led us to consider the limitations imposed by the choice of underlying representations in a reflective system. Such a system is intended to be open and extensible. Yet our experience shows that it must be designed with forethought to the kinds of extensions a user might want to make. Otherwise, representations that are too closed may impose undue restrictions on the user and render the language inflexible, making it impossible to implement desired extensions.

2 Representation of the first-class interpreter

We get our clue as to how to make the interpreter first-class from denotational semantics. There, a part of the computational state, such as the environment or continuation, is made explicit by making it an argument of the semantic functions. We make our interpreter explicit by making it an argument of itself: it is written in what might be called *interpreter-passing style*. A representation of it is thus available at all times to be reified. This leads to a consideration of the choice of the interpreter's representation.

In most interpretive systems, the interpreter is in the background, not accessible to the user. We desire our representation of the interpreter to include the entire system, leaving little or nothing in the background if possible. A table representation, containing a list of expression types and actions to interpret each type of expression, would not satisfy this criterion, since a driver routine in the background is still required to interpret the table.

So instead we would like to represent the interpreter as a Scheme procedure. This is a natural way to package the entire system in the representation, and also would allow the interpreter to be extended by function composition. Unfortunately, using a single procedure does not work. We want to do two kinds of extension: adding new special forms and adding preliminary actions to be done before the dispatch on the type of the expression begins. Adding both to a single procedure would result in intermixing preliminary actions with new special forms. This would be incorrect since preliminary actions shadowed by a special form might not be carried out.

Accordingly, we represent the interpreter by two procedures, a *prelim* and a *dispatch*. The dispatch is the main part of the interpreter. It determines the expression's type and performs the appropriate action. The prelim represents actions that are to be performed before the dispatch each time the interpreter is called. Inclusion of the prelim is the key that allows the implementation of such features as stepping, tracing, and simulation of multiprocessing.

We represent the infinite reflective tower by a self-referencing *metacontinuation*. It simulates all levels above the current active level. Each level is represented by its continuation and interpreter. Thus the type of the metacontinuation is

$$\mathrm{MetaCont} = \mathrm{Cont} \times \mathrm{Prelim} \times \mathrm{Dispatch} \times \mathrm{MetaCont}$$

The dispatch takes the usual expression, environment, and continuation arguments, together with the prelim, dispatch, and metacontinuation:

$$\begin{aligned}\mathrm{Dispatch} &= \mathrm{Expr} \to \mathrm{Env} \to \mathrm{Cont} \to \mathrm{Prelim} \to \mathrm{Dispatch} \to \mathrm{MetaCont} \to \mathrm{Ans}\\ &\approx (\mathrm{Expr} \times \mathrm{Env} \times \mathrm{Cont} \times \mathrm{Prelim} \times \mathrm{Dispatch} \times \mathrm{MetaCont}) \to \mathrm{Ans}\end{aligned}$$

A prelim acts as a dispatch transformer, turning a dispatch into another that performs the preliminary action first:

$$\mathrm{Prelim} = \mathrm{Dispatch} \to \mathrm{Dispatch}$$

The whole interpreter can be obtained by passing the dispatch to the prelim, and it has the

```
(make-prelim
  (lambda (dd e r k p d)
    (if (is-type? e 'foo)
        ...
        (dd e r k p d))))
```

Figure 1: Outline of a prelim implementing a special form

same type as the dispatch.

Now all extension can be done by writing prelims. For example, a special form foo can be added by extending the dispatch with a prelim of the type shown in Figure 1. It checks whether the expression *e* is of the type it handles. If so, it handles it; if not, it passes control to the dispatch *dd* it extends. If this prelim handles the expression, the current interpreter, represented by *p* and *d*, is used to interpret the subexpressions of *e*. User code for a prelim, such as that shown here, does not take the metacontinuation as an argument. The metacontinuation is not directly accessible by the user and is managed entirely by the system.

The user can add a preliminary action to the interpreter by writing it as a prelim and using it to extend the existing prelim. The default prelim simply calls its first dispatch argument *dd* and serves as a basis for extension.

A prelim must be interpreted like any other user code. Since it is to be run at the level of the interpreter, there must be a shift one level up in the tower (metacontinuation), accessing the interpreter one level above to interpret the prelim's code. The primitive make-prelim, which turns user closures into prelims, incorporates into the prelim representation all necessary shifting code to cause the shift up when the prelim is run and back down after it completes.

A further feature of prelims arises since the Refci interpreter is written in continuation-passing style. All calls to any prelim or dispatch are tail calls, meaning that they never cause the Scheme control stack to grow. Instead, what is yet to be done is remembered in the interpreter's continuation argument. The action of a prelim can include installing a new continuation, representing an action to be performed after the evaluation of the current expression. Thus no "postlim" component of the interpreter is needed, and prelims can function as both the before and after methods of CLOS.

3 Limits imposed by representations

In this section we illustrate the limits that underlying representations can impose on a reflective system, and the care and forethought needed in making design decisions. Our illustration is the use of our reflective system to implement *quasi-static abstractions*, developed by Lee and Friedman [12].

3.1 Quasi-static abstractions

Quasi-static abstractions allow sharing of variable bindings between lexical scopes. We can declare a variable to be *quasi-static* within a scope. Such a variable does not receive any binding from its lexical scope. It is totally unbound unless it is later *resolved* with a binding from some other scope.

We extend Scheme with special forms **quasi-static** and **resolve**. The **quasi-static** form declares certain variables to be unbound within its body. Any procedures created within a **quasi-static** expression are quasi-static abstractions. For example,

```
(define qsa-xy
  (quasi-static (x y)
    (lambda (a)
      (list a x y))))
```

creates a quasi-static abstraction *qsa-xy* in which x and y are quasi-static. Applying *qsa-xy* causes an unbound variable error when the quasi-static variables are referenced.

We can resolve some or all of the quasi-static variables in a quasi-static abstraction with the **resolve** form. This form creates a new quasi-static abstraction, which has in its environment the bindings taken from the lexical scope of the **resolve**. For example,

```
(define qsa-y
  (let ([x 2])
    (resolve (x) qsa-xy)))
```

The quasi-static abstraction *qsa-y* still contains y as a quasi-static variable. We create a closure with

```
(define c
  (let ([y 3])
    (resolve (y) qsa-y)))
```

The closure c shares the code of *qsa-xy* but its variable bindings are from two different lexical scopes. The application (c 1) returns (1 2 3). Bindings once established cannot be changed. Thus the closure *c2* produced by

```
(define c2
  (let ([x 5] [y 6])
    (resolve (x y) c)))
```

is the same closure as c.

3.2 Implementing quasi-static abstractions

Our implementation depends largely on the features available in the implementation language. We shall mention four representations, determined by two choices of representation:

- The representation of quasi-static variables
- The representation of quasi-static abstractions

3.2.1 Representation of quasi-static variables

We can use the environment to store information about quasi-static variables. The two methods we consider are explicit tagging and removal of bindings.

Representation by tagging amounts to extending the environment in the usual way with bindings of the quasi-static variables to a special tag ***quasi-static***. Declaring variables quasi-static thus amounts to a simple environment extension. Resolving a quasi-static variable requires two lookups. We must look up the variable in the environment where it is to be resolved to see if it is fact still quasi-static. We then look it up in the current environment and extend the environment where it is to be resolved with its binding in the current environment. This causes the two environments to share the binding, so that a side effect to one affects both.

Using this representation requires the ability to modify the variable lookup. This cannot be done with reifiers, but can be done with the extensible interpreter. When we look up a variable, we must check whether it is still quasi-static. If so, we report an error. If not, the computation can proceed.

The tagging representation is most natural, since it requires features most likely available to the user in any reflective system: the ability to extend reified environments and to look up variables in them. Yet even here we get our first taste of the forethought necessary in designing the environment representation: the user must be able to look up the cell holding the value, not just the value. Else sharing of bindings would be impossible. Such a feature might not normally be included in an environment representation.

As an alternative, we could handle a **quasi-static** declaration by removing the quasi-static variable's bindings from the environment. To resolve a variable, we extend the environment at its base instead of its top, so that the new binding does not shadow any that already appear. In this representation, declaring a variable quasi-static involves more work, since the environment must be searched and all the variable's bindings removed. Resolving a variable is simple. We can perform the extension at the base without even bothering to check whether the variable is quasi-static in the environment. No harm is done since no previously occurring binding will be shadowed. Also, no modification to the default variable lookup is necessary. In this representation, quasi-static variables are indistinguishable from unbound variables.

This representation requires features that likely would not be expected to be available in an environment abstract data type: the ability to remove bindings and to extend the environment below. Again we see that forethought about the kinds of use the user might want to make of reified environments is required if the design is to be flexible enough to support this option.

3.2.2 Representation of quasi-static abstractions

The choice of representation for quasi-static abstractions is influenced by the underlying representation of closures and reifiers. If this representation is sufficiently open to have its lexical environment accessible, then underlying closures are in effect already quasi-static abstractions, and all that is necessary is to write the code to handle the **quasi-static** and **resolve** special forms.

If the underlying representation of closures does not make the lexical environment accessible, then we must define our own representation for quasi-static abstractions. This implies that the system must allow the addition of user-defined types of applicable objects. Now **lambda** expressions must evaluate to quasi-static abstractions, so we must override the default closure creation. The system does not know how to apply this representation, so we must do it ourselves. This is done by writing a prelim to handle application and using it to shadow the default application line. To handle an application, it evaluates the application's operator and checks whether it is a user-defined quasi-static abstraction. If so, it handles the application. If not, it passes control to (presumably) the default dispatch to handle it.[1]

Another subtlety appears at this point. Since the user can install any applicable object (of one argument) as a continuation, the underlying system must be able to handle our new representation of quasi-static abstractions as continuations. Again, this is a point that would be easy to overlook when implementing a reflective system.

3.2.3 Impact of syntax of reifier creation

Another design decision interacts with the representation of closures and reifiers to determine the possibility of implementing quasi-static reifiers. We might, as in Brown, create reifiers by passing a closure to the primitive make-reifier:

(make-reifier (**lambda** (e r k p d) ...))

On the other hand, we might, as in Blond, create reifiers by a special form:

(**alpha** (e r k p d) ...)

Suppose that the underlying system uses a closed representation for closures and reifiers, so that we must define our own representation for the quasi-static versions. If reifiers are

[1]Unfortunately, this introduces non-compositionality into the system. Since we do not wish to repeat the work of evaluating the operator, we must create a new expression representing the application of the evaluated operator to the yet unevaluated operands. It is this expression that is passed on to be handled by the default dispatch.

Environments in quasi-static abstractions					
Variable representation	Extension		Modify lookup	Remove bindings	Which variables are quasi-static?
	Above	Below			
Removal		$\surd$		$\surd$	All unbound
Tagging	$\surd$		$\surd$		Declared only

Table 1: Features determining variable representation

Representations of quasi-static abstractions			
Representation	Environment accessible	Override application	Override **lambda**
Underlying	$\surd$		
User defined		$\surd$	$\surd$

Table 2: Features determining quasi-static abstraction representation

created by special form, all is well. We can handle interpretation of the **alpha** form and create quasi-static reifiers properly. If reifiers must be created by make-reifier, however, we face an insoluble problem. The argument to make-reifier can be an arbitrary closure created at any point in the program. Since the closed representation is in use, we cannot obtain the closure's lexical environment to include in the reifier's representation. Only with complete analysis of all uses that might ever be made of this fragment could we determine the locations of all **lambda** expressions that might generate closures it could be required to turn into reifiers, and this is in general impossible. So we cannot even implement quasi-static reifiers if the underlying representation has closed representation of closures and also requires that reifiers be created from closures.

We summarize the decisions discussed in this section in Tables 1 and 2.

4 Conclusion

The power of a reflective system can be extended by making the system's interpreter a first-class object. Reifiers, the standard means of allowing the user to modify a reflective system, provide only the ability to add new special forms to the language. Extensible first-class interpreters also allow the addition of actions to be performed before and after each interpretation, as well as the modification (by shadowing) of the basic actions of the interpreter: the interpretation of identifiers and compounds (applications). This can all be done in a uniform way through the interpreter by adding prelims. Yet for efficiency it is desirable to retain reifiers in the language.

The use of first-class interpreters to implement quasi-static abstractions leads us to consider the limitations of reflection. The openness and extensibility we desire can conflict with simplicity of design. The extensibility of a reflective system depends on the choices made for the representations of its underlying structures. Sufficient forethought must be taken of the kinds of extensions the user might want to make so that the system being designed will be flexible enough to handle them. This is at variance with the premise that simply making a system reflective is enough to make it as open and extensible as the user may desire.

References

[1] Clinger, W., and Rees, J., "Revised^4 report on the algorithmic language Scheme," *Lisp Pointers 4*:3, pp. 1–55, 1991. Also available as a technical report from Indiana University, MIT, and the University of Oregon.

[2] Common Lisp Object System Specification X3J13 Document 88-002R. *SIGPLAN Notices* (September 1988).

[3] Danvy, O., and Malmkjær, K. Aspects of computational reflection in a programming language. Extended abstract (1988).

[4] Danvy, O., and Malmkjær, K. Intensions and Extensions in a Reflective Tower. *ACM Conference on Lisp and Functional Programming* (1988) 327–341.

[5] des Rivières, J., and Smith, B. The Implementation of Procedurally Reflective Languages. *Proceedings of the Symposium on Lisp and Functional Programming*, ACM (August 1984) 331–347.

[6] Dybvig, K. *The Scheme Programming Language.* Prentice-Hall (1987).

[7] Friedman, D., and Wand, M. Reification: reflection without metaphysics. *Proceedings of the Symposium on Lisp and Functional Programming* ACM (August 1984) 348–355.

[8] Haynes, C., and Friedman, D. Abstracting timed preemption with engines. *Journal of Computer Languages* (Pergamon Press) **12**, 2 (1987) 109–121.

[9] Jagannathan, S. Environment-based reflection. Technical Report 91-001-3-0050-1, NEC Research Institute. Princeton, NJ (January 1991).

[10] Jefferson, S., and Friedman, D. A Simple reflective interpreter. *Proceedings of the International Workshop on New Models for Software Architecture '92: Reflection and Meta-level Architecture.* Edited by Akinora Yonezawa and Brian C. Smith. Tokyo, Japan (November 4–7, 1992). 48-58.

[11] Keene, S. *Object-oriented programming in Common Lisp: A programmer's guide to CLOS.* Addison-Wesley, Reading MA (1989).

[12] Lee, S.-D., and Friedman, D. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 479–492, 1993.

[13] Maes, Pattie. Computational reflection. Technical Report 87_2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel.

[14] Malmkjær, K. On some semantic issues in the reflective tower. *Fifth Conference on Mathematical Foundations of Programming Semantics.* Springer-Verlag, *Lecture Notes in Computer Science* 442 (1990).

[15] Malmkjær, K. A Blond Primer. DIKU Research Report 88/21 (September 12, 1988).

[16] Smith, B. Reflection and semantics in a procedural language. MIT/LCS/TR-272 (1982).

[17] Steele, G. and Sussman, G. The Revised Report on SCHEME. MIT AI Memo #452. Cambridge, MA (1978).

[18] Wand, M., and Friedman, D. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation* **1**, 1 (June 1988) 11–38.