A Reflective System is as Extensible as its Internal Representations: An Illustration

Indiana University Computer Science Department
Technical Report #366

John Wiseman Simmons II*
Daniel P. Friedman†

October 21, 1992

Abstract

Reflective systems are intended to be open enough to allow the user to extend and modify them easily. In this paper we show that the openness and extensibility of a reflective system depends to a great degree on the choice of its underlying representations. Giving the language the necessary expressive power requires forethought in the design stage of the kinds of extensions the user might wish to make. We illustrate this conclusion by considering what features a reflective language should have in order to implement quasi-static binding. Since quasi-static binding involves modification of the environment, it is a natural candidate for reflective implementation. We show that the ease, and even possibility, of the implementation depends on the representation choices of the underlying reflective system.

1 Introduction

Reflective systems allow the user to extend the system at run time. The standard procedurally reflective languages, such as 3-Lisp [13, 4], Brown [6, 14], and Blond [2, 3, 11, 12], allow the user to add new special forms to the language by writing *reifying procedures*, or *reifiers*, to be run at the level of the interpreter. While powerful, this facility limits the extensibility of the language. For example, reifiers cannot modify variable lookup, closure creation, or procedure application—the basic features of the language.

In this paper we investigate the use of a reflective system to implement *quasi-static binding* [9], which uses *quasi-static abstractions* to allow the sharing of bindings between multiple lexical scopes. Quasi-static abstractions are closures in which certain variables are *quasi-static*, having no binding in the lexical scope. Quasi-static variables can later be *resolved* with bindings from other lexical scopes, thus allowing shared bindings. Such a binding discipline is useful in modeling inheritance and method creation in object-oriented systems.

The experience of implementing quasi-static abstractions in a reflective system gives us insight about the limitations of such a system. Though intended to be flexible and extensible, such a system

*Supported by grants NSF DCR 85-21497 and NSF CCR 89-01919. Email address: *simmonsj@cs.indiana.edu*
†Supported by grants NSF CCR 87-02117 and NSF CCR 90-00597. Email address: *dfried@cs.indiana.edu*

will lack the necessary expressive power unless forethought is given in its design to the types of extensions the user might wish to make. We discuss our findings about the underlying design decisions necessary to give the system the flexibility we desire. These decisions are incorporated in the language Refci[**?**], an extensible reflective language with first-class interpreters.

Section 2 briefly describes quasi-static abstractions and their use in object-oriented systems. Section 3 discusses the implementation of quasi-static abstractions in a reflective system and the features the system must have to support them. Section 4 discusses the Refci language and its use as a platform for the implementation of quasi-static abstractions. Section 5 gives the details of the implementation representing quasi-static abstractions as user-defined applicable objects. Section 6 describes related work and presents the conclusion.

2 Quasi-static abstractions

In a lexically scoped language such as Scheme [1, 5], all non-global variables obtain their bindings from the enclosing lexical scope. There is no way to export a binding from one scope to another. To allow such sharing, we can declare certain variables to be *quasi-static* within a scope. A quasi-static 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 define the special forms needed to create quasi-static abstractions and show how they can be used to create an inheritance-based object-oriented system.

2.1 Quasi-static and resolve

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)))
```

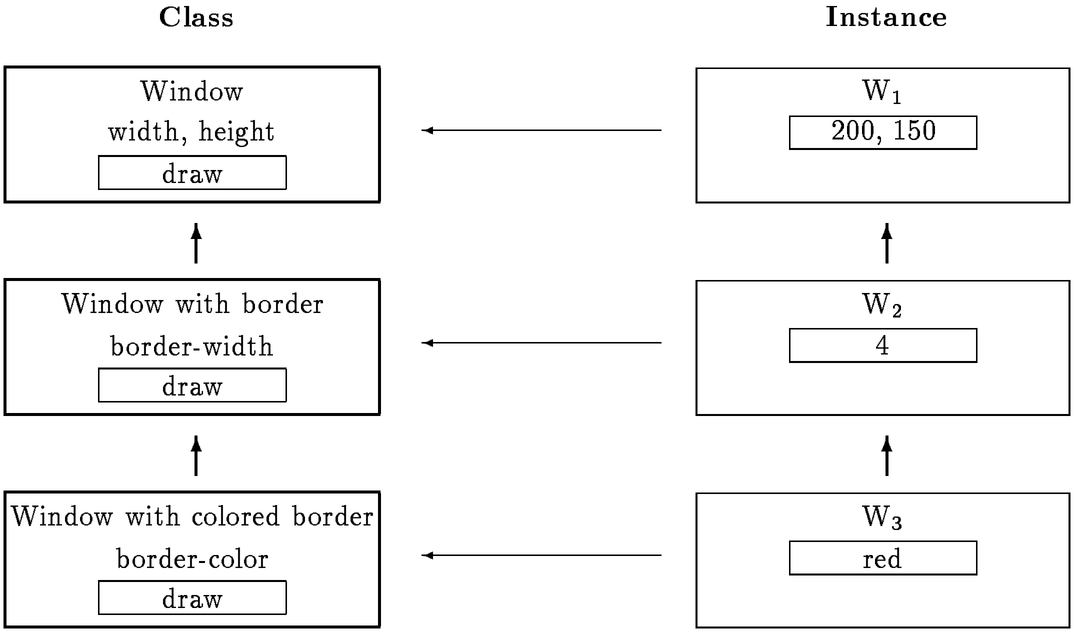

Figure 1: An inheritance example

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.

2.2 Implementing inheritance

The ability to obtain bindings from different lexical scopes can be used in object-oriented systems for defining inheritance. Figure 1 shows a typical example. It defines a class of windows with colored borders, which inherit from windows with borders and from windows. Each class has its instance variables and a draw method. The draw method for the class of windows with colored borders refers to the instance variables for all classes. The code of the method is shared over the whole class, so the values of the instance variables it must use are not known until the instance to which it is to be applied is determined.

The inheritance can be handled by making the method an *open method*, represented by a quasi-static abstraction. The instance variables are declared quasi-static in the method's body, and are resolved only when the instance to which the method is to be applied is known. Thus, the method could be represented as

(**quasi-static** (*width height border-width border-color*)
  (**lambda** (*position*) ...))

Each instance could then be represented as a *resolver*, a closure that establishes a binding for a quasi-static variable:

(**let** ([*border-color* 'red])
  (**lambda** (*open-method*)
    (**resolve** (*border-color*) *open-method*)))

When a method is to be applied, the inheritance hierarchy is traversed from bottom to top, resolving all instance variables at each level. The ability to resolve in several scopes makes this possible, while the inability to resolve a variable again once it is resolved assures that an instance variable will properly shadow an identical variable in a superclass.

3 Implementing quasi-static abstractions in a reflective system

The way we implement quasi-static abstractions in a reflective system depends to a large extent on the tools the underlying system provides. The experience we gain from writing the implementation sheds light on the design of the reflective system, and shows that such a system must be designed with forethought about the kinds of extensions the user might want to make. We shall discuss four implementation models, determined by two representation choices. The choices are:

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

We shall discuss these in turn. We conclude the section with a discussion of the interaction of the creation strategy and underlying representation of reifiers on the creation of quasi-static reifiers.

3.1 The representation of quasi-static variables

Using the tools provided by the reflective system, we can keep information about quasi-static variables in the environment. This gives correct behavior with respect to quasi-static abstraction creation. A closure formed within a **quasi-static** declaration will be created in an environment in which the quasi-static variables have been appropriately designated. No other record of the quasi-static variables need be kept.

A disadvantage of this representation is that there is in general no way to tell when all the quasi-static variables in a quasi-static abstraction have been resolved. In the implementation of the object-oriented system mentioned in Section 2.2, we prepared to apply a method by traversing the inheritance chain, resolving the quasi-static instance variables at each level. It would save work if we could stop when there were no more quasi-static variables in the method, even though several levels remained. To do this we would have to keep a separate list of quasi-static variables throughout the computation, complicating the implementation and going outside the reflective system.

3.1.1 Representation by explicit tagging

One choice is to keep the information in the environment explicitly. When a variable is declared as quasi-static, we extend the environment with a binding of the variable to a special value (tag) marking it as quasi-static. An advantage of this model is that a variable is quasi-static only if it has been specifically declared quasi-static, and we can always test whether a variable has been so declared. Also, a **quasi-static** declaration is cheap, amounting to a simple environment extension. For this representation to work, the environment abstract data type must provide the ability to do ordinary functional extension of environments, something we may reasonably expect to be provided.

The resolve operation for this representation is more expensive. We must look up each variable to be resolved and see if it is marked as quasi-static in the environment. If it is not, we do nothing, since we do not allow existing bindings to be changed. If it is, we shadow the tag binding with the one from the context of the **resolve** expression. We use the actual cell from this context to allow sharing of bindings across multiple lexical scopes.

This choice also requires that we be able to modify the system's variable lookup procedure. When looking up a variable that occurs within the scope of a **quasi-static** declaration, we must check the value returned to see whether the variable is quasi-static, and signal an error if it is.

3.1.2 Representation by removing bindings

From a semantic point of view, we can think of a quasi-static variable as having no binding in the current environment. This leads to a second choice of representation. When a variable is declared quasi-static, we remove its bindings from the current environment. To resolve a quasi-static variable, we extend the environment below—that is, we adjoin the new binding at the bottom. If we define an environment concatenation operation by

$$(\rho_1 \oplus \rho_2)(x) = \begin{cases} \rho_2(x) & \text{if } \rho_2(x) \text{ is defined} \\ \rho_1(x) & \text{otherwise} \end{cases}$$

then extending ρ below by a binding for v amounts to constructing $\rho_v \oplus \rho$, where ρ_v leaves all variables except v unbound. Concatenating the new binding at the bottom maintains static scoping. Placing it at the top, as in [7], would lead to dynamic binding.

In this representation, the **quasi-static** declaration becomes expensive, while the **resolve** operation is cheap. The search in the environment for a quasi-static variable is now done when it is declared quasi-static rather than when it is subsequently resolved. Furthermore, we must search the entire environment and remove all the variable's bindings. In the previous model it was enough just to find the topmost binding to verify whether the variable was quasi-static.

The **resolve** operation now requires no search at all. If the variable to be resolved is quasi-static, it will not occur in the environment above the new binding, so the proper binding will be seen. If it is not quasi-static, the correct binding above will shadow the new one, and the correct binding will still be found. This enforces the convention that **resolve** cannot be used to change a binding that already exists.

Variable lookup in this model is standard, since there is no need to test for a special tag. Thus the underlying system need not allow its default variable lookup to be overridden in order to support this model.

This model requires that the environment abstract data type allow environments to be extended below, or equivalently that it support the environment concatenation operator $\oplus$ defined above. It also requires the ability to remove bindings from an environment. These are features that we might not expect to find supported in a standard environment representation. They provide examples of the kind of design decisions that must be made to make a reflective system flexible.

A characteristic of this model is that quasi-static variables are not only those that have been declared quasi-static. Any variable with no binding in the current environment is quasi-static by default. This means that **resolve** can be used to bind any unbound variable, not just those that have been declared quasi-static. Thus, for example,

```
(let ([qsa1 (quasi-static (x) (lambda () (+ x y)))])
  (let ([x 3] [y 4])
    (let ([qsa2 (resolve (x y) qsa1)])
      (qsa2))))
```

yields 7 in this model, while its value in the original model depends on the global binding of y.

3.2 The representation of quasi-static abstractions

The amount of work necessary to implement quasi-static abstractions depends on the underlying system's representation of closures. We consider two cases. If the internal structure of the closure, specifically the environment, is not accessible, then we must define a new representation for quasi-static abstractions. This representation will have the creation-time environment of a quasi-static abstraction explicitly represented, so that it can be obtained and modified.

The underlying system has various kinds of applicable objects, such as primitives, closures, environments, continuations, and reifiers. But it does not know how to apply quasi-static abstractions. So we must be able to override the default procedure application in order to provide an application strategy for quasi-static abstractions. This is not possible in standard reflective systems. We shall show how the reflective system we use allows it.

Since any **lambda** expression in the scope of a **quasi-static** declaration must generate a quasi-static abstraction, we must also be able to override the default closure creation mechanism. This is possible in standard reflective systems since **lambda** is a special form.

Allowing the definition of new types of applicable objects on the fly poses another problem. Any applicable object that takes one argument can be used as a continuation when spawning a new level. To fully support user-defined applicable objects, the system must be able to use them as continuations. This cannot be added on the fly, but must be built into the system design. It makes the system's semantics non-compositional. For the system does not know how to apply such a continuation. It must form a new expression representing the application, and interpret it using the extensions we have provided.

The other case is that the underlying representation of closures makes the environment explicit. This case allows a much simpler implementation, since there is no need to define a new representation for quasi-static abstractions. We can represent quasi-static abstractions as ordinary closures. Since the environment is explicit, all closures are in effect quasi-static abstractions. All we need to do is add the special forms **quasi-static** and **resolve**. There is no need to be able to override the default closure creation and application.

In this representation, there is no way to tell a closure from a quasi-static abstraction. This is really no loss. In neither representation are we able to ascertain the important fact: when all the quasi-static variables in a quasi-static abstraction become resolved.

3.3 The creation and representation of quasi-static reifiers

In Brown, reifiers are created by passing a closure to the primitive make-reifier:

```
(make-reifier (lambda (e r k) ...))
```

Blond, on the other hand, creates reifiers by the special forms **gamma** and **delta**:

(**gamma** (e r k) ...)

Refci follows Blond in creating its reifiers by a special form. Since the interpreter handles this special form directly, it is easy to include extensions to create the proper representations of quasi-static reifiers.

Suppose instead that Refci used make-reifier to create its reifiers, both ordinary and quasi-static. Suppose also that the underlying representation of closures and reifiers is being used. Then the creation-time environment of the quasi-static abstraction used to form a quasi-static reifier is not available when the reifier is created. For application this is not a serious problem, since the quasi-static abstraction's stored environment will be obtained and used when the quasi-static abstraction is applied.

Resolving a quasi-static reifier, however, now presents an insoluble problem. The constituent quasi-static abstraction, in which the environment is explicit, is now encased in the system representation of the reifier, in which the environment is not explicit. Thus there is no way this environment can be obtained to resolve quasi-static variables.

Using make-reifier with the open underlying representation, however, presents no problem. In this case, make-reifier simply obtains the environment from its closure argument and includes it in its result.

Once again, the representation choices of the underlying system have led to unexpected problems. In this case, a choice of syntax interacts with a choice of internal representation to determine even the possibility of implementing quasi-static reifiers.

4 Implementation in Refci

The discussion in Section 3 shows that there are several features a reflective system should have to permit the implementation of quasi-static abstractions. Different features are necessary for each of the possible representations. For all but the simplest of the cases, the standard reflective systems will not suffice. We describe the Refci language and show that it is a suitable platform for implementation of and experimentation with the various representations of quasi-static abstractions.

4.1 The Refci language

Refci is a reflective dialect of Scheme with first-class extensible interpreters. It uses the infinite reflective tower model, with every level of the tower having its own interpreter as well as its own continuation. User (Refci) code can be added to the default interpreter, allowing the user to extend or modify the system on the fly. To interpret user code in the interpreter, the system shifts up and uses the interpreter from the next higher level. The user code can shift back down by applying a reified interpreter. Thus all level shifting can be handled through the extensible interpreters, instead of by reifiers as in standard reflective systems.

An extensible interpreter is represented as two procedures, a *prelim* and a *dispatch*. The dispatch procedure does the main work of interpreting the expression, while the prelim allows additional actions to be performed before (or after) the main interpretation. Composing a prelim with a prelim yields another prelim. Composing a prelim with a dispatch gives the entire interpreter (itself of type dispatch). The interpreter is divided in this way to allow a uniform method of extension.

Any code the user wants to add to the interpreter is written as a prelim procedure. Adding the new prelim to the dispatch amounts to adding a new line or modifying an existing line. Adding the prelim to the old prelim adds an action to perform before the dispatch. This uniform style of functional extension would not be possible if the interpreter were represented as a single procedure.

For example, we can add a new special form to exit the current level and continue computation one level up the tower. We write it as a prelim and use it to extend the dispatch. The special form **common-define** places a definition in the common environment, shared by all levels.

```
(common-define exit-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (and (pair? e) (eq? (1st e) 'exit))
          (if (null? (cdr e))
              "Exit wants an argument."
              (p d (2nd e) r (lambda (x) x) p d))
          (dd e r k p d)))))
```

The *dd* argument is the dispatch being extended, while *p* and *d* represent the complete interpreter, used to evaluate the argument. If the expression is not an *exit* expression, the call to *dd* continues the dispatch. The system shifts down when applying *p* (or *dd*). The shift up to end the level comes from ignoring the lower level's continuation *k* and giving the lower level an identity continuation instead. The primitive make-prelim changes the type of the user's procedure to make it suitable for inclusion in the interpreter.

Though reifiers are no longer necessary, we keep them in the language for convenience. Reifiers are created by the special form **alpha**. For example, (**alpha** (*e r k p d*) *body*) creates a reifier, closed in the lexical scope, that obtains the current expression, environment, continuation, prelim, and dispatch, then evaluates *body* at the next level up.

To evaluate an expression in an extended interpreter we can use the primitive install, which evaluates its expression argument using its prelim and dispatch arguments (and the current environment and continuation). The numbered prompts show the shift from level 0 to level 1.

```
0-1> (install
       '(exit 30)
       (reify-new-prelim)
       (extend-dispatch exit-prelim (reify-new-dispatch)))
1-0: 30
1-1>
```

4.2 Sketch of the implementation

The implementation is simplest when the underlying system provides these features:

- Closures with explicit, accessible environments
- Removal of bindings from environments
- Extension of environments from below

```
(common-define quasi-static-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (is-type? 'quasi-static e)
          (p d (3rd e)
             (if (null? (2nd e))
                 r
                 (remove-bindings (2nd e) r))
             k p d)
          (dd e r k p d)))))
```

Figure 2: Adding quasi-static variables: first case

In this case, we do not need to define a new type of applicable object nor override the default variable lookup, closure creation, or application. This model can be implemented with reifiers, since it requires only the addition of the special forms **quasi-static** and **resolve**.

Figure 2 shows this implementation of **quasi-static**. All that is required is to evaluate the body of the **quasi-static** expression in an environment with the bindings of the quasi-static variables removed.

The implementation of **resolve** in Figure 3 is also straightforward. We evaluate the body, which should evaluate to a quasi-static abstraction (else **resolve** does nothing). The expression's quasi-static variables are looked up in the current environment to get their cells, and these cells are used to extend the quasi-static abstraction's stored environment. The auxiliaries *qsa→env* and *replace-env* get a quasi-static abstraction's stored environment and form a new quasi-static abstraction from an old by replacing its stored environment.

Implementing any of the other representations requires overriding the system's default behavior. This cannot be done with reifiers, but can be done with the Refci extensible interpreter. Of the three remaining cases, we discuss only one here, deferring the others to Section 5.

The next simplest model also uses the underlying system's closure representation, but extends the environment with tags for quasi-static variables. This requires modification of the default variable lookup. We do this by extending the default dispatch with *quasi-static-lookup-prelim* (Figure 4). This shadows the default variable lookup and allows us to signal an error (and exit the current level) if the variable being looked up is quasi-static.

The code to add **quasi-static** and **resolve** has the same basic form as in the simplest case. When evaluating the body of a **quasi-static** expression, we must extend the default dispatch with *quasi-static-lookup-prelim*. Instead of removing the quasi-static variables from the environment, we extend it with bindings that mark them as quasi-static:

```
(extend-reified-environment
  (2nd e)
  (map (lambda (i) (make-cell '***quasi-static***)) (2nd e))
  r)
```

To resolve variables in a quasi-static abstraction, we now extend its stored environment above instead of below. Also we must look up each variable to be resolved to see if it is still quasi-static.

```
(common-define resolve-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (is-type? 'resolve e)
          (p d (3rd e) r
            (lambda (qsa)
              (k (if (or (is-type? 'closure qsa)
                         (is-type? 'alpha-reifier qsa))
                     (let ([stored-r (qsa→env qsa)]
                           [quasi-static-vars (2nd e)])
                       (if (null? quasi-static-vars)
                           qsa
                           (replace-env qsa
                             (extend-reified-environment-below
                               quasi-static-vars
                               (map (lambda (i) (L-lookup i r)) quasi-static-vars)
                               stored-r))))
                     qsa)))
            p d)
          (dd e r k p d)))))
```

Figure 3: Resolving quasi-static variables: first case

```
(common-define quasi-static-lookup-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (symbol? e)
          (let ([v (r e)])
            (if (eq? v '***quasi-static***)
                (list "Unbound quasi-static variable" e)
                (k v)))
          (dd e r k p d)))))
```

Figure 4: Variable lookup in a **quasi-static** expression

```
(common-define make-qsa-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (cond
        [(is-type? 'lambda e)
         (k (list 'qsa (2nd e) (3rd e) r))]
        [(is-type? 'alpha e)
         (k (list 'reflective-qsa (2nd e) (3rd e) r))]
        [else
          (dd e r k p d)]]))))
```

Figure 5: Creating quasi-static abstractions

5 Defining new applicable objects: the complex cases

The representation of closures in the underlying system may not be suitable to represent quasi-static abstractions, or we may choose to define a new representation for other reasons. In this case we must define a new representation for quasi-static abstractions, and provide the mechanism to apply these new objects.

Within the body of a **quasi-static** expression, each **lambda** or **alpha** form must give rise to a quasi-static abstraction. While evaluating the body, we extend the dispatch with *make-qsa-prelim* (Figure 5), shadowing the default handling of these forms. We choose a list representation to correspond to the representation of applicable objects in the underlying system and facilitate type checking.

5.1 Applying ordinary quasi-static abstractions

The system cannot apply the new representation of quasi-static abstractions. We must modify (shadow) the application line of the interpreter to check for and handle the case when the operator is a quasi-static abstraction. Figure 6 gives the code. We first evaluate the operator to determine whether it is a quasi-static abstraction. Applying a quasi-static abstraction is like applying a closure: we evaluate the arguments (with *evlis*), then evaluate the body of the quasi-static abstraction in an environment extended with bindings for the arguments. Since the code we are evaluating was originally within the body of a **quasi-static** expression, we must use an interpreter *df* extended to handle such code.

If the operator is not a quasi-static abstraction, we must allow the default interpreter to continue processing the application. We cannot simply start over because we have done part of the work in evaluating the operator. Proceeding non-compositionally, we create a new application from the unevaluated arguments and a fresh variable. The new expression is evaluated in an environment with the fresh name bound to the value of the operator. For this evaluation, we use the interpreter *dd* without the new application code, allowing the default interpreter to handle the application.

```
(common-define apply-qsa-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (application? e)
         (p d (1st e) r
           (lambda (f)
             (cond
               [(is-type? 'q-s-abstraction f)
                (let ([df (extend-for-quasi-static-body d)])
                  (evlis (rest e) r
                    (lambda (lv)
                      (p df (3rd f)
                         (if (null? (2nd f))
                             (4th f)
                             (extend-reified-environment
                               (2nd f)
                               (map make-cell lv)
                               (4th f)))
                         k p df))
                    p d))]
               [(is-type? 'reflective-q-s-abstraction f)
                (install
                  '(meta-alpha f (rest e) r k p d)
                  p
                  (extend-dispatch meta-alpha-prelim d))]
               [else
                 (let ([fname (gensym)])
                   (p dd (cons fname (rest e))
                      (extend-reified-environment
                        (list fname) (list (make-cell f)) r)
                      k p d))]))
           p d)
         (dd e r k p d)))))
```

Figure 6: Applying a quasi-static abstraction

```
(let ([x ...])
  (alpha (e r k p d)
    (if x
        (p d (1st e) r k p d)
        (k (1st e)))))
```

Figure 7: A reifier in a lexical scope

```
(common-define meta-alpha-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (is-type? 'meta-alpha e)
          (let ([df (extend-for-quasi-static-body d)])
            (evlis (rest e) r
              (lambda (lv)
                (p df (3rd (1st lv))
                   (extend-reified-environment
                     (2nd (1st lv))
                     (map make-cell (rest lv))
                     (4th (1st lv)))
                   k p df))
              p d))
          (dd e r k p d)))))
```

Figure 8: Handling a reifying quasi-static abstraction

5.2 Applying reflective quasi-static abstractions

Reifying procedures can also be quasi-static abstractions. Reifying procedures in Refci are created using the special form **alpha**, and are closed in the lexical environment in which they are formed. They allow the user access to the current prelim and dispatch as well as the expression, environment, and continuation. For example, the reifying procedure created in Figure 7 returns its first argument, evaluated if x contains a true value, and unevaluated otherwise.

Our extensions allow **alpha** expressions within **quasi-static** expressions and evaluates them to quasi-static reifiers. The procedure *make-qsa-prelim* constructs the quasi-static reifiers, and *apply-qsa-prelim* applies them. This evaluation requires another shift up, which is handled by creating and evaluating a **meta-alpha** expression. Figure 8 shows the handler for **meta-alpha**. This application is handled as if the operator were an ordinary quasi-static abstraction, except that the arguments are not evaluated.

5.3 A reflective quasi-static abstraction example

As an example, Figure 9 shows how to define a family **rf/tp** ("remember the future, throw to the past") of special forms with state. Each **rf/tp** form has a local variable *stored-k* in which it remembers the continuation of its initial use. On each subsequent use, it remembers the current

```
(common-define rf/tp
  (quasi-static (stored-k)
    (alpha (e r k p d)
      (let ([s-k stored-k])
        (begin
          (set! stored-k k)
          (p d (1st e) r s-k p d))))))

(common-define init-rf/tp
  (alpha (e r k p d)
    (let ([stored-k k])
      (begin
        (r (1st e)
           (install '(resolve (stored-k) rf/tp) ip bd))
        (p d (2nd e) r k p d)))))

(common-define rf-ex-setup
  (lambda (x)
    (+ 5 (init-rf/tp toss x))))

(common-define rf-ex1
  (lambda (y)
    (list (toss 7) 6)))
```

Figure 9: An example of a reifying quasi-static abstraction

```
0-10> (common-define toss '*)
0-10:  toss
0-11> (openloop 'qrad r1 ip qrad)
qrad-0:  "Refci"
qrad-1>

qrad-5> (rf-ex-setup 9)
qrad-5:  14
qrad-6> toss
qrad-6:  (reflective-q-s-abstraction
           (e r k p d)
           (let ([s-k stored-k])
             (begin (set!  stored-k k)
                    (p d (1st e) r s-k p d)))
           (environment .  #<procedure>))
qrad-7>

qrad-8> (rf-ex1 88)
qrad-5:  12
qrad-6> (+ 76 (toss 9))
qrad-8:  (9 6)
qrad-9> (+ 12 (toss 33))
qrad-6:  109
qrad-7>
```

Figure 10: Using an *rf/tp* form

continuation but continues the computation with the previously remembered continuation from *stored-k*.

We create the quasi-static reifier **rf/tp** with a quasi-static variable *stored-k* as a template. To create a specific form we can resolve the quasi-static variable of **rf/tp** with **init-rf/tp**. This form takes two arguments: the name of the form to be created and a value with which to continue the current computation. It establishes a binding for *stored-k*, resolves **rf/tp** in this scope, and stores the result in the current environment before continuing the computation. Figure 10 shows a transcript demonstrating the use of an **rf/tp** form. We omit some irrelevant evaluations. To use the form we open a new level with a dispatch[1] extended to handle quasi-static abstractions. The repetition of the numbered prompts helps us to see that in fact we are returning to a previous continuation each time the form **toss** is invoked.

[1] **qrad**, for quasi-static, resolve, apply dispatch.

6 Comparison with related work and conclusion

Much of the relationship with related work has already been mentioned. The standard procedural reflective systems, 3LISP, Brown, and Blond, permit the user to extend the language by defining new special forms. They do not allow modification of variable lookup, closure creation, and application. Thus they do not provide the full support necessary to implement extensions such as quasi-static abstractions.

An object-oriented approach such as 3-KRS [10] can provide a suitable basis for such extensions by allowing redefinition of the standard method lookup. Our work shows what is necessary to give a procedural reflective system the same flexibility.

A goal of reflection is to provide a general language extension method. Our experience in implementing quasi-static abstractions shows that this goal is not achieved with the design of the standard systems. Sufficient thought must be given to system design and implementation to take into account the kinds of extensions the user might want to make. The system must be open and flexible and must provide support for various extension models. It should, for example, provide for the creation and use of new kinds of applicable objects not provided in the system itself.

The design of the Refci system with its first-class extensible interpreters addresses these needs. It provides enough power not only to extend the language but also to modify its basic features. This provides the flexibility to accommodate unanticipated extension models.

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

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

[4] 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.

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

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

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

[8] Simmons, J., Jefferson, S., and Friedman, D. Language Extension via First-class Interpreters. Indiana University Computer Science Department Technical Report #362.

[9] Lee, S., and Friedman, D. Quasi-static abstractions: a device for connecting variables across multiple lexical scopes. To appear in the twentieth annual ACM Symposium on the Principles of Programming Languages, 1993.

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

[11] 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).

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

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

[14] 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.