Language Extension via First-class Interpreters

Indiana University Computer Science Department
Technical Report #362

John Wiseman Simmons II*
Stanley Jefferson†
Daniel P. Friedman‡

September 29, 1992

Abstract

Refci is an extensible reflective language based on the reflective tower model. The Refci interpreter procedures are reifiable, first-class objects, and user programs can directly modify the interpreter by extending them. This allows user code to be run directly as part of, and at the level of, the interpreter. By installing a modified or extended interpreter, user programs can modularly extend the language and implement debugging aids. We present the extensible first-class interpreter and give examples of its use. Examples include stepping, breakpoints, and interrupts. We show how the reflective tower, modeled by the metacontinuation, maintains the proper level of interpretation when running an interpreter containing both system code and user code.

1 Introduction

There are various methods, mostly *ad hoc*, to allow the user to extend and modify the semantics of a language. Some machine level languages allow the user to perform unrestricted, undisciplined (and dangerous) direct modification of running code. Some programming languages, such as Lisp, allow the user to extend the language by defining new special forms. Dialects of Lisp frequently provide the user with *ad hoc* methods to change the mechanism of evaluation and procedure application. In Symbolics Zeta Lisp [14], for example, these facilities are provided by evalhook and applyhook. These methods, however, do not permit the user to gain access to the full state of the computation, represented by the current expression, environment, and continuation.

For an interpreted language, the *reflective tower* model provides the user with a unified way to access the current state of the computation and extend the interpreter. *Reifying procedures* can be used to access the state and to define new special forms. They do not, however, permit the user to modify the existing semantics nor the process of interpretation itself.

*Supported by grants NSF DCR 85-21497 and NSF CCR 89-01919. Email address: *simmonsj@cs.indiana.edu*

†Supported by grant NSF CCR 89-01919. Current address: Hewlett-Packard, 3500 Deer Creek Road, Building 26U, Palo Alto, CA 94304-1392. Email address: *stan@hplstj.hpl.hp.com*

‡Supported by grants NSF CCR 87-02117 and NSF CCR 90-00597. Email address: *dfried@cs.indiana.edu*

In this paper we show how to provide the user with the ability to modify and extend the interpreter in a safe, controlled, coherent, and modular manner. We present Refci (reflective extension by first-class interpreters), an extensible reflective language based on the metacontinuation model. In Refci, the interpreter is a first-class object, obtainable by the user. It is represented as a pair of procedures, a *prelim* and a *dispatch*. The dispatch procedure is the main part of the interpreter and dispatches on the syntactic type of the interpreter's expression argument. The prelim allows the addition of actions to be performed during interpretation before the dispatch takes place.

The extensible first-class interpreter we present provides a unified, coherent, and general language extension method, which includes the capabilities of reflection and of the Lisp hooks, as well as of Bawden's Stepper [1]. It allows the user to access, extend, and modify the state of the interpreter with a single mechanism.

Once modified by the user, the interpreter contains both system code and user code. The system code is run directly, while the user code must be interpreted by the system. We thus face the problem of getting these two types of code to run at the same level. We shall see how the reflective tower enables us to maintain coherence of levels.

Section 2 discusses previous work in reflection and this work's relation with it. Section 3 gives a quick overview of the Refci language and the extensible interpreter. Section 4 illustrates some simple applications. Section 5 explains the mechanism that allows the user to add code to and modify the system. Section 6 gives two more advanced applications. Section 7 establishes the coherence of the model with respect to levels of interpretation. Section 8 suggests directions for future work. Section 9 states conclusions. Appendix A gives a denotational semantics for Refci.

2 Relation with other work

Computational reflection allows user code to access and manipulate the state of its own computation. That is, it has access to the components of the system running it. For a Lisp-type language, the current expression, environment, and continuation represent the state of the computation. These can be viewed as arguments of the Lisp interpreter.[1] Thus, we can view reflection as allowing the user to write code that directly manipulates the data structures of the interpreter. Since this is exactly what interpreter code does, we are allowing the user to write code to be run as part of, and at the level of, the interpreter.

Lisp has sufficient expressive power to express its own evaluator. Such an interpreter, expressed in the language it interprets, is a *metacircular interpreter*. We can imagine using the implementation of this interpreter to interpret its own source code, which in turn interprets the user's code. This idea can be applied again, yielding a third level of interpretation, and so on for as many levels as desired. In the limit, we can envision an infinite *reflective tower* of interpreters, each interpreting the code of the one below, and the one at the bottom interpreting the user's code.

The seminal work on computational reflection is Brian Smith's thesis [6, 13]. Smith proposes a semantically rationalized dialect of Lisp as a basis for a reflective implementation. The result of this development is the reflective dialect 3-Lisp. The key observation permitting the finite implementation of the infinite tower is that all but the bottom few levels of the tower will always be doing identical work: interpreting the default interpreter code. Only the finite number of levels

[1] They can also be viewed as registers in a machine, as in Bawden's Stepper. The interpreter view is more suitable for our purposes.

interpreting user-supplied code need actually be running. The rest can be virtual. The infinite tower is modeled by use of a *metacontinuation*, an infinite stack of virtual levels. Each level of the tower is represented by its continuation: where it would begin if it had to start interpreting code.

$$\text{MetaCont} = \text{Cont} \times \text{MetaCont}$$

In the 3-Lisp system, the user writes code to be run at the level of the interpreter as a *reifier*, or reifying procedure. Application of a reifier causes a virtual level of interpretation to become actual. The current expression, environment, and continuation are made explicit and bound to the reifier's arguments. Interpretation of the reifier's body proceeds, with the continuation of the next level above removed from the metacontinuation and installed as current.

Thus the application of a reifier causes a shift up. The extra level of explicit interpretation could persist indefinitely. But there is no need. As soon as the extra level begins interpreting default interpreter code rather than user code, it can become virtual again. The system shifts down by pushing the extra level's continuation back onto the metacontinuation, and the interpreter again begins directly interpreting user code.

The Brown [8, 15] and Blond [4, 5, 11, 12] languages are also based on the metacontinuation model. They are designed to allow a clean description in denotational semantics.

Our work retains the metacontinuation representation of the reflective tower, but alters the way the user accesses the tower's levels. Our intent is to increase the user's power to extend and modify the language, while at the same time making sure that all user modifications are run at the proper level.

Reifiers are often regarded as adding a new form to the language by "adding a line to the interpreter," that is, as providing code to be executed at the level of, and as part of, the interpreter. However, reifiers allow only the addition of new special forms to the language. They do not permit the user to modify directly the existing behavior of the interpreter. Also, reifiers do not permit the user to add debugging facilities such as stepping and variable monitoring.[2]

We instead give the user access to the system by making the interpreter a first-class object. The user can access the interpreter and install a modified version in the system. All access to the tower can be mediated through these modified interpreters.

Pattie Maes [10] presents an object-oriented reflective language 3-KRS. In 3-KRS, programs as well as data are represented as objects. The computational information about an object is stored in its meta-object. This system uses a metacircular interpreter in direct style. The interpreter is divided among the default meta-objects. Supplying program objects with non-standard meta-objects causes implicit reflection and allows user code to be employed in their interpretation. Different local modifications are possible for different objects. Explicit reflection is achieved by having an object's code mention its meta-object.

The 3-KRS interpreter has an underlying driver that is not made explicit. Our first-class interpreter makes the entire system computation explicit and available to the user.

[2] Brown does implement variable monitoring, but not with reifiers. It is done through the environment and depends on the implementation of environments as functions.

3 The Refci language and the extensible interpreter

In this section we give a quick overview of our language and the extensible interpreter. Our language, called Refci, is derived from Blond [4, 5, 11, 12]. It is implemented in Scheme [2, 7] and has Scheme-like syntax. There are only three forms of expression: constant, variable, and application (with an arbitrary number of arguments). Primitive procedures are included as *subrs*. The basic special forms, such as **lambda**, **if**, and **let**, are represented as *fsubrs*, procedures that do not evaluate their arguments. Environments, continuations, prelims, and dispatches, as well as subrs, fsubrs, user procedures, and reifiers, are applicable objects.

The purpose of the extensible interpreter is to allow the user to modify the behavior of the system at run time. In most implementations, the interpreter is hard-wired into the system, represented as code in the implementation language. To make it accessible to the user, we represent it as two procedures, a *prelim* and a *dispatch*. The dispatch procedure is the main part of the interpreter and dispatches on the syntactic type of the interpreter's expression argument. It can be extended by adding implementations of special forms or modified to change the basic process of interpretation. The prelim represents actions to be performed before the dispatch. Its type is the same as that of the dispatch:

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

We curry the types for clarity. A dispatch takes as arguments the current expression, environment, and continuation, a metacontinuation representing the tower, and a prelim and a dispatch representing the current interpreter. The default dispatch contains lines for constants, identifiers, combinations (applications and special forms), and bad form errors. A prelim is a dispatch transformer. The user extends the interpreter by writing a prelim. If it represents a preliminary action it is sequenced with the existing prelim. If it represents a new special form or a modification of the dispatch, it is used to transform the existing dispatch to one with the desired behavior.

This explains why we choose to represent the interpreter as two procedures instead of one. Preliminary actions are supposed to be performed before any dispatch lines. If the interpreter were represented as a single procedure, a dispatch, prelims and dispatch lines could become mixed as the interpreter is extended. Using two procedures allows two insertion points: the dispatch for new lines and the prelim for new preliminary actions.

The prelim and dispatch procedures are to be applicable objects. They are not of the same type as user procedures. To extend the interpreter, the user creates a closure and applies the wrapping primitive `make-prelim` to it to obtain a prelim. We can best illustrate the process by examples.

4 Simple applications

In this section we illustrate the use of the extensible interpreter mechanism to add simple features to the language. These are exit, stepping, and variable monitoring. The examples included serve two purposes. They illustrate the different methods available to modify the interpreter. More significantly, they show how the extensible interpreter encompasses the power of reifiers and also adds features that reifiers cannot.

```
(common-define exit
  (make-reifier
    (lambda (e r k p d)
      (if (null? e)
          "Exit wants an argument."
          (p d (car e) r (lambda (x) x) p d)))))
```

Figure 1: The form *exit* as a reifier

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

Figure 2: The form *exit* as an interpreter line

To use this facility, the user must have procedures available to construct, access, and modify interpreters. Calling reify-new-prelim returns a copy of the default prelim, which does nothing but call the dispatch. Calling reify-new-dispatch returns the default dispatch. The procedures extend-prelim and extend-dispatch, compose a new prelim with an existing prelim or dispatch, respectively.

There are two mechanisms available to install a new interpreter during an evaluation. The prelim and dispatch can be applied directly to reified expression, environment, continuation, prelim, and dispatch arguments. This has the effect of pushing a new level onto the metacontinuation, thus running the interpreter one level lower than the current level and simulating an extra level of interpretation. Alternatively, the install primitive can be used to install a new interpreter at the current level for the evaluation of expression *e*:

(install *e new-prelim new-dispatch*)

4.1 Exit

The first example shows that the extensible interpreter includes the power of reifiers: adding a special form to the language. The user may simply wish to abort the current computation and exit to the next higher level of the tower. The expression

(*exit e*)

evaluates expression *e* at the current level and returns its value to the next level up in the tower.

We illustrate adding the *exit* form to the language as a reifier (see Figure 1) and as an interpreter line (see Figure 2). Each evaluates the argument[3] of *exit* with an identity continuation. When this

```
(common-define step-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (begin
        (pretty-print e)
        (display "Step: ")
        (case (read-at-prompt)
          [(abort) 'aborted]
          [else (dd e r k p d)])))))
```

Figure 3: Stepping the computation

continuation is invoked, it terminates the current level and causes the computation to continue one level higher in the tower.

The interpreter line version of *exit* must take responsibility for testing whether the current expression is an *exit* expression, for handling it if it is, and for continuing the interpretation if it is not. The argument *dd* represents the dispatch to which the line is being added, and which will be used to continue the interpretation if the expression is not an *exit* expression. The argument *d* represents the entire extended dispatch. As an example,

```
0-2> (common-define ip (reify-new-prelim))
0-2: ip
0-3> (common-define id (reify-new-dispatch))
0-3: id
0-4> (install '(exit 30) ip (extend-dispatch exit-line id))
1-0: 30
1-1>
```

4.2 Stepping

Single-stepping the computation (see Figure 3) is possible with the extensible interpreter, but not with a reifier. We insert a new prelim that, at each call to the interpreter, prints the current expression (other information could also be printed) and a prompt, then waits for user response. Various actions could be taken based on the response. This simple example allows the user to abort the computation (and the current level) by typing **abort**. Any other response continues the computation. To illustrate,

```
0-2> (common-define foo 30)
0-2: foo
0-3> (install '(+ foo 5) ip (extend-dispatch step-prelim id))
(+ foo 5)
Step: c
```

[3] The keyword has already been removed from the expression argument of a reifier, but not from that of a prelim. This is why the reifier obtains its argument as the first element of e, while the prelim uses the second element.

```
(common-define id-line
  (make-prelim
    (lambda (dd e r k p d)
      (if (symbol? e)
         (if (monitored? e)
            (begin
              (display "Looking up ")
              (display e)
              (newline)
              (let ([v (r e)])
                (begin
                  (display "value is: ")
                  (display v)
                  (newline)
                  (k v))))
            (dd e r k p d))
         (dd e r k p d)))))
```

Figure 4: Monitoring variable lookup

```
+
Step: c
foo
Step: c
5
Step: c
0-3: 35
0-4>
```

4.3 Variable monitoring

A facility often used in debugging is monitoring accesses to variables. This fundamental modification of the system's behavior is impossible using reifiers. It is easy to implement with the extensible interpreter by replacing (shadowing) the line for identifiers (see Figure 4). The new line checks whether the identifier is currently being monitored. If so, it prints a message stating which variable is being looked up, does the lookup, and prints and returns the value to the waiting continuation.

```
0-5> (install '(+ 5 foo) ip (extend-dispatch id-line id))
0-5: 35
0-6> (monitor-id 'foo)
0-6: ()
0-7> (install '(+ 5 foo) ip (extend-dispatch id-line id))
Looking up foo
```

```
(define _make-closure
  (lambda (type par body env)
    (list type (length par)
      (lambda (lv k p d tau)
        (p d body (_extend_env par lv env) k p d tau)))))
```

Figure 5: Representing a closure

```
value is: 30
0-7: 35
0-8>
```

5 Adding user code to the interpreter

We have represented the interpreter by two procedures, with the promise that it will be extensible. We have made the prelim and dispatch first-class, reifiable objects by passing them as arguments to themselves. Now we show how the user writes code to extend the interpreter. The mechanism we provide will use the tower (metacontinuation) to keep all code running at the proper level.

Figure 5 shows how a closure is represented as a Scheme procedure that evaluates the closure's body in the appropriate environment. The closure's arity is included for error checking. Thus the representation of a closure is a Scheme procedure of type

$$\mathrm{ExpV}^* \times \mathrm{Cont} \times \mathrm{Prelim} \times \mathrm{Dispatch} \times \mathrm{MetaCont} \rightarrow \mathrm{Ans}$$

A quick comparison with Section 3 shows that this is not the proper type for a prelim. Thus we must provide the user with a primitive that takes a closure and returns a prelim. Since the user code is to be interpreted, there must be a shift up to obtain the correct interpreter from the tower. The action of shifting up is provided by the primitive and implicitly becomes part of the extended interpreter.

Figure 6 shows the primitive wrapper *_make-prelim*, which changes a closure into a prelim. It expects (in the list l) one user-supplied argument, an expression. It evaluates this expression to obtain c, the closure to be wrapped. If c has the correct arity for its intended use, the primitive returns a prelim that reifies its arguments, shifts up, and applies c to run the user code.

We have seen how the invocation of a user-defined prelim causes a shift up. When does the system shift back down? The system must shift down whenever the user's code invokes a default prelim or dispatch. There is really no need to shift down when invoking a user-written prelim. However, the wrapping for such a prelim always causes a shift up. Thus there must always be a shift down when user code invokes any prelim or dispatch.

The prelim and dispatch procedures, which represent the interpreter, now become first-class, reifiable objects. They are explicitly maintained as arguments to themselves and can be obtained by the user via reification. Thus reifiers now take five arguments instead of three: the current expression, environment, continuation, prelim and dispatch. The reified prelim and dispatch are applicable objects and can be directly applied by the user with reified expression, environment, continuation, prelim, and dispatch as arguments. Such an application causes a new level to be

```
(define _make-prelim
  (lambda (l r k p d tau)
    (p d (car l) r
      (lambda (c tau)
        (if (and (_closure? c) (= (_appl-obj→arity c) 6))
            (continue
              k
              (_pre-up
                (lambda (dd e r k p d tau)
                  ((_appl-obj→proc c)
                   (list (_dis-up dd) (_exp-up e) (_env-up r) (_cont-up k)
                     (_pre-up p) (_dis-up d))
                   (_top-cont tau)
                   (_top-prelim tau)
                   (_top-dispatch tau)
                   (_meta-pop '_prelim_applier tau))))
              tau)
            (_terminate-level
              (_wrong '_make-prelim "bad function or wrong arity" c) tau)))
      p d tau)))
```

Figure 6: Creating a prelim from a closure

spawned, the old level to be pushed onto the metacontinuation, and the reified prelim and dispatch to be installed as the current interpreter at the new level. This performs the function of the Brown and Blond primitive meaning.

Reifiers themselves are no longer necessary, since their function of adding lines to the interpreter is now performed directly. In Section 7 we shall see how the extensible interpreter facility enables us to move up the tower without using reifiers.

We retain reifiers in the language for convenience and efficiency. Reifying procedures are created from closures by the primitive make-reifier. They are analogous to the Brown reifiers in that they are closed in the environment current at the time of their creation. They do not use the environment from the level above at the time of either their creation or application, as do the **gamma** and **delta** reifiers of Blond.

There is one difference in the behavior obtained by using reifying procedures and by adding lines directly to the interpreter. The addition of a line modifies only the interpreter in which it is installed and any interpreters generated from the modified interpreter. It thus affects only the finite number of levels of the tower at which these interpreters may later be installed. Reifiers, on the other hand, do not directly modify any interpreter. If they are placed in the common environment, they can be called from any level of the tower. Reifiers have the ability to cause global additions to all levels of the tower; the extensible interpreter, as currently implemented, does not.

6 Applications

In this section we show how to use the extensible interpreter mechanism to add some more advanced features to the language. These are break and interrupt handling.

6.1 Break

A break facility allows the user to set breakpoints in the code. When the system reaches this point, it stops and opens a break loop, allowing the user to examine the current state, evaluate the current expression or step its computation, modify variables, return an alternate value from a subexpression, or abort the computation.

Implementing *break* requires two interpreter extensions. A line must be added to recognize and interpret the *break* special form. Figure 7 shows the *break* line, which opens a new top-level loop with an interpreter extended with lines to perform the various break functions. Figure 8 shows some of the internal lines. An example of *break*:

```
0-3> (common-define foo 30)
0-3: foo
0-4> (install '(* 3 (break (+ 5 foo))) ip (extend-dispatch break-line id))
break-0: "Refci"
break-1> e
break-1: (+ 5 foo)
break-2> (eval-exp)
35
break-2: 35
break-3> (go)
```

```
(common-define break-line
  (make-prelim
    (lambda (dd e r k p d)
      (if (is-type? 'break e)
         (let ([return-line (make-return-line k)]
               [break-e (cadr e)])
           (let ([eval-line (make-eval-line break-e)]
                 [go-line (make-go-line break-e k)]
                 [ok-line (make-ok-line break-e k)]
                 [step-line (make-step-line break-e)])
             (let ([break-d
                     (extend-dispatch eval-line
                       (extend-dispatch go-line
                         (extend-dispatch ok-line
                           (extend-dispatch return-line
                             (extend-dispatch abort-line
                               (extend-dispatch step-line d))))))])
               (openloop
                 'break
                 (extend-reified-environment
                   '(e r k p d) (list break-e r k p d) r)
                 p break-d))))
         (dd e r k p d)))))
```

Figure 7: The *break* line

```
(common-define make-eval-line
  (lambda (break-e)
    (make-prelim
      (lambda (dd e r k p d)
        (if (is-type? 'eval-exp e)
            (p d break-e r
              (lambda (v)
                (begin
                  (display v) (newline)
                  (k v)))
              p d)
            (dd e r k p d))))))

(common-define make-ok-line
  (lambda (break-e break-k)
    (make-prelim
      (lambda (dd e r k p d)
        (if (is-type? 'ok e)
            (p d break-e r break-k p d)
            (dd e r k p d))))))

(common-define abort-line
  (make-prelim
    (lambda (dd e r k p d)
      (if (is-type? 'abort e)
          (p d (cadr e) r (lambda (v) v) p d)
          (dd e r k p d)))))

(common-define make-step-line
  (lambda (break-e)
    (make-prelim
      (lambda (dd e r k p d)
        (if (is-type? 'step e)
            (step-prelim d break-e r k step-prelim d)
            (dd e r k p d))))))
```

Figure 8: Some of the internal *break* lines

```
(common-define interr-prelim
  (make-prelim
    (lambda (dd e r k p d)
      (if (is-interrupt-pending?)
          (handle-interrupts dd e r k p d)
          (dd e r k p d)))))

(common-define handle-interrupts
  (lambda (dd e r k p d)
    (begin
      (display "An interrupt occurred ")
      (display (global-interrupt-pending?)) (newline)
      (global-set-interrupt-pending! #f)
      (break-line dd (list 'break e) r k p d))))
```

Figure 9: Constructing an interruptible interpreter

```
35
0-4: 105
0-5>
```

6.2 Interrupts

Interrupt handling can be added to the interpreter by inserting a new prelim procedure and using custom-designed input procedures that buffer the input, check for interrupt characters, and store any interrupt character detected in a global flag. Three new primitives allow the user access to the interrupt status. The first, is-interrupt-pending?, is called by the interrupt prelim at the beginning of each evaluation. It reads into the buffer any characters that are waiting and returns the status of the interrupt flag. The others, global-interrupt-pending? and *set-global-interrupt-pending!*, access and set, respectively, the status of the flag. Figure 9 shows the code to construct an interruptible interpreter. If an interrupt is detected, the interrupt handler is called in tail position. The one shown opens a break loop to allow the user to inspect the state of the computation. The user can continue the computation by passing a value to the continuation with which the loop was invoked.

In this example, the $ is used as the interrupt character to request a break. Other interrupt characters can be used with different meanings.

```
0-5> (install '(fact 5) (extend-prelim interr-prelim ip) id)
$
An interrupt occurred $
break-0: "Refci"
break-1> e
break-1: sub1
break-2> (ok)
0-5: 120
0-6>
```

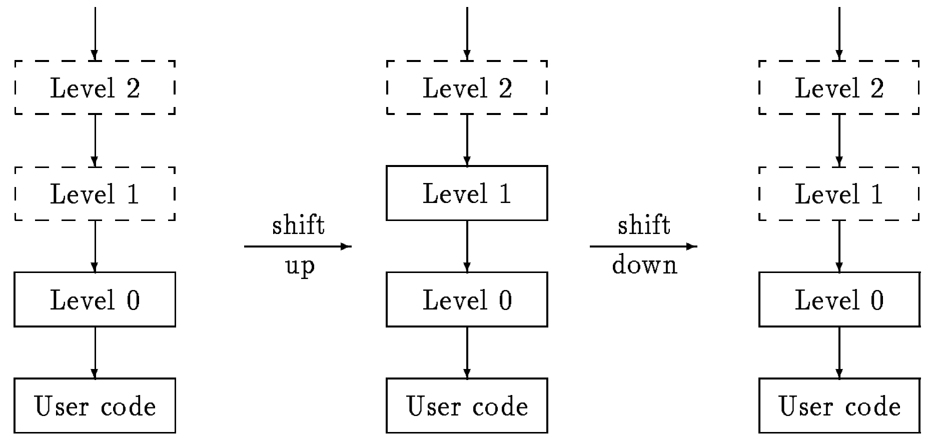

Figure 10: Shifting up for non-default and down for default code

7 Maintaining the proper level

So far we have taken for granted the crucial role of the metacontinuation in maintaining the correctness of the levels in the computation. It is now time to consider this role in detail.

The default interpreter, the one with which the system is started, contains only system code. If the user obtains a reified copy of this default interpreter and modifies it by inserting a new prelim or adding a new line, the resulting interpreter will contain both system and user code. As the computation progresses, control shifts back and forth between the two types of code. Since they are both interpreter code, they must be run on the same level. However, the default code, written in Scheme, must be run directly, and the Refci code requires an extra level of interpretation. So it appears impossible to get them to run on the same level.

Here the tower, as represented by the metacontinuation, comes into play. We can use tower levels to balance needed actual levels of interpretation, ensuring that the levels are always correct.

The level shifts are depicted in Figure 10. Suppose that the default interpreter, running at level 0, is replaced by an extended interpreter, also at level 0, interpreting some user code. At some point in the computation, a non-default prelim is encountered in the interpreter. The Refci code it represents must be interpreted. The non-default prelim must pass the appropriate list of arguments representing the current level to its internal closure. It shifts up in the tower by popping the metacontinuation, obtaining the continuation, prelim, dispatch, and metacontinuation for level 1. The interpreter from level 1 interprets the non-default code, which is now effectively running at level 0, interpreting the original user code. A "virtual" level has been removed from the metacontinuation and an actual level of interpretation added, a net change of 0. Thus the levels are correct. If the interpreter at level 1 happens to contain a non-default procedure, another shift up occurs, and the interpreter from level 2 is used. All levels in the tower are initially running the default interpreter, and only a finite number of them can be changed after a finite amount of time. So eventually this process must stop and we must reach a default interpreter.

What happens when we apply a reified prelim or dispatch in the non-default prelim after we have shifted up to interpret non-default code? We must shift back down to apply the reified procedure. In the example above, we would push the state of level 1 back onto the metacontinuation and apply the default procedure directly at level 0. The level changes are exactly opposite to those just mentioned. We gain one virtual level and lose one actual level, again a net change of 0.

We have seen how we can use the infinite tower represented by the metacontinuation as a convenient bookkeeping device. It allows us to keep track of all the levels and make sure that all interpretation is done at the correct level. We have also seen how the use of non-default code in the extended interpreter can cause a shift up the tower. Thus extensible interpreters have the same capabilities as reifiers, making reifiers themselves unnecessary.

On the other hand, it is more efficient to use reifiers to add special forms. Adding a special form directly to the dispatch requires additional work, including a shift up and back down, to be done each time the interpreter is applied. This work is unnecessary for a reifier, since the shifts occur only when the reifier is applied. Thus extensions to the dispatch should be used primarily to override the default interpretation of identifiers and applications.

8 Future work

One interesting possibility for future work is the topic of extended continuations. Currently, implementation continuations take a value and a metacontinuation, while user continuations take a value. The interpreter that interprets the continuation code is determined when the continuation is created, not when it is applied. It would be interesting to allow continuations to take an interpreter as an additional argument. This would allow the extent of an interpreter's use to extend beyond the evaluation of the expression for which it was installed, and continue through the rest of the computation.

The organization of the interpreter suggests that the system would benefit from an object-oriented approach. The interpreter should be viewed as an object taking messages to modify and extend its methods.

9 Conclusions

Procedural reflection can be implemented by making the interpreter a first-class object. The user can directly extend such an interpreter by inserting user-written code. We have derived an extensible interpreter from a standard continuation-passing style interpreter. The first-class interpreter enables us to shift levels without the use of reifiers and to add features that reifiers cannot implement. The reflective system maintains the coherence of levels during an extended interpreter's computation. Applications of the extensible interpreter include the implementation of debugging facilities such as stepping, break and interrupts.

References

[1] Bawden, A. Reification without evaluation. *Proceedings of the ACM Conference on LISP and Functional Programming* (1988).

[2] 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.

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

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

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

[6] 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.

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

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

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

[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] Symbolics Lisp Manual, Vol. 4: *Program Development Utilities.* Symbolics, Inc. (August 1986).

[15] 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.