Construction of the Reflective Tower Based on Open Implementations

Kris De Volder[1] , Patrick Steyaert[2]

--- DRAFT (1.1) ---

Abstract

It is our opinion that the traditional view on reflection, the notion of towers of interpreters interpreting each other, is not sufficiently detailed to give a thorough understanding of reflection. Expressions such as "somehow the levels must be connected" and "adding lines to the interpreter above" are typical when talking about towers of meta-circular interpreters. This alone gives an indication that the model is not detailed enough, lacking a way to formalise the relation between levels of the tower. The connection remains a magical ingredient in the recipe to cook up a reflective system. This is the main reason why reflection has hitherto remained covered in a mystical veil.

We present an alternative view on reflection. Rather than being based on meta-circular interpreters, this model is based on open implementations. An open implementation hides the implementation details of the interpreter, but shows how the interpreter can be extended/adapted. In this approach reflection is obtained by explicitly generating the limit of an infinitely ascending chain of open implemented interpreters through a fix-point operation.

We argue that the connection between interpreters in the traditional view is ad-hoc and counter-intuitive. The open implementation point of view yields a notion of reflection which is highly similar to the traditional view, but improves upon the ad-hoc way of relating interpreters at different levels. In a tower of open implementations the connection between levels is established in a natural way through the parameters for the open-implementation which are provided by the implementing level above.

1 Motivation

Every reflective system needs an accessible, causally connected self-representation. As every representation defines a certain terminology to talk about the entities it represents, so does this self-representation define a terminology for the system to talk about itself. The self-representation determines the system’s aspects that can be reasoned about and modified by the system. As is true for any representation the self-representation can not be “complete”, i.e. any representation will always ignore certain aspects of the system it represent. For reflective systems this is known as the “theory relativity” of reflective systems [Maes87].

For procedurally reflective languages it is said that the procedural code in the meta-circular processor serves as the “theory” or causally connected self-representation [Smith84]. It is our opinion that this is a misleading, or even wrong, statement. And that exactly this statement hampers our true understanding of reflective systems. In this paper we will discard with meta-circular processors as self-representations. Moreover, since the meta-circular processors are used in the tower model, we will also discard the notion of towers of meta-circular processors. We will not discard with the notion of tower-architectures! Only with towers of meta-circular processors.

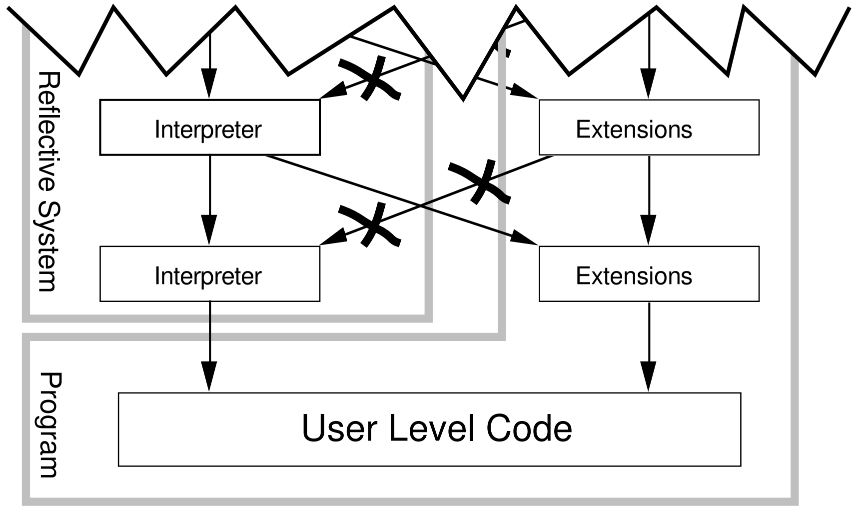
One of the motivations for our work is the demystification of the magical "link between levels" ingredient. As stated before, we believe that the traditional model is not sufficiently detailed in this respect.

Another and perhaps more important consideration is the fact that the traditional view defies the notion of theory

1 Programming Technology Lab; Computer Science Department; Vrije Universiteit Brussel; Pleinlaan 2, B-1050 Brussel, Belgium; email: kdvolder@vnet3.vub.ac.be
2 Programming Technology Lab; Computer Science Department; Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium; email: prsteyae@vnet3.vub.ac.be

relativity. When using the procedural code in the meta-circular processor for a self representation it is tempting to think that if one can change the code of the meta-circular interpreter in any way one likes, one must be able to do just about anything. This of course is not true because a meta-circular evaluator alone does not define a programming language. An external processor is needed to process the meta-circular interpreter.

If the procedural code of the meta-circular processor is to serve as a causally connected self-representation then any modification to this code must not only affect the interpretation of user programs but also the interpretation of the meta-circular processor's code itself. In practice—in any existing implementation—this is not the case. Thus the meta-circular code does not truly serve as a self representation. The following figure tries to illustrate this.



This picture represents a traditional infinite tower of meta-circular interpreters. The arrows indicate the "… plays a part in the interpretation of …" relationship. The reflective system is a tower of meta-circular interpreters. This tower is used to interpret a program. The program usually contains some normal non reflective "User Level" code but also some reflective code that will be installed as part of the interpreter (shown in the drawing as "Extensions"). The traditional model is misleading because it gives the impression that the extended interpreter is used to interpret itself meta-circularly. This is not true, the extended interpreter is only used to interpret the extensions, while the "core" of the interpreter is not affected by the extensions. That is why we have crossed out the arrows leading from "extensions" to "interpreter".

In any existing implementation, extensions to the language do not affect the interpretation of the "core interpreter". The extensions only affect evaluation of a) “User Level Code” and b) the code implementing the extensions themselves. They *do not* affect the core-interpreter. In many systems this is so because the "core interpreter" is explicitly written in a subset of the language that can not be altered by reflective programming (e.g. a variant of Scheme with reifier-procedures, whereby the implementation does not make use of reifiers, nor is it possible for reifiers to override the pre-defined special forms). In other implementations the core interpreter is written in a part of the language that can be changed by reflective programming (e.g. a variant of Scheme whereby the pre-defined special forms can be redefined), but even then the scope of the changes will only include the extension's implementation and the user level code, but never the actual interpreter itself.

The traditional model is deceiving because it does not distinguish the extensions (added by reflective programming) and the core of the interpreter from each other. Our approach attempts to remedy this by dividing the interpreter into a fixed and a parameterised part.

In this paper we will start with building an open implemented interpreter. This open implementation will be written meta-circularly, meaning that it can be evaluated with some “basic” evaluator obtained from that open implementation itself. Then we will experiment a little with finite[3] literal towers of open implementations. The finite tower experiment serves as a stepping stone towards reflection, providing an easy way to experiment with towers of open implementations. After playing around with finite towers for a while we will introduce “real” reflection characterised by a fix-point equation and show the relation between this equation and infinite towers.

2 The Open Implemented interpreter

Our approach is an attempt to refine the traditional model. We will represent a level of the tower by an open-implemented interpreter, explicitly representing the "fixed-core" of the system as a separate entity.

We will use the following simple example language (ASEL), which is a subset of Scheme.

```
<exp>          =  <var> | <constant> | <lambda> | <if> | <definition> |
                  <assignment> | <application>
<var>          =  <scheme-symbol>
<constant>     =  <scheme-literal> | <quoted>
<quoted>       =  '<scheme-value>
<lambda>       =  (lambda <formals> <sequence>)
<sequence>     =  <exp>+
<formals>      =  () | <var> | (<var>+ [. <formals>])
<if>           =  (if <exp> <exp> <exp>)
<definition>   =  (define <var> <exp>)
<assignment>   =  (set! <var> exp>)
<application>  =  (<exp> <exp>*)
```

An open implementation is in essence nothing more than a parameterised interpreter. The parameterised interpreter will take the form of a function we will name *meta***.** Applying *meta* to a parameter will return an evaluator based on that parameter. Thus we can obtain a range of evaluators, by applying *meta* to a variety of parameters. The fixed core is explicitly represented by *meta*.

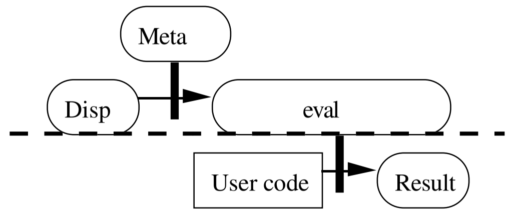
The way we write *meta*, the choice of parameterisation, establishes beforehand in what way we can adapt the evaluator. In the system we will implement here as an example, the goal is to be able to extend the evaluator so that it can handle new kinds of syntax structured as follows:

```
(<syntactic-keyword> <arg>*)
```

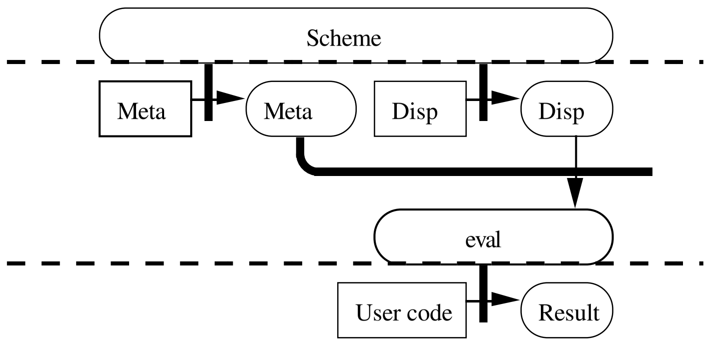
For example we could extend the evaluator with *cond*, *let*, *let**, … expressions. The parameter that is passed to *meta* will take the form of an assoc-list which associates an appropriate evaluation procedure with a *<syntactic-keyword>*. We will also provide a standard parameter, yielding the basic evaluator which handles the vanilla version of ASEL. The definition of *meta* will be written meta-circularly, which in our case means that it is implemented in vanilla ASEL and thus can be evaluated with the basic-evaluator.

The following figure gives a schematic view of what our open implementation looks like.

[3] This approach is inspired by the work of Jefferson and Friedman [Jefferson&Friedman92]. They introduce reflection through explicitly constructing finite towers meta-circular interpreters that are interpreting each other literally. This gives poor performance, but it does give a very clear, simple and understandable account of procedural reflection. One can observe the behaviour of a finite tower and “extrapolate” upon this to understand how an infinite (procedurally) reflective tower behaves.



The convention is that rectangular objects represent syntactic entities, i.e. pieces of source code. Round objects represent functions. The application of a function is represented by a thick black line, an arrow points through the black line, pointing from the argument to the result of the application. This diagram shows the open implementation as a function called *meta*, that is applied to a dispatcher. The result of this application is a function called *eval*, this is the evaluator. The evaluator is then used to evaluate user code. One last remark: the parameter to our meta function is called *disp*, in our system this is an association list, which associates a syntactic keyword with an evaluation function. Thus, strictly spoken, *disp* is not a function but to keep the figures simpler we will treat it as such. Regarding *disp* as a function does not change anything in an essential way. Similarly *result* is also drawn as a function although it usually is something else like a number, a list, ….



The previous figure gives a more detailed view of the open implementation, this time also showing the open implementation's source code, and the dispatcher's source code. Both are evaluated by the underlying scheme's evaluation function, yielding the *meta* and *disp* functional objects.

We can view *meta* as establishing the *meta-theory*. It determines what aspects of the interpreter we can talk about, and how we must do this (i.e. what parameters must be supplied to *meta*). In this respect we can consider the argument(s) to *meta* as the *representation* for some evaluator. The meta-theory relates the representation to the evaluator it represents. Without a proper meta-theory, the representation is meaningless. This model explicitly exhibits *theory relativity*. The representation is not complete it only determines an interpreter in the context of some meta-theory: some of the interpreters aspects are contingent to this meta-theory.

The *representation* (the round Disp in the drawing) is a semantical object. We must denote this semantical object by some syntactical structure (the rectangular Disp in the drawing). An interpretation function will be needed to relate the syntactical *description* with the corresponding semantical *representation*. In this drawing that interpretation function is the Scheme evaluator. From here on we will use "*representation of E*" for referring to the semantical object that represents an evaluator *E* in the sense described above. Respectively we will use "*description of E*" for referring to the syntactical structure denoting the representation of *E*.

The overall structure of *meta* is displayed below. It takes a dispatch-table assoc-list as argument and returns an evaluator. The implementation of the evaluator, which is hidden inside the body of *meta*, is written in continuation passing style. All evaluation procedures like evaluate, basic-evaluate, evaluate-constant, … take 3 arguments *e*, *r* and *k*. These are respectively the expression to be evaluated, the current environment (= set of bindings of variables to values) and the current continuation. The evaluation procedures in the dispatcher take another extra argument: *evaluate*. This is the evaluator itself, passed as an argument to dispatcher procedures so that they may use it to evaluate sub-expressions.

```
(define meta
  (lambda (dispatch-table)

    (define evaluate
      (lambda (e r k)
        (if (pair? e)
           (find-pair (car e) dispatch-table
              (lambda (success-pair)
                ((cdr success-pair) evaluate e r k))
              (lambda ()
                (basic-evaluate e r k)))
           (basic-evaluate e r k))))

    (define basic-evaluate
      (lambda (e r k)
        ((if (constant? e)
           evaluate-constant
           (if (variable? e)
             evaluate-variable
             (if (if? e)
               evaluate-if
               (if (assignment? e)
                 evaluate-assignment
                 (if (definition? e)
                   evaluate-definition
                   (if (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         e r k)))

    (define evaluate-constant …)
    (define evaluate-variable …)
    …
    (define …)

    evaluate))
```

The main procedure, *evaluate*, checks the dispatch table for an appropriate evaluation procedure to call. If one is found, then it will be used to evaluate the expression. If the dispatcher does not contain a procedure for this type of expression, then *basic-evaluate* gets called. *Basic-eval* handles all "vanilla" ASEL expressions, it distinguishes between different types of expressions and dispatches to an appropriate evaluation procedure for that particular expression type.

Most of the sub-task procedures that *basic-eval* dispatches to are rather straightforward, so we won't explain all of them here. The full source code can be found in appendix A. Since some of the rest of this paper will involve dealing with complications that arise when procedures are moving up and down in the tower of interpreters, and inter level (in)compatibility of procedures, we will now take a look at the code of the evaluator involving procedure creation (evaluation of lambda expressions) and procedure calls.

```
(define evaluate-abstraction
  (lambda (e r k)
    (k (make-compound (formals-part e) (body-part e) r))))
```

Evaluation of a lambda expression (abstraction) is very straightforward: create a representation for a procedure by calling *make-compound*, and pass the result to the continuation.

```
(define make-compound
  (lambda (formals body r)
    (lambda (k . args)
```



```
      (evaluate-sequence body (extend r formals args) k))))
```

As can be seen from the definition of *make-compound*, procedures are represented by procedures. The representation procedure has an extra first argument. This extra argument is a continuation that will receive the result of the procedure-call. The remaining arguments are the "real" arguments. Application of a procedure represented in this way is written as follows.

```
(define apply-procedure
      (lambda (proc args k)
        (if (procedure? proc)
          (apply proc (cons k args))
          (wrong "operator is not a procedure" proc))))
```

The *evaluate-combination* procedure is the one that is called for evaluating procedure calls. It first evaluates the operator part (the expressions that yields the procedure to be called). Then it evaluates the operands one by one by calling the procedure *evaluate-operands*. Finally the procedure (*proc* = result from evaluating the operator part) is applied to the list of evaluated arguments.

```
(define evaluate-combination
      (lambda (e r k)
        (evaluate (operator-part e) r
           (lambda (proc)
             (evaluate-operands (operands-part e) r
                (lambda (args)
                  (apply-procedure proc args k)))))))
```

We also provide a read-eval-print loop mechanism, so that we can type in expressions to be evaluated and see the result printed on the screen. A read-eval-print loop can be started by calling the function *openloop* passing the evaluator as an argument.

```
(define openloop
  (lambda (evaluate read-prompt write-prompt)
    (display read-prompt)
    (evaluate (read) global-env
      (lambda (v)
        (display write-prompt)
        (if (eq? v (void))
          "Nothing is displayed"
          (write v))
        (newline)
        (openloop evaluate read-prompt write-prompt)))))
```

Now we have everything we need to start a session, we can open a read-eval-print loop on a basic evaluator, or variation of the evaluator we create by applying meta to a parameter. The following is an example of how one might create an extended evaluator that understands a special *exit* expression. When an *exit* expression is evaluated, it causes the evaluator to terminate promptly, discarding all pending computations. The example shows the evaluation of some simple expressions, an invocation of the *exit* construct ends the session. Things typed in **bold** where typed in by the user. Things in normal font where responses or prompts printed by the read-eval-print loop or by the underlying scheme system.

```
scheme>  (initialise-global-env)
scheme>  (define  exit-evaluator
             (meta  (list  (cons  'exit  (lambda  (evaluate  e  r  k)  e)))))
scheme:
scheme>  (openloop  exit-evaluator  "exit>  "  "exit:  ")
exit>  (* 3 4)
exit: 12
exit>  (define  foo  (lambda  (x)  (*  x  x)))
exit:
exit>  foo
exit: #[procedure #x8B2D2]
```

```
exit> (foo 5)
exit: 25
exit> (exit)
scheme: (exit)
scheme> …
```

3 Finite Towers

3.1 Construction

The next step towards a reflective tower will be to use the code from (2) meta-circularly, building a finite tower of a fixed number of literal levels of open implementations. For this purpose we add a procedure *loadfile*, that enables us to read a file from disk and interpret its expressions one by one with an interpreter of our choice. *loadfile* is very similar to a read-eval-print loop, but also checks for *end-of-file* and reads expressions from a file instead of from the keyboard.

```
(define loadfile
  (lambda (evaluate file)
    ((lambda (port)
       ((lambda (loop)
          (set! loop
            (lambda (v)
              (if (eof-object? v)
                (close-input-port port)
                (evaluate v global-env
                   (lambda (ignore)
                     (loop (read port)))))))
          (loop (read port)))
        '*))
      (open-input-file file))))
```

Now it's easy to build a finite tower of open implementations. In the following example we will build a tower of 2 levels, with at every level of the tower an interpreter that is extended with a *climb* construct. The climb construct takes one argument. This argument must evaluate to a strictly positive integer value. Evaluation of climb will cause an exit from exactly the number of levels indicated by the argument. The climb construct is not a very useful thing, but we employ it because it is a simple example of a construct that needs an arbitrary number of reflection levels (depending on the argument).

First we load the file "open-simple.scm", which contains the definitions for *meta*, *loadfile*, *openloop*, … . Then we initialise the level 1 global environment for the first level and load "open-simple.scm" again, but this time into the level 1 global environment, using the level 1 basic evaluator.

```
scheme> (load "open-simple.scm")
scheme:
scheme> (initialise-global-env)
scheme:
scheme> (loadfile basic-eval "open-simple.scm")
scheme:
```

Next we define the dispatcher and evaluator for level 1. The evaluator for level 1 is an evaluator extended to handle *climb*.

```
scheme> (define climb-proc
              (lambda (evaluate e r k)
                   (evaluate (car (cdr e)) r
                       (lambda (how-many)
                         (if (= 1 how-many)
                           how-many
                             'cannot-climb-further)))))
scheme:
```



```
scheme> (define climb-dispatcher (list (cons 'climb climb-proc)))
scheme:
scheme> (define climb-evaluator (meta climb-dispatcher))
scheme:
```

After doing all of the above, the system is ready to start the level 1 read-eval-print-loop with a call to *openloop*.

```
scheme> (openloop climb-evaluator "1> " "1: ")
1>
```

To add another level to the tower we simply go through the same steps again, defining the dispatcher and evaluator and starting a read-eval-print loop. This time we can skip loading "open-simple.scm" because we do not intend to add a third level below level 2, so we don't need to load another meta-circular open implementation.

```
1> (initialise-global-env)
1:
1> (define climb-proc
            (lambda (evaluate e r k)
                 (evaluate (car (cdr e)) r
                     (lambda (how-many)
                       (if (= 1 how-many)
                         how-many
                           (climb (- how-many 1)))))))
1:
1> (define climb-dispatcher (list (cons 'climb climb-proc)))
1:
1> (define climb-evaluator (meta climb-dispatcher))
1:
1> (openloop climb-evaluator "2> " "2: ")
2>
```

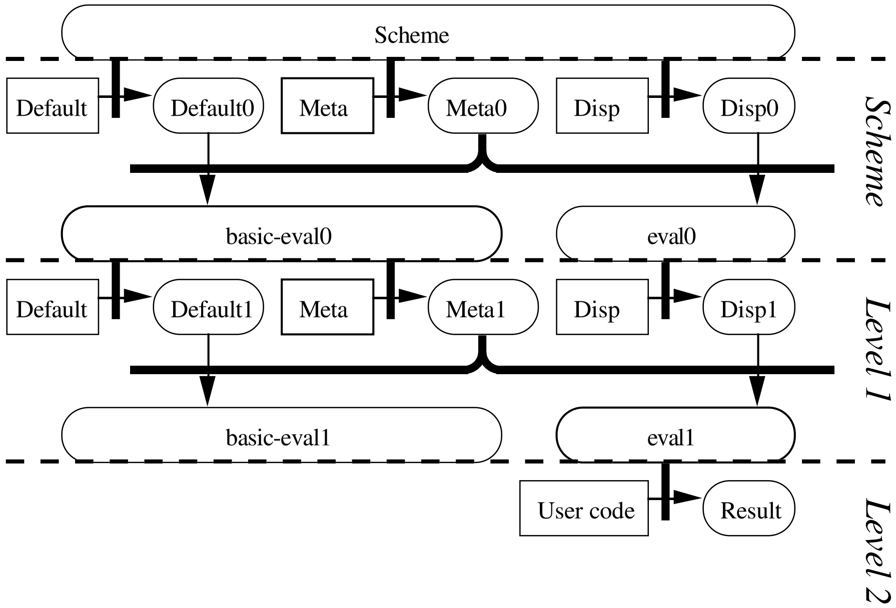
Notice that the definitions of climb for level 1 and level 2 are not identical. Normally climb calls "itself" recursively when it needs to climb more than one level. Strictly spoken "itself" is not correct, since the *climb* that is called and the one that is being implemented are in different levels of the tower. The level 2 *climb* is taken care of by *climb-proc* at level 1, which relies on level 1's climb. Since scheme, which coincides with level 0 in our tower, does not understand *climb*, the level 0 *climb-proc* (implementing level 1 climb), cannot rely on it. This is the reason why the level 0 *climb-proc* instead of calling *climb* "recursively" returns the symbol "cannot-climb-further". The result is that on level 2 we can climb at most 2 levels, and at level one we can climb only one level. Trying to climb more than this number of levels leaves us in Scheme, with the message "cannot-climb-further".

```
2> (* 3 4)
2: 12
2> (climb 1)
1: 1
1> (climb 2)
scheme: cannot-climb-further
scheme> …
```

3.2 Explanation

The following figure illustrates the configuration of the finite tower we just created. A little explanation is in order here. Dashed horizontal lines separate the different levels of the tower. Evaluation functions are drawn at the boundary, just above the dashed lines. Their application protrusions point through the dashed line into the level below, because they "reason about" objects from the level below, the level they are implementing. Notice that we actually have 2 towers here, standing right next to each other. On the left is a tower of basic evaluators, and on the right a tower of customised evaluators. The end product, the evaluator that is used to evaluate user code is the bottom of the tower of customised evaluators.

Although this is only a finite tower, and not a real reflective system, it already illustrates some interesting things. By representing a level of the tower as an open implemented interpreter, we have exactly the right amount of detail to be able to represent the independence of the fixed core of the system from the extensions. *Meta* and *default* are evaluated explicitly with *basic-eval*. This ensures that changes to the language introduced by the reflective parts of the user program (*disp* in the drawing) will not affect the interpretation of the fixed core.



In this implementation, the dashed line boundaries are very strict, function representations at different levels are incompatible, a function of one level cannot be used at another level. To see that this is true let's examine the representation of procedures at different levels, for an example let's consider the representation of the **+** procedures. At scheme's level this is simply represented by the native primitive addition procedure. On every level procedures are represented as procedures of the implementing level with one extra continuation argument in front of the real argument list (remember the definition of *make-compound* discussed in section 3). So at level 1 the representation for the primitive addition procedure will roughly correspond to the result of evaluating the following expression in native scheme:

```
(lambda (k1 . args) (k (apply + args)))
```

Similarly, the addition procedure at level 2 will correspond to the evaluation of the following at level 1:

```
(lambda (k2 . args) (k2 (apply + args)))
```

Which in turn corresponds to the following evaluated in scheme:

```
(lambda (k1 k2 . args) (k2 k1 (apply + args)))
```

Every level of the tower introduces an extra continuation, thus a procedure representation at level 1 takes one extra continuation argument, and a procedure at level 2 takes 2 extra continuation arguments. This obviously shows that procedures at different levels differ in the number of hidden continuation arguments they expect and are therefor not interchangeable.

4 Reflection

4.1 Why the Finite Tower is not Reflective

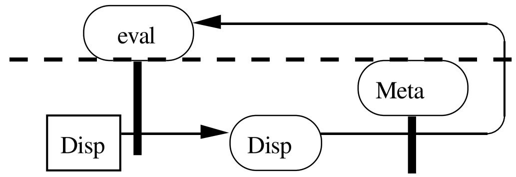
There are strong arguments to say that it would be a mistake to call the finite tower architecture just presented a reflective system. It isn't much more than an open implemented interpreter, it is just a number of open implemented interpreters executing each other. An open implemented interpreter in itself is not reflection, it is merely an interpreter that can be varied upon by supplying different parameters. The most important aspect of reflection, the ability of a program/interpreter to reason about itself is completely absent, an interpreter cannot reason about itself, it can only reason about the interpreter below. The interpreter below is a similar, but nevertheless a different interpreter, it need not even be extended in the same way.

4.2 The Fix-point Equation of Reflection

Merely an open implementation by itself is not reflection. What do we expect from a "reflective language/interpreter"? First of all, we need some meta-theory to talk about the interpreter. Second a *representation* of the evaluator is needed under this meta-theory. Obviously these 2 things are not sufficient to get a reflective system because both of these are present in open implementations: there is a meta-theory established by *meta* and the parameters to *meta* serve as representation for the evaluator. One essential thing is missing however. In a reflective system we want to be able to express the description for the interpreter in the language implemented by that interpreter itself! That is the essence of reflection! We express this in the following equation:

$$E = (M\ (E\ d))$$

Our convention is to write functional objects (round objects in the drawings) with capitals, and "source code" objects (rectangular in the drawings) with small letters. In this equation the evaluator, *E*, that is being created by applying the meta function, *M*, to a dispatcher function (representation of the evaluator) is the same as the evaluator that is being used to create the dispatcher by evaluating its source code (description) *d*. This fix-point equation is the key to reflection in a system of open implemented interpreters. Schematically we can draw this fix-point equation as follows:



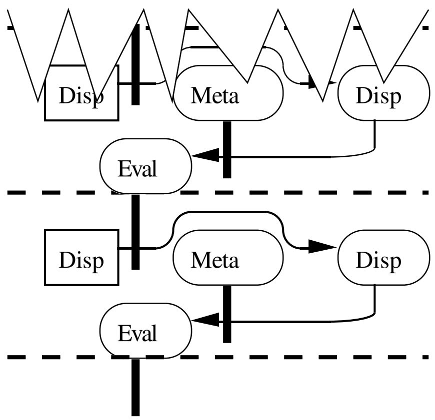
The "recursiveness" of the fix-point equation shows up in this diagram under the form of a circularity in the drawing. Notice that the arrow that leaves from the dispatcher goes through the level boundary. Remember that we pointed out before there cannot be arrows crossing level boundaries, the boundaries are strict. This is a complication we will deal with in the following 2 sections.

4.3 Reflection and Infinite Towers

The equation $E = (M\ (E\ d))$ actually implies an infinite tower of open implementations. This can be seen when rewriting the equation into longer and longer equivalent equations as follows:

$$E = (M\ (E\ d))$$
$$E = (M\ ((M\ (E\ d))\ d))$$
$$E = (M\ ((M\ ((M\ (E\ d))\ d))\ d))$$
$$\ldots$$
$$E = (M\ ((M\ ((M\ ((M\ ((M\ (\ldots\ d))\ d))\ d))\ d))\ d))$$

The last equation, the result of substituting $(M\ (E\ d))$ an infinite number of times into $E = (M\ (E\ d))$ can be regarded as an infinite tower, corresponding to the next figure.



In this figure there is an infinite number of levels (imagine that the level is repeated ad infinitum). Every level contains an evaluator, a meta and a dispatcher. You should consider all rounded rectangles containing the word "meta" to stand for one and the same meta-function, simply drawn multiple times. Similarly consider the rectangles and rounded rectangles representing dispatcher, dispatcher source code and evaluator as multiple drawings of the same objects. It is not difficult to see that actually the 2 drawings, the infinite tower and the one with the circularity, are "isomorphic", that is, if we simply look at the way the arrows go, and do not distinguish between the multiple copies at different levels, but regard them as identical, the two drawings are the same. In both drawings there are 2 arrows. One that starts from dispatcher source code goes through the evaluator and points to the dispatcher function, representing the application $(E\,D)$. The other arrow starts from the dispatcher function goes through the meta-function and points to the evaluator, representing the application $(M(E\,D))$.

Notice that in the infinite tower figure there are no arrows crossing level boundaries. This shows that we can think of the fix-point equation as an infinite tower of open implementations, without the discrepancy of level crossing procedures.

We consider both drawings to represent equivalent views of the fix-point equation. The circular one directly represents the recursion by a circularity in the drawing, thus looking at the equation from a rather direct, "implementational" viewpoint. While the infinite one looks at it from a conceptual, behavioural angle, viewing it as the representation for an infinite tower of identical open implementations. The first drawing we conceive as being of an implementational nature, because we are going to implement reflection directly by supplying an suitable *M*, that can be used to define an evaluator by directly expressing the fix-point equation, using recursion in ASEL to define *E* in terms of itself. The infinite tower drawing gives the more conceptual view, expressing that the result will be something that behaves like an infinite tower of identical open implementations, with identical extensions at every level.

5 Implementing the Infinite Tower

This section gives a brief description of how to implement an infinite tower of open implemented interpreters. This can be done in 2 stages.

5.1 Stage 1: Implementing *M*

The level crossing arrow in the figure from section (4.2) indicates that there is something strange about the *M* in the fix-point equation. Normally level crossing arrows are not possible.

Our implementation of *M* is inspired upon the left side of the finite tower from section 4, extended into infinity. An important aspect of the infinite tower is that procedures at different levels (near the bottom of the tower) are interchangeable. This is different from the situation in finite towers where procedures at different levels differ in the number of continuation arguments they receive as hidden arguments. At the bottom of an infinite tower, a procedure expects an infinite number of continuation arguments. Informally we could argue that adding one more continuation to an already infinite number doesn't make much of a difference, there still are an infinite number of them.

We have built a "level shifting" implementation, based on the idea of a *meta* at the bottom of an infinite tower. For technical reasons it was not possible to mimic the behaviour of an infinite tower exactly, but what matters is that we were able to ensure interlevel compatibility of procedures. The whole thing boils down to simulating an infinite tower by maintaining a stack of meta-continuations that is virtually infinite, but from which only a finite number of the topmost levels will actually be used. We will not go into detail because this is very similar[4] to the traditional implementation of a level shifting interpreter as previously described in [desRivières&Smith84] and [Smith84]. The result is an implementation of *meta* (from here on called $meta_\infty$) as an "infinite level procedure". It can be applied on dispatchers that contain "infinite level" procedures and returns an evaluator that also is an "infinite level" procedure.

5.2 Stage 2: Fix-point Equation

Given $meta_\infty$ we can use the fix-point equation given under (4.2) to spawn an infinite tower of customised interpreters. The system we have implemented provides a read-eval-print loop, in which the user can type in expressions. The evaluator used to evaluate the expressions is an instance of *basic-eval*, created by applying $meta_\infty$ to the default-dispatcher. The global environment contains a reference to $meta_\infty$ in a variable called "meta*". Thus the user can create his own dispatcher and pass it to $meta_\infty$ to create an infinite tower of customised evaluators.

The following is an example showing the creation of an infinite tower of evaluators that understand the *climb* syntax. The code is a bit more complicated than expected but this is merely the result of some technical matters. A dispatcher is not really a function, but an assoc-list, containing functions, this makes things a bit more intricate. Another complication is that Scheme, and ASEL (a subset of scheme) do not have delayed evaluation so we have to throw in some extra η-redexes here and there to avoid infinite loops.

First the variable “climb-code” is bound to the source code (notice the quote) of the climb-dispatch procedure. Thus the contents of “climb-code” corresponds to *d* in the equation.

```
0> (define climb-code ;; d
     '(lambda (evaluate e r k)
        (evaluate (car (cdr e)) r
           (lambda (how-many)
             (if (= 1 how-many)
               how-many
                 (climb (- how-many 1)))))))
0:
```

Next we construct a dispatcher assoc-list that contains the evaluation of the source code from “climb-code” and store that in a variable “climb-dispatcher”. The evaluator that should be used for evaluating the source code should be “eval-climb” the evaluator we are constructing. Since this evaluator will be declared later and is still unavailable, we must delay the evaluation with an extra η-redex.

```
0> (define climb-dispatcher ;; (E d)
     (list (cons 'climb
                 (lambda (eval e r k) ;; extra η-redex
                    (eval-climb climb-code global-env
```

4 Actually our implementation looks simpler and more elegant than similar things written for the traditional model because the ad-hoc notion of things moving up and down the tower has disappeared. The link between levels are through the parameters for *M*. Source code can be found in the Appendix.

```
                        (lambda (ED)
                          (ED eval e r k)))))))
0:
```

Subsequently we obtain the evaluator by giving the dispatcher created above as an argument to “meta*”. Here we also need an extra η-redex, this time to avoid infinite looping.

```
0> (define eval-climb ;; E = (M (E D))
     (lambda (e r k)   ;; extra η-redex
       ((meta* climb-dispatcher) e r k)))
0:
```

Finally we can use the evaluator. For example we can start a read-eval-print loop and evaluate some expressions.

```
0> (openloop eval-climb "1> " "1: ")
0:
1> (climb 3)
-2: 0
-2> …
```

6 Why the Infinite Tower of Meta-Functions is Reflective

It can be argued, on the basis of the time of definition and installation of dispatcher functions, that the tower of meta-functions is still a weaker form of reflection than the more common reifier functions sort of reflection. We will argue that the difference is merely a matter of 1) a lower degree of reflective overlap (something one really tries to avoid) and 2) pragmatics and the choice of the particular open implementation.

Dispatcher functions are defined and installed in the evaluator prior to their usage in some user program. It might seem that there is an even stronger form of reflection whereby a user program can install dispatcher functions while it is running. In this set-up dispatcher functions are defined in the context of the user program. First and most obvious one can remark that this leads to reflective overlap regarding the environment. Dispatcher functions are evaluated in an (implicit) environment that will later be given as explicit argument to them.

Furthermore it can be argued that the fact whether extensions to the interpreter are made prior to rather than during the execution of a program, depends largely on the architecture of the open implementation. We claim that in the system given here, this is mostly the result of the direct mapping of the theoretical "meta-functional" view of reflection onto an implementation which retains the functional nature of parameterisation. Most (if not all) present day implementations of reflective system involve some kind of side effect in the installation of parameters into an evaluator (with procedural reflection for example this is the side effect that installs a reifier procedure into the current environment). This naturally yields a more dynamic behaviour and gives a more direct impression that the system/program reasons about itself or about its evaluator.

To illustrate our point, it is possible to imagine a practical implementation that has the capability to destructively alter the dispatcher after the instantiation of an evaluator. This yields something that is highly similar to the reifier approach, be it with one big difference, it is impossible to pass on reifiers as arguments. This in itself is not a problem, some people even claim that the ability to pass on reifiers is a flaw in the procedural reflection approach [Bawden88].

7 Comparing Static Reflection, Dynamic Reflection and Finite Towers

The above defined *meta** need not be used to spawn infinite reflective towers. It need not be used to express programs that can climb arbitrarily high in the tower. Consider the following example. Here again a climb syntax is defined, but in contrast with the above climb syntax it can only be used to climb a number of levels in the tower which is statically defined. The evaluators being defined are not the result of some fix-point equation.

```
0> (define climb-proc-0
     (lambda (evaluate e r k)
       (evaluate (car (cdr e)) r
```



```
         (lambda (how-many)
           (if (= 1 how-many)
             how-many
               'cannot-climb-further)))))
0:
0> (define climb-dispatcher-0 (list (cons 'climb climb-proc-0)))
0:
0> (define climb-evaluator-0 (meta* climb-dispatcher-0))
0:
0> (openloop climb-evaluator-0 "1> " "1: ")
1> (define climb-proc-1
     (lambda (evaluate e r k)
       (evaluate (car (cdr e)) r
         (lambda (how-many)
           (if (= 1 how-many)
             how-many
               (climb (- how-many 1)))))))
1:
1> (define climb-dispatcher-1 (list (cons 'climb climb-proc-1)))
1:
1> (openloop climb-evaluator-1 "2> " "2: ")
2> (climb 1)
1: 1
1> (climb 2)
0: cannot-climb-further
0>
```

Although the above sequence of climb-evaluators looks very similar to the climb-evaluators defined with the finite tower of section 3, they have noticeably different properties. The following equations show the construction of both. Again capitals denote functions and small letters denote pieces of source code. *CD, D, BE*, *M* and *CE* stand for *climb-dispatcher*, *default-dispatcher*, *basic evaluator*, *meta* and *climb-evaluator* respectively. The indexing convention is that we put an index to denote the level a function belongs to if this level is finite. For example BE_1 is an instance of the basic evaluator that is a level 1 procedure (taking one continuation argument). If the level is infinite the index is omitted. Thus *BE* denotes an instance of basic-eval that is an "infinite level" procedure. Indexes to source code don't signify a certain level because pieces of source code are never bound to a certain level and can be freely interchanged without compatibility problems. Thus cd_0, … , cd_n merely denote *n* different pieces of source code.

$$CE = (M\ ((\ldots((M\ ((M\ (BE\ cd_0))\ cd_1))\ \ldots))\ cd_n)) \qquad \text{with meta*}$$

$$CE_{n+1} = (M_{n+1}\ ((\ldots((M_2\ ((M_1\ (BE_0\ cd_0))\ cd_1))\ \ldots))\ cd_n))$$

$$BE_0 = Scheme$$

$$M_i = (BE_{i-1}\ m) \qquad \text{finite tower}$$

$$BE_i = (M_i\ D_i)$$

$$D_i = (BE_{i-1}\ d)$$

Most noticeable is a difference in performance. In the second case there is very large interpretation overhead. There are *n* levels of interpreters literally interpreting each other, where *n* is the statically predetermined maximum number of levels one can *climb* in the tower. The first is much more efficient because the interpretation overhead will only occur for interpretation of the specific code in the dispatcher implementing the *climb* construct.

The performance cost of extra numbers of evaluation levels is avoided due to the special properties of $meta_\infty$. The extra performance cost associated with finite towers is due to extra flexibility in the architecture. Whereas using $meta_\infty$ implies that the meta function is fixed, the meta-function of finite towers need not all be evaluated with the same evaluator. This brings us back to the issues raised in the introduction. For a finite tower it *is* possible to influence the 'core' of the interpreter. For example one could define a sequence of evaluators:

$$CE_1 = ((BE_0\ m)\ (BE_0\ cd_0))$$
$$CE_2 = ((CE_1\ m)\ (CE_1\ cd_1))$$
$$\ldots$$
$$CE_{n+1} = ((CE_n\ m)\ (CE_n\ cd_n))$$

Unlike the previously employed finite tower that explicitly used a basic evaluator for evaluating *m*, the evaluator that would result from continuing this sequence into infinity cannot be represented by a single fix-point equation. We might propose the following equation:

$$E = ((E\ m)\ (E\ d))$$

However, this equation merely corresponds to a meta-circular "definition" of *E*. It is a well known fact that this does not really define anything. There is no unique solution to this equation thus it does not specify an interpreter or a language at all.

There is also a difference pertaining to procedure compatibility. In the literal tower, procedures at different levels cannot be interchanged whereas in the "static reflection" tower they can be interchanged. This is due to the nature of *meta** which was specifically written with procedure compatibility in mind.

Before going on let's first introduce some terminology. Two languages are *related* if both are a customisation of the same open implementation. For example in the drawing of the finite tower in section 3.2 the evaluators *basic-eval0* and *eval0* implement related languages since both where created from the same *meta0*. Whereas *eval0* and *eval1* do not implement related languages because they where created from different *meta*s. It is important to note that we *do not* consider *meta0* and *meta1* to be the same open implementation. Although they share the same source code, they are still distinct because they are procedures belonging to different levels and thus have different procedure representations (different number of continuations!). Note that in the example in this section all evaluators implement related languages because they where created from the same open implementation namely $meta_\infty$.

Using this terminology we can categorise open implementations into two different categories. The first is the category of "plain" open implementations in which customisation parameters are expressed in a language not necessarily related to the language they engender. The second is the category "with reflective potential" in which customisation parameters are expressed in a language that is related to the engendered language.

We can also distinguish 2 categories of reflection: static-reflection and dynamic reflection. An evaluator constructed without fix-point operations will be categorised as static reflection. If the construction involves some kind of fix-point operation than we will categorise it as dynamic reflection. Note that this fix-point need not be so direct as in the *climb* example. More exotic things like mutually recursive equations are also possible.

It is not difficult to see that any evaluator constructed without using a fix-point will always be limited beforehand in the number of levels it needs to “reflect”. Hence the term static reflection, the number of possible levels of reflection is statically limited by the construction of the evaluator. In the case of dynamic reflection there is no guaranteed statically determined upper bound on the number of reflection levels a program might require.

The “limited climb” in this section is a sample of static reflection. The number of reflection levels required is statically limited beforehand. The climb construct only works up to a limited number of levels, beyond that upper-bound it will stop and return the message “cannot-climb-further”. In section 5.2 the “unlimited” climb is an example of dynamic reflection. The number of reflection levels actually required is dynamically dependant on the execution of the program: it depends on the argument passed to *climb*.

8 Reflective Programming Languages Based on an Open Implementation

First let us consider what a reflective language is. Traditionally one considers two important requirements a language must conform to in order to be reflective [Smith...]. First it needs "an account of itself embedded

within it". In other words some kind of representation for the language must be accessible from within itself. Secondly this "self-representation" must be causally connected to the system so that changes to it directly affect the system itself.

Under this definition, what we have built does not qualify as a reflective system. It is more like a low-level "do-it-yourself kit". A global variable called *meta** contains a reference to $meta_\infty$. It is up to the user to construct recursive definitions over evaluators for defining dynamic reflection or open read-eval-print loops for constructing static reflection. This can be somewhat involved sometimes, e.g. the construction of the climb dispatcher is more complicated than need be due to need for lazy evaluation. Furthermore, this reference can be used to generate different interpreters, that can each be used to evaluate different parts of a program (an often useful property). No true support is given to manage all this.

Although it is not a reflective system in the traditional sense—it has no self representation embedded within it—it can be used to create interpreters from a description expressed in that same language. It is even possible to actually provide access to this "self-representation" from within the language itself. The user will have to do some programming to accomplish this however.

When building a real reflective system based on an open-implementation, we would use the parameters to *meta* as a self representation. Of course it is not practical to burden the user with explicitly constructing fix-points to obtain dynamic reflection etc. Normally one would determine some practical, sufficiently flexible ways for accessing the self-representation and hard-code this into the system. A suggestion for a practical implementation, that has the capability to destructively alter the dispatcher, was already given in section 6. In this case access to the self-representation could (for example) take place by storing the dispatcher table in a special global variable which is made available to the user. Less destructive forms are imaginable. For example, a reflective variant of scheme's *let*, and *letrec*, can be provided to the user as standard mechanisms for writing statically or dynamically reflective code respectively.

9 Conclusion

The traditional model of reflection is not sufficiently detailed for expressing the fact that every interpreter has a fixed “untouchable” core that cannot be affected by reflective programming. Our open-implementation approach adds some detail to fix this, dividing an interpreter into a fixed and a parameterised part.

The open-implementation view gives a better and cleaner understanding of reflection. For one thing, it improves upon the ad-hoc and obscure way levels are linked to each other in the traditional approach. Another important aspect of the open-implementation model is that it clearly exhibits the notion of theory relativity. It actually takes the fact that some parts of the system will never be represented in its (self-)representation as a premise and puts these things separately into a *meta* function. This *meta* function establishes the meta-theory and the parameters to the functions constitute the representation of a language/representation. Thus the “meta-theory” and the notion of “representation” are clearly defined before we even start thinking about reflection. Reflection is then obtained by making a representation (parameters to *meta*) available from within the language.

It is our opinion that this alternative view on reflective systems will strongly influence the definition, implementation and theory of reflective systems. In fact it can now be argued that a large part of the literature on reflective systems is devoted to "variations on open implementations" for particular systems (e.g. alternative open implementations for Scheme). Which is an important topic, of course, but a topic that can be considered as a research topic that is more general than reflective systems.

The good news is that, given this alternative view on reflection, it is possible to start considering a generalised theory of reflective systems. As pointed out in [Mendhekar,Friedman93] when the view taken on reflective systems is: base system + reflective operators (the traditional view) then: "*we can never hope to have a generalised theory about reflective systems since the theory will have to take into account the operational behaviour of every base system*". This problem is entirely resolved when reflection is based on open implementations: the operational behaviour of the base system has already been taken care of in defining the open implementation, and is a prerequisite before turning a system into a reflective one.

10 References

[Bawden88] A. Bawden: **Reification without Evaluation**, Conference Record of the 1988 ACM Symposium on LISP and Functional Programming, 1988.

[desRivières&Smith84] J. des Rivières and B. C. Smith: **The Implementation of Procedurally Reflective Languages**, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp 331-347, Austin, Texas (August 1984)

[Friedman&Wand84] D.P. Friedman, and M. Wand: **Reification: Reflection without Metaphysics**, Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pp 348-355, Austin, Texas (August 1984)

[Jefferson&Friedman92] S. Jefferson, and D.P. Friedman: **A Simple Reflective Interpreter**, IMSA'92 International Workshop on Reflection and Meta-Level Architecture, Tokyo, November 4-7, 1992. (1992)

[Kickzales, des Rivières,Bobrow91] G. Kickzales, J. des Rivières, and D. G. Bobrow: **The Art of the Metaobject Protocol**, The MIT Press, Cambridge, Massachusetts, 1991.

[Maes87] P. Maes: **Computational Reflection**, VUB AI-Lab technical report 87-2. (1987)

[Maes88] P. Maes: **Issues in Computational Reflection**, Meta-Level Architectures and Reflection, P. Maes and D. Nardi (eds.) Elsevier Publishers B.V. (North-Holland). (1988)

[Mendhekar,Friedman93] A. Mendhekar, D.P. Friedman: **Towards a Theory of Reflective Programming Languages.** In informal proceedings of the OOPSLA'93 workshop on Object-Oriented Reflections and Meta-level Architectures, October 1993.

[Rao91] R. Rao: **Implementational Reflection in Silica**, Lecture Notes in Computer Science, P. America (ed.), ECOOP'91, European Conference on Object-Oriented Programming, Springer Verlag. (1991)

[Simmons&Friedman92] J.W. Simmons II, and D.P. Friedman: **A Reflective System is as Extensible as its Internal Representations: An Illustration**, Indiana University Computer Science Department Technical Report #366. (1992)

[Simmons,Jefferson&Friedman92] J.W. Simmons II, S. Jefferson, and D.P. Friedman: **Language Extensions via First-class Interpreters**, Indiana University Computer Science Department Technical Report #362. (1992)

[Smith84] B. C. Smith: **Reflection and Semantics in Lisp**, Conf. Rec 11th ACM Symp on Principles of Programming Languages (Salt Lake City, January 1984, pp23-35. (1984).

[Wand&Friedman88] M. Wand, and D. P. Friedman: **The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower**, Meta-Level Architectures and Reflection, P. Maes and D. Nardi (eds.) Elsevier Publishers B.V. (North-Holland). (1988)

A Appendix: "plain" open-implementation for building finite towers

```
;-------------------------
; Open evaluator for ASEL
; possible to use meta-circularly
;-------------------------

(define meta
  (lambda (dispatch-table)

    (define evaluate
      (lambda (e r k)
        (if (pair? e)
           (find-pair (car e) dispatch-table
              (lambda (success-pair)
                ((cdr success-pair) evaluate e r k))
              (lambda ()
                (basic-evaluate e r k)))
           (basic-evaluate e r k))))

    (define basic-evaluate
      (lambda (e r k)
        ((if (constant? e)
           evaluate-constant
           (if (variable? e)
             evaluate-variable
             (if (if? e)
               evaluate-if
               (if (assignment? e)
                 evaluate-assignment
                 (if (definition? e)
                   evaluate-definition
                   (if (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         e r k)))

    (define evaluate-constant
      (lambda (e r k)
        (k (constant-part e))))

    (define evaluate-variable
      (lambda (e r k)
        (get-pair e r
           (lambda (success-pair)
             (k (cdr success-pair)))
           (lambda ()
             (wrong "symbol not bound: " e)))))

    (define wrong
      (lambda (message object)
        (display "Error:")
        (display message)
        (display object)
        (newline)))

    (define evaluate-if
      (lambda (e r k)
        (evaluate (test-part e) r
           (lambda (v)
             (if v
               (evaluate (then-part e) r k)
               (evaluate (else-part e) r k))))))

    (define evaluate-assignment
      (lambda (e r k)
        (evaluate (value-part e) r
```

```
          (lambda (v)
            (find-pair (id-part e) (car r)
               (lambda (success-pair)
                 (set-cdr! success-pair v)
                 (k (void)))
               (lambda ()
                 (set-car! global-env
                      (cons (cons (id-part e) v)
                            (car global-env)))
                 (k (void)))))))

    (define evaluate-definition
      (lambda (e r k)
        (evaluate (value-part e) r
          (lambda (v)
            (find-pair (id-part e) (car r)
               (lambda (success-pair)
                 (set-cdr! success-pair v)
                 (k (void)))
               (lambda ()
                 (set-car! r
                      (cons (cons (id-part e) v)
                            (car r)))
                 (k (void)))))))

    (define evaluate-abstraction
      (lambda (e r k)
        (k (make-compound
            (formals-part e) (body-part e) r))))

    (define evaluate-combination
      (lambda (e r k)
        ;(display "@: ")
        ;(write e)
        ;(newline)
        (evaluate (operator-part e) r
          (lambda (proc)
            (evaluate-operands (operands-part e) r
              (lambda (args)
                (apply-procedure proc args k)))))))

    (define evaluate-operands
      (lambda (operands r k)
        (if (null? operands)
          (k '())
          (evaluate (car operands) r
            (lambda (v)
              (evaluate-operands (cdr operands) r
                (lambda (w)
                  (k (cons v w)))))))))

    (define evaluate-sequence
      (lambda (body r k)
        (if (null? (cdr body))
          (evaluate (car body) r k)
          (evaluate (car body) r
            (lambda (v)
              (evaluate-sequence (cdr body) r k))))))

    (define make-compound
      (lambda (formals body r)
        (lambda (k . args)
          (evaluate-sequence body (extend r formals args) k))))

    evaluate))

(define apply-procedure
      (lambda (proc args k)
        (if (procedure? proc)
```



```
          (apply proc (cons k args))
          (wrong "operator is not a procedure" proc))))

(define extend
    (lambda (r ids vals)
      (cons (extend-frame '() ids vals) r)))

(define extend-frame
  (lambda (f ids vals)
    (if (null? ids)
      f
      (if (pair? ids)
        (extend-frame (cons (cons (car ids) (car vals)) f)
                      (cdr ids)
                        (cdr vals))
        (cons (cons ids vals) f)))))

(define get-pair
    (lambda (id r success failure)
      (if (null? r)
        (failure)
        (find-pair id (car r)
          success
          (lambda ()
            (get-pair id (cdr r) success failure))))))

(define find-pair
   (lambda (elt alist success failure)
     ((lambda (assq-result)
        (if assq-result
          (success assq-result)
          (failure)))
      (assq elt alist))))

(define empty-env '())

(define 1st (lambda (l) (car l)))
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))

(define test-tag
  (lambda (tag)
    (lambda (e)
      (if (pair? e) (eq? (car e) tag) #f))))

(define make-primitive ;;only for "non-higher order" primitives
  (lambda (op)
    (lambda (k . args)
      (k (apply op args)))))

(define primitive-identifiers
   (lambda ()
    '(car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
      close-input-port open-input-file void procedure?)))

(define primitive-procs
  (lambda ()
    (list car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
      close-input-port open-input-file void procedure?)))

(define variable? symbol?)
(define if? (test-tag 'if))
(define assignment? (test-tag 'set!))
```

```scheme
(define definition? (test-tag 'define))
(define abstraction? (test-tag 'lambda))
(define quote? (test-tag 'quote))

(define constant?
  (lambda (e)
    (if (pair? e) (quote? e)
      (if (symbol? e) #f #t))))

(define constant-part
    (lambda (e) (if (quote? e) (2nd e) e)))

(define test-part 2nd)
(define then-part 3rd)
(define else-part 4th)

(define id-part 2nd)
(define value-part 3rd)

(define formals-part 2nd)
(define body-part (lambda (e) (cdr (cdr e))))

(define operator-part 1st)
(define operands-part cdr)

(define void
    ((lambda (v) (lambda () v)) (cons '* '*)))

(define mapper
      (lambda (f l)
        (if (null? l)
          '()
          (cons (f (car l)) (mapper f (cdr l))))))

(define initialize-global-env
      (lambda ()
        (set! global-env
           (extend
             empty-env
             (cons 'apply (primitive-identifiers))
             (cons (lambda (k proc args)
                     (apply-procedure proc args k))
                   (mapper make-primitive (primitive-procs)))))))

(define default-dispatcher '())

(define basic-eval (meta default-dispatcher))

(define openloop
  (lambda (evaluate read-prompt write-prompt)
    (display read-prompt)
    (evaluate (read) global-env
      (lambda (v)
        (display write-prompt)
        (if (eq? v (void))
          "Nothing is displayed"
          (write v))
        (newline)
        (openloop evaluate read-prompt write-prompt)))))

(define loadfile
  (lambda (evaluate file)
    ((lambda (port)
       ((lambda (loop)
          (set! loop
            (lambda (v)
              (if (eof-object? v)
                (close-input-port port)
                (evaluate v global-env
```



```scheme
                    (lambda (ignore)
                      (loop (read port)))))))
           (loop (read port)))
         '*))
     (open-input-file file))))

(define boot-1-level
  (lambda (evaluate input-prompt output-prompt)
    (initialize-global-env)
    (loadfile basic-eval this-file-name)
    (openloop evaluate input-prompt output-prompt)))

(define start
  (lambda (evaluate input-prompt output-prompt)
    (initialize-global-env)
    (openloop evaluate input-prompt output-prompt)))

(define global-env 'dummy);;just so that global-env exists and can be set! to

(define this-file-name "open-simple.scm")
```

B. Open Implementation "with reflective potential" for fix-point equations

```scheme
;-------------------------
; Open evaluator for ASEL with reflective potential
; read this file into scheme and then evaluate `(start)'

;;---- Stack of meta continuations for simulating an infinite tower

(define make-default-stack
  (lambda (level)
    (list 'cstack level (- level 1))))

(define push
  (lambda (base-stack cont)
    (list 'cstack cont base-stack)))

(define pop
  (lambda (stack)
    (if (number? (3rd stack))
        (make-default-stack (3rd stack))
        (3rd stack))))

(define top
  (lambda (stack)
    (if (number? (2nd stack))
      (make-loop (2nd stack))
      (2nd stack))))

;; procedure for creating the revpl procedure for default continuations
(define make-loop
  (lambda (level)
    (define loop
      (lambda (m v)
        (display level)
        (display ": ")
        (if (eq? v (void))
          "nothing is displayed"
          (write v))
        (newline)
        (display level)
        (display "> ")
        (basic-eval m (read) global-env loop)))
    loop))

;;sometimes we need something that behaves like (lambda (v) v) as
; continuation
(define id-cont
```

```
  (push 'should-not-be-used (lambda (m v) v)))

;;---------
;This file is based on a copy of "open-simple.scm"
;It has been converted a bit to simulate an infinite tower.
;All lambdas have been replaced by similar lambdas with an extra first
;argument:
;a stack of meta-continuations
;all calls to such procedures similarly have been converted to pass on a
;stack of meta continuations.
;Note that continuations are also procedures and thus also have to receive
;a stack of meta-continuations as first argument.
;
;The following variable names are used throughout the file
;m : stack of meta continuations
;k : continuation
;e : expression
;r : environment
;;---------

(define meta
  (lambda (m dispatch-table)

    (define evaluate
      (lambda (m e r k)
        (if (pair? e)
           (find-pair (car e) dispatch-table
              (lambda (success-pair)
                ((cdr success-pair) m evaluate e r k))
              (lambda ()
                (basic-evaluate m e r k)))
           (basic-evaluate m e r k))))

    (define basic-evaluate
      (lambda (m e r k)
        ((if (constant? e)
           evaluate-constant
           (if (variable? e)
             evaluate-variable
             (if (if? e)
               evaluate-if
               (if (assignment? e)
                 evaluate-assignment
                 (if (definition? e)
                   evaluate-definition
                   (if (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         m e r k)))

    (define evaluate-constant
      (lambda (m e r k)
        (k m (constant-part e))))

    (define evaluate-variable
      (lambda (m e r k)
        (get-pair e r
           (lambda (success-pair)
             (k m (cdr success-pair)))
           (lambda ()
             (wrong m "symbol not bound: " e)))))

    (define evaluate-if
      (lambda (m e r k)
        (evaluate m (test-part e) r
           (lambda (m v)
             (if v
               (evaluate m (then-part e) r k)
               (evaluate m (else-part e) r k))))))
```



```
    (define evaluate-assignment
      (lambda (m e r k)
        (evaluate m (value-part e) r
           (lambda (m v)
             (find-pair (id-part e) (car r)
                (lambda (success-pair)
                  (set-cdr! success-pair v)
                  (k m (void)))
                (lambda ()
                  (set-car! global-env
                       (cons (cons (id-part e) v)
                             (car global-env)))
                  (k m (void))))))))

    (define evaluate-definition
      (lambda (m e r k)
        (evaluate m (value-part e) r
           (lambda (m v)
             (find-pair (id-part e) (car r)
                (lambda (success-pair)
                  (set-cdr! success-pair v)
                  (k m (void)))
                (lambda ()
                  (set-car! r
                       (cons (cons (id-part e) v)
                             (car r)))
                  (k m (void))))))))

    (define evaluate-abstraction
      (lambda (m e r k)
        (k m (make-compound
              (formals-part e) (body-part e) r))))

    (define evaluate-combination
      (lambda (m e r k)
       ;(display "@: ")
       ;(write e)
       ;(newline)
        (evaluate m (operator-part e) r
           (lambda (m proc)
             (evaluate-operands m (operands-part e) r
                (lambda (m args)
                  (apply-procedure m proc args k)))))))

    (define evaluate-operands
      (lambda (m operands r k)
        (if (null? operands)
          (k m '())
          (evaluate m (car operands) r
             (lambda (m v)
               (evaluate-operands m (cdr operands) r
                  (lambda (m w)
                    (k m (cons v w)))))))))

    (define evaluate-sequence
      (lambda (m body r k)
        (if (null? (cdr body))
          (evaluate m (car body) r k)
          (evaluate m (car body) r
            (lambda (m v)
              (evaluate-sequence m (cdr body) r k))))))

    (define make-compound
      (lambda (formals body r)
        (lambda (m . args)
          (evaluate-sequence (pop m) body (extend r formals args) (top m)))))

    ((top m) (pop m) evaluate)))
```

```
(define wrong
  (lambda (m message object)
    (display "Error:")
    (display message)
    (display object)
    (newline)
    ((top m) (pop m) 'error)))

(define apply-procedure
      (lambda (m proc args k)
        (if (procedure? proc)
          (apply proc (cons (push m k) args))
          (wrong m "operator is not a procedure" proc))))

(define extend
    (lambda (r ids vals)
      (cons (extend-frame '() ids vals) r)))

(define extend-frame
  (lambda (f ids vals)
    (if (null? ids)
      f
      (if (pair? ids)
        (extend-frame (cons (cons (car ids) (car vals)) f)
                      (cdr ids)
                        (cdr vals))
        (cons (cons ids vals) f)))))

(define get-pair
    (lambda (id r success failure)
      (if (null? r)
        (failure)
        (find-pair id (car r)
           success
           (lambda ()
             (get-pair id (cdr r) success failure))))))

(define find-pair
   (lambda (elt alist success failure)
     ((lambda (assq-result)
        (if assq-result
          (success assq-result)
          (failure)))
      (assq elt alist))))

(define empty-env '())

(define 1st (lambda (l) (car l)))
(define 2nd (lambda (l) (car (cdr l))))
(define 3rd (lambda (l) (car (cdr (cdr l)))))
(define 4th (lambda (l) (car (cdr (cdr (cdr l))))))
(define 5th (lambda (l) (car (cdr (cdr (cdr (cdr l)))))))

(define test-tag
  (lambda (tag)
    (lambda (e)
      (if (pair? e) (eq? (car e) tag) #f))))

(define make-primitive ;;use only for "non-higher order" primitives
  (lambda (op)
    (lambda (m . args)
      ((top m) (pop m) (apply op args)))))

(define primitive-identifiers
   (lambda ()
    '(car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
```



```
      close-input-port open-input-file void procedure?)))

(define primitive-procs
  (lambda ()
    (list car cdr cons set-car! set-cdr! assq memq
      null? = eq? newline write display read
      + - * symbol? list pair? eof-object?
      close-input-port open-input-file void procedure?)))

(define variable? symbol?)
(define if? (test-tag 'if))
(define assignment? (test-tag 'set!))
(define definition? (test-tag 'define))
(define abstraction? (test-tag 'lambda))
(define quote? (test-tag 'quote))

(define constant?
  (lambda (e)
    (if (pair? e) (quote? e)
      (if (symbol? e) #f #t))))

(define constant-part
    (lambda (e) (if (quote? e) (2nd e) e)))

(define test-part 2nd)
(define then-part 3rd)
(define else-part 4th)

(define id-part 2nd)
(define value-part 3rd)

(define formals-part 2nd)
(define body-part (lambda (e) (cdr (cdr e))))

(define operator-part 1st)
(define operands-part cdr)

(define void
    ((lambda (v) (lambda () v)) (cons '* '*)))

(define mapper
      (lambda (f l)
        (if (null? l)
          '()
          (cons (f (car l)) (mapper f (cdr l))))))

(define openloop
  (lambda (m evaluate read-prompt write-prompt)
    (display read-prompt)
    (evaluate m (read) global-env
      (lambda (m v)
        (display write-prompt)
        (if (eq? v (void))
          "Nothing is displayed"
          (write v))
        (newline)
        (openloop m evaluate read-prompt write-prompt)))))

(define initialize-global-env
      (lambda ()
        (set! global-env
           (extend
             empty-env
             (cons 'apply (primitive-identifiers))
             (cons (lambda (m proc args)
                     (apply-procedure (pop m) proc args (top m)))
                   (mapper make-primitive (primitive-procs)))))
        (set! global-env
              (extend global-env
```

```
                      '(meta* default-dispatcher openloop)
                  (list meta  default-dispatcher openloop)))))

(define default-dispatcher '())

(define basic-eval (meta id-cont default-dispatcher))

(define start
  (lambda ()
    (initialize-global-env)
    (set-car! global-env (cons (cons 'global-env global-env) (car global-env)))
    (let ((s (make-default-stack 0)))
      ((top s) (pop s) 'begin))))

(define boot
  (lambda ()
    (initialize-global-env)
    (let ((s (make-default-stack 0)))
      ((top s) (pop s) 'begin))))

(define global-env 'dummy)
;;just so that global-env exists and can be set! to
```