Vrije Universiteit Brussel
Faculteit Wetenschappen

Construction of the Reflective Tower Based on Open Implementations

Kris De Volder & Patrick Steyaert
email: {kdvolder|prsteyae}@vnet3.vub.ac.be

Techreport vub-prog-tr-95-01

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Construction of the Reflective Tower Based on Open Implementations

Abstract

It is our opinion that the traditional model of reflection—towers of meta-circular interpreters—is not sufficiently detailed. We propose a more fine grained construction of the reflective tower not by means of meta-circular interpreters, but rather by means of (meta-circular) open implementations. We find that this approach adds precisely the needed amount of detail missing from the traditional tower model.

1 Introduction

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.

In [Smith84] a very thorough analysis of procedural reflection is given. The ideas presented by Smith have been widely accepted and have been the basis of most (if not all) implementations/treatments of reflection to date. We too subscribe to the opinion that “towers” are an essential part of reflection. One thing that bothers us though is the rather add-hoc fashion whereby interpreters are extended by “adding lines” to them.

Recently open implementations have been investigated [Kickzales,desRivières,Bobrow91], [Kickzales92], [Rao91]. Open implementations are useful even without reflection. The connection between open implementations and reflection has already been made [Kickzales, des Rivières,Bobrow91]. However nobody has bothered yet to relate the tower model directly with open implementations—that is, construct a tower of open implementations instead of meta-circular interpreters. This is exactly what we do in this paper, with a genuine open implementation for a Scheme-like programming language.

We show that open implementations provide exactly the amount of detail needed to overcome the add-hoc notion of “adding lines to the interpreter”. An open implementation has a separate and clearly defined meta-interface for doing just this. By using open implementations we get a more fine-grained view of the reflective tower.

2 Open Implementations of Programming Languages

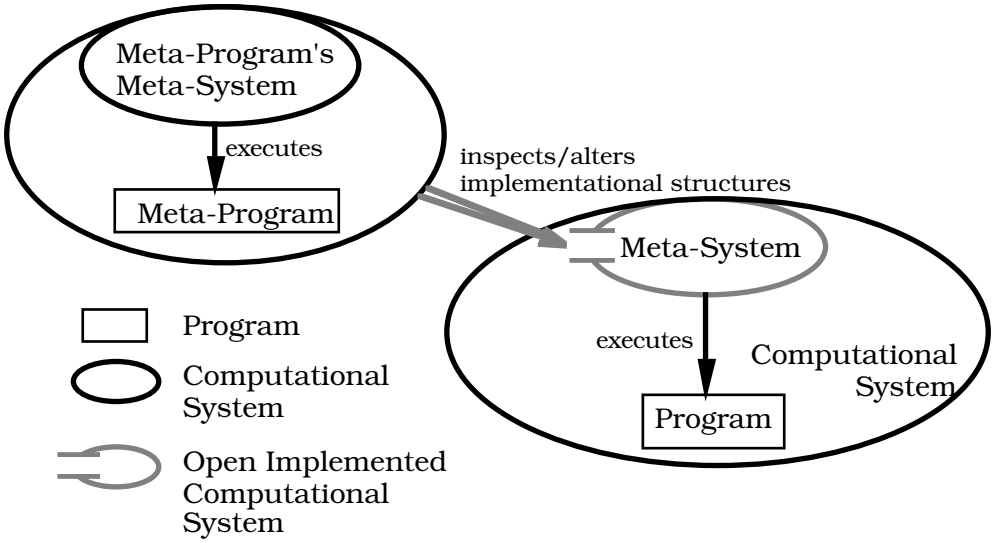
Open implementations distinguish themselves from plain 'black-box' or fixed implementations by giving a structured, 'open-ended' access to their implementation.

> ***Open Implementations*** *[Rao91] : A system with an open implementation provides (at least) two linked interfaces to its clients, a base-level interface to the system's functionality similar to the interface of other such systems, and a meta-level interface that reveals aspects of how the base-level interface is implemented.*

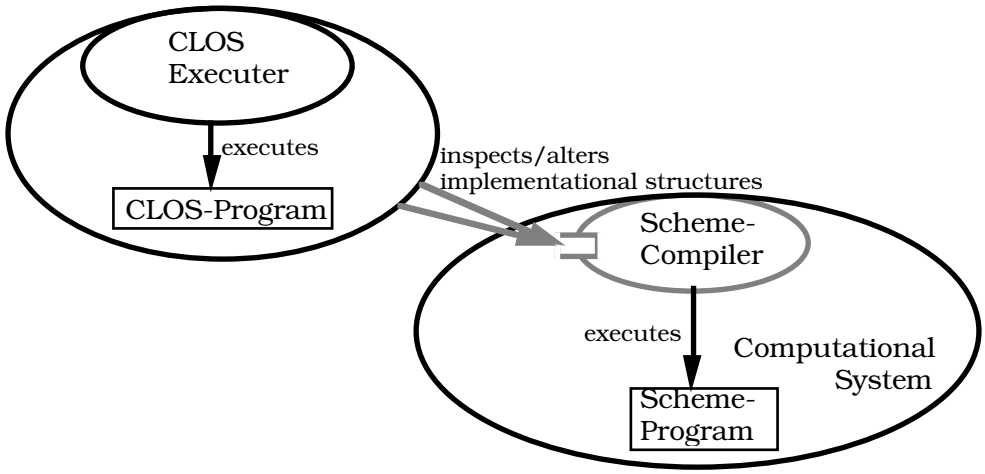
The idea is that a user of an open implemented system can, by means of the meta-level interface, have a substantial influence on the implementation, and accordingly, the behaviour of the system. In an open implementation the meta-level interface specifies points where the user can provide alternative implementations. Such an alternative implementation can differ from the default implementation of the system in performance characteristics, or it can alter the behaviour of the system, or it can extend the system with new behaviour. The extent to which the behaviour of the system can be altered, or extended, depends on the meta-level interface and its link to the base-level interface. In the most literal sense the meta-level interface only gives access to implementational issues of the system. In a more liberal sense it also allows extending the behaviour of the system. Thus, rather than defining a single system, an open implementation defines a entire design space of systems. Open implementations for windowing systems [Rao91], programming languages [Kiczales&al.91], compilers [Rodriguez92], and operating systems [Yokoto&al.89] have been documented in the literature.

We are interested in open implementations for programming languages. For this kind of open implementation the base-level functionality is "executing programs". The meta-level interface allows controlled access to and modification of the interpretation (or compilation) process. The purpose of this can be optimisation [Kickzales92]—fine tuning an implementation for a special type of application in order to improve performance. The purpose can also be extending the implemented language—adapting the language to the needs of an application domain in order to improve conciseness of programs in the domain. Both purposes do not exclude one another.

The picture below illustrates the typical configuration of an open implementation for a programming language. We call a program that is being executed a *computational system*. The program itself is executed by another computational system called the *meta-system*. In our case the meta-system is an open implementation that provides a meta-interface for acting upon its implementational structures. In "plain" open implementations this meta-interface is programmed from another computational system consisting of a *meta-program* and a meta-system that executes the meta-program.



In an open implementation of a programming language two (kinds of) languages are of importance. First, the language in which the open implementation, and consequently all code that is added to this open implementation by means of the meta-level interface, is expressed. And second, the languages that are being implemented (not one but many). We will call the former language the *meta-language* of the open implementation, and the latter the languages *engendered* by the open implementation. The meta-language and the engendered languages need not be related. Even in practice examples can be found where it is advantageous to have a meta-language that totally differs from the engendered languages. For example in [Kiczales93] a CLOS open implementation for a Scheme compiler is briefly mentioned.



3 An example Open Implementation

We will use one particular example open implementation—we call it *meta*—throughout the rest of this paper. The open implementation is written "meta-circularly". In this context, *meta-circularity* means that the open implementation is written in the default language of that open implementation itself. The syntax of the default language, *metaD*, is depicted below:

```
<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>*)
<if>           =  (if <exp> <exp> <exp>)
<definition>   =  (define <var> <exp>)
<assignment>   =  (set! <var> <exp>)
<application>  =  (<exp> <exp>*)
```

An evaluator for *metaD* can be expressed as a dispatcher on the type of expression to be evaluated. Each expression is tagged with an expression type. A tag can for example be an atom at the head of each list that represents an expression, except in a few special cases such as procedure application and variable reference. Typical tags for expression types are 'lambda', 'if',…. This tag is used by the dispatcher to invoke an appropriate evaluation function. For the above listed tags these evaluation function would respectively be a function to construct a closure, evaluate an if expression, etc.

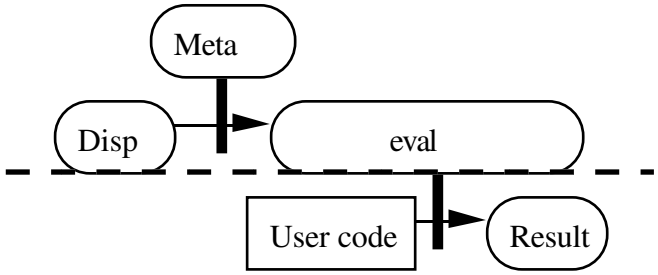
A useful open implementation would be one in which clauses can be added to this dispatcher, thereby allowing to add new expression types and their corresponding evaluation function. An extension to the dispatcher can be formulated as a list that associates tags to evaluation functions. The open implementation takes the form of a function (the meta-function) that has such a list as argument and returns an extended evaluator.

```
(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 (abstraction? e)
                     evaluate-abstraction
                     evaluate-combination))))))
         e r k)))

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

    evaluate))
```

The basic evaluator accepts an expression *e*, an environment *r* and a continuation *k*. It is nothing but a bunch of nested *if*s to decide the type of the expression and call the appropriate evaluation procedure. The open implementation itself accepts an assoc-list, *dispatch-table,* and returns a modified evaluator, *evaluate*. The modified evaluator uses *dispatch-table* to override or extend the basic evaluator's default dispatching mechanism. The *dispatch-table* contains associations of symbols—identifying the type of expression—to specific evaluation procedures. Such an evaluation procedure accepts all the expected parameters—expressions, environment and continuation—and one special parameter, the evaluator itself, to facilitate recursive evaluation of sub-expressions.

Note that the open implementation *reifies*[1] continuations as well as environments. It would be much simpler to absorb continuations and only reify environments. However such an open implementation can be turned into a reflective system much to easily for the example to have any value. The need for a reflective tower becomes apparent very naturally when continuations are reified.

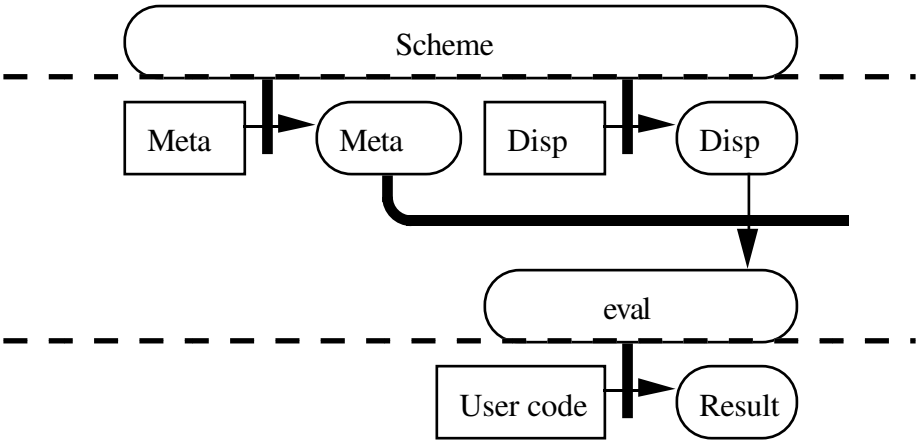
The base-level interface of this simple open implementation is the evaluator. The meta-level interface is the above meta-function. Base and meta-level interface are linked by the fact that the meta-function, given an appropriate extension to the dispatcher, returns an extended evaluator as a result. The following figure gives a schematic view of what our open implementation looks like.



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. One 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, ….

---

1 Reification is a commonly accepted term. It signifies "making explicit" a certain aspect of computation in a program. The antonym, *absorption*, means using the implementation languages runtime structures to *absorb*—i.e. implicitly use—the aspect. A good example is the use of an explicit stack (i.e. reifying the stack) for implementing backtracking or absorbing the recursion stack of the implementation language.



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* objects.

4 Using the open implementation

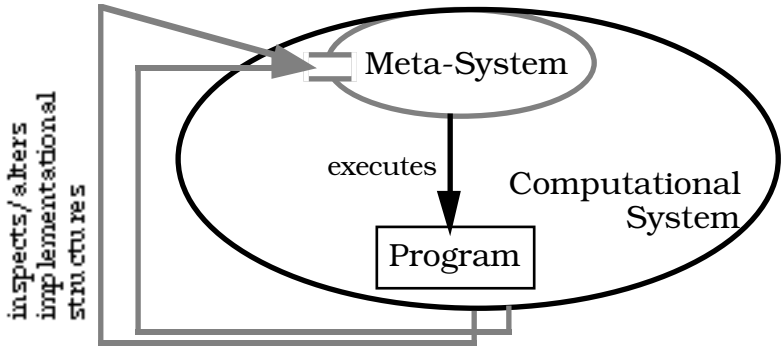
Our example implementation provides a procedure *open-loop* that can be used to start a *read-eval-print* loop with a given evaluator. The read-eval-print loop is parameterised with an “input-prompt” and an “output-prompt” string which are printed before prompting the user for input or displaying output. We can open a read-eval-print loop on a basic evaluator, or variation of the evaluator we created by applying meta to a parameter. The following example shows the creation of an extended evaluator that understands a special *exit* expression. When an *exit* expression is evaluated, the continuation is not called. This 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. For clarity and uniformity we have edited the transcript so that it appears as if the underlying Scheme has input and output prompts “scheme>” and “scheme:” respectively.

```
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> …
```

5 Open Implementations and Reflection

5.1 Reflection: Accessing One's Own Meta-system

As we have described above, in a "plain" open implementation one computational system acts upon another computational system, accessing the meta-interface of that other system's meta-system. Essentially, reflection means that a computational system has the possibility to access its own meta-system's meta-interface directly as depicted below:



5.2 The Fix-point Equation of Reflection

If we apply the above "definition" of reflection to our example open implementation this means that we have to be able to supply a dispatcher for *meta* from within the very same language created by meta with that dispatcher. Or, put otherwise, we want to be able to create a dispatcher by evaluating some source code with the evaluator that is engendered by that dispatcher itself. We can characterise this with the following fix-point equation:

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

Where E stands for an evaluator, M stands for *meta* and d is the source code for a dispatcher-table. The convention is to write functional objects (round objects in the drawings) with capitals, and "source code" objects (rectangular in the drawings) with small letters. This equation can be looked upon as a recursive definition of the evaluator E.

5.3 Open Implementations with "reflective potential"

Not every open implementation can be used to do reflective programming. For example the CLOS open implementation of a Scheme compiler can not be used as such. This is because the meta-level interface must be programmed in CLOS whereas the base language is Scheme. Surprisingly, even if the base and meta-language are the same, it usually is not possible to use the open implementation reflectively. This is so because even though the language is the same, we have two different implementations of the language at hand. One of these is engendered by the open implementation and the other is the one that is used to implement the open implementation. At the implementation level they have different representations for things such as procedures, etc. Compare this with for example two PASCAL compilers from different manufacturers. Both implement the

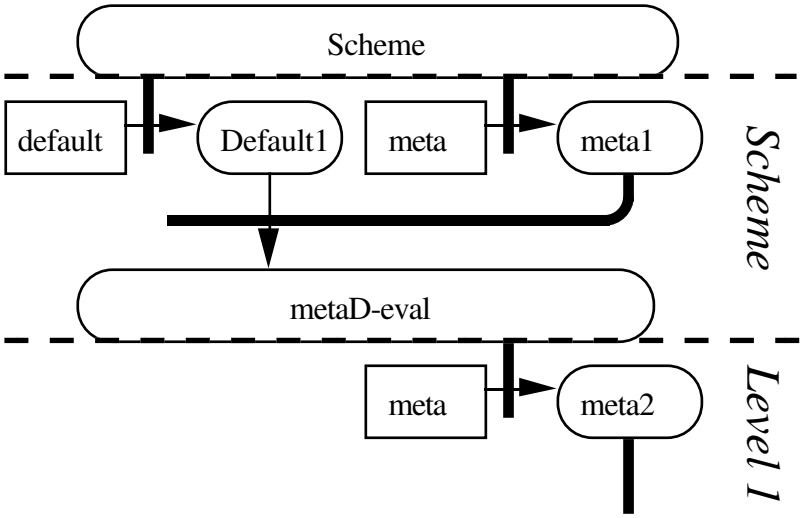
same language, Pascal, but it is not trivial to call procedures compiled in one of them and pass them arguments from a program compiled with the other one.

We call an open implementation that allows expressing such fix-point equations *an open implementation with reflective potential*. Our open implementation does not have reflective potential. The reason is that procedure representations of an engendered language differ from the procedure representation of the meta-language. The representation we have chosen for procedures in the engendered languages is a procedure of the meta-language with an extra continuation argument at the beginning. We will illustrate this with an example: the representation of the primitive addition procedure. In scheme (assuming Scheme is the meta-language) level this is simply represented by the native primitive addition procedure. In the engendered languages the representation will be a scheme procedure with a continuation argument:

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

Obviously this is quite different from the primitive **+** procedure. Similarly for any other procedure. The difficulty now is that *meta* is evaluated in Scheme and hence it expects a dispatcher with "normal"—i.e. without the continuation argument—Scheme procedures so it cannot handle procedures from the engendered languages.

It is a natural response to bootstrap meta not with plain Scheme, but with a version of metaD engendered by the meta implemented in Scheme. The picture below illustrates this:



It would appear that now *meta2* could be used with dispatchers from engendered languages because it is itself written in an engendered language. This is true, but there is a catch, it can indeed be used with dispatchers written in engendered languages, as long as they are engendered by *meta1*. Dispatcher written in languages engendered by *meta2* itself still cannot be used. Every level of bootstrapping introduces an extra continuation, thus a procedure representation at level 1—languages engendered by meta1—takes one extra continuation argument, and a procedure at level 2 takes 2 extra continuation arguments. Even if we bootstrap the tower 3 (or more) times, the incompatibility remains.

What we need is some kind of *meta* similar to the meta-circular one, but with the special property that it "appears" to be evaluated in the default evaluator engendered by itself. Of course, a description like this is hardly a definition. For this paper however we will not try to define formally the relationship between meta and its reflective alter ego—from now on called *meta**. In [De Volder,Steyaert94] a more formal and accurate definition of the relationship is given.

Our implementation of *meta** is inspired upon an infinitely bootstrapped *meta*. This reasoning is similar to the reasoning in traditional reflective towers such as the construction of 3-lisp [Smith84] where 3-lisp is defined as the limit of an infinitely ascending tower of meta-circular 3-lisp interpreters that is bootstrapped by a 2-lisp interpreter at the bottom of the tower.

We have built a "level shifting" implementation, based on the idea of a *meta* at the bottom of an infinite tower. 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[2] to the traditional implementation of a level shifting interpreter as previously described in [desRivières&Smith84]. The result is an implementation of *meta** 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. All of these infinite level procedure are represented equally as Scheme procedures with a stack of meta-continuations (instead of a single continuation) as hidden argument. Since all procedures—meta itself, dispatchers, evaluators, procedures in engendered languages—are represented the same way, meta* has reflective potential.

5.4 Fix-point Equations and Infinite Towers

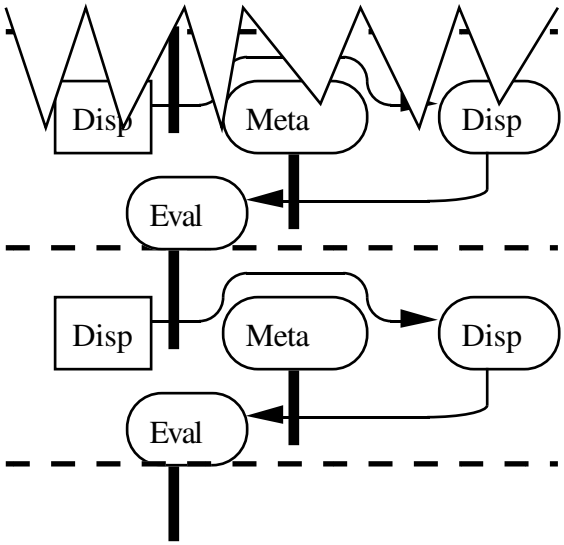
The equation $E = (M\ (E\ d))$ actually implies an infinite tower of open implementations. This can be seen when rewriting the equation into longer and longer equivalent equations as follows:

$$
\begin{aligned}
&E = (M\ (E\ d)) \\
&E = (M\ ((M\ (E\ d))\ d)) \\
&E = (M\ ((M\ ((M\ (E\ d))\ d))\ d)) \\
&\dots \\
&E = (M\ ((M\ ((M\ ((M\ ((M\ (\dots\ d))\ d))\ d))\ d))\ d))
\end{aligned}
$$

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.

---

2 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.



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. All rounded rectangles containing the word “meta” stand for one and the same meta-function (meta*), simply drawn multiple times. Similarly the rectangles and rounded rectangles representing dispatcher, dispatcher source code and evaluator are multiple drawings of the same objects.

6 Using meta*

Given *meta** we can use the fix-point equation to spawn an infinite tower of customised interpreters. The system we have implemented provides a tower of read-eval-print loops, in which the user can type in expressions. The evaluator used to evaluate the expressions is an instance of the default evaluator created by applying *meta** to the default-dispatcher. Every level of this tower is identical except for the input and output prompts which are numbered from -∞ up to 0. If computation at one level is terminated somehow (e.g. by not calling the continuation as in the previously given *exit* example) then computation falls back into the level above. The global environment contains a reference to *meta** in a variable called “meta*”. Thus the user can create his own dispatcher and pass it to *meta** 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 *climb* syntax is similar to *exit* but it allows exiting any arbitrary amount of levels at once. 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 metaD (a subset of scheme) are not “lazy” languages so we have to manually delay evaluation by wrapping lambda’s around some expressions.

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.

```
0> (define climb-dispatcher ;; (E d)
       (list (cons 'climb
                   (lambda (eval e r k) ;; wrapper lambda (delay)
                      (eval-climb climb-code global-env
                       (lambda (ED)
                          (ED eval e r k)))))))
0:
```

Subsequently we obtain the evaluator by giving the dispatcher created above as an argument to “meta*”.

```
0> (define eval-climb ;; E = (M (E D))
      (lambda (e r k)   ;; wrapper lambda (delay)
       ((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. For example we can climb 3 levels in the tower.

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

7 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 [Smith84]. 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**. It is up to the user to construct recursive definitions over evaluators for defining reflection or use meta just as any other normal open implementation. 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.

It can be argued that in our approach an open implementation serves to establish the meta-theory and the parameters to this open implementation are the self-representation of a system. Although our implementation 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 reflection etc. Normally one would determine some practical, sufficiently flexible ways for accessing the self-representation and hard-code these into the system. A straightforward way could be to build a system that provides access to its dispatcher table by storing it in the global environment of its engendered languages. In this case access to the self-representation can take place by referencing and assigning to this global variable. 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 programs that access the meta-interface to spawn new evaluators. These *rlet* and *rletrec* syntax use their bindings to specify a dispatcher table rather than an extension to the environment, thus providing a way to access the meta-interface.

The design of a suitable set of reflection operators—such as *rlet* or *rletrec*—is an important step in the specification of a reflective system. In this paper we do not intend to explore this issue but merely state that in the open implementation approach the design of the meta-interface and the design (and implementation) of reflection operators that provide user friendly access to the meta-interface are two orthogonal design issues. If both are well designed than the user should be able to add his own reflection operators by reflective programming.

8 Conclusion

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. In our example the *meta* function establishes the meta-theory and the parameters to the functions constitute the representation of a language/evaluator. 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 [DeVolder&Steyaert94]. 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.

9 References

[Bawden88] A. Bawden: **Reification without Evaluation**, Conference Record of the 1988 ACM Symposium on LISP and Functional Programming, pp 342-351, Smowbird, Utah (July 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)

[De Volder&Steyaert94] K. De Volder and P. Steyaert: **Towards a General Theory of Computational Reflection Based on Open Implementations and Linguistic Symbiosis**, Submitted to POPL '95, Brussels Free University, department of computer science (July 1994).

[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.

[Kickzales92] G. Kickzales: **Towards a New Model of Abstraction in Software Engineering**, in IMSA'92 International Workshop on Reflection and Meta-Level Architecture, pp 1-11, A. Yonezawa and B. C. Smith (Eds.), Tokyo, November 4-7, 1992. (1992)

[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. Open Implementation “with reflective potential” for fix-point equations

Source code is available from URL "ftp://progftp.vub.ac.be/tech_report/appendix-prog-tr-95-01.tar.Z"