A Two-Staged Introduction of Reflection: A Computational Characterisation of Reflection Based on Open Implementations

Patrick Steyaert[1]

Submitted to the OOPSLA'93 Workshop on
Object-Oriented Reflection and Metalevel Architectures

Abstract: In this paper we take an alternative view on the relation between reflection and extensibility in reflective programming languages. Rather than viewing extensibility as a consequence of a reflective architecture, we consider extensibility as an a priori condition for the definition of reflection. This is made concrete by a two staged introduction of reflection in a hypothetical programming language. The two stages are: 1) defining an open implementation for this programming language 2) making this open implementation explicit in this programming language. It is our premise that a two staged introduction of reflection will lead to better defined reflective architectures. The advantages are discussed.

1 Introduction

Reflective programming languages are praised for their extensibility. Extensibility, however, is not uniquely ascribed to reflective programming languages. Other techniques exist to make a programming language (or a system in general) extensible. For example, extensibility is a major issue in the development of object-oriented systems. What is typical for reflective programming, however, is that they allow a programming language to be extended from within itself, thus gaining a higher degree of 'self-containedness'.

Reflective programming languages acquire extensibility by allowing 'user code to effectively add new lines to the interpreter' [Smith84][Jefferson&Friedman92]. Major concerns in such endeavours are aspects that have to do with causal connection (see to it that the lines added to the interpreter are effectively used) and aspects that have to do with avoiding reflective overlap and providing 'vantage points' for reflective code [Smith84][Maes88] (see to it that user code that is 'part of the interpreter' is effectively run at the level of the interpreter).

Extensibility can also be achieved by appropriate parametrisation of a system. This is the basis for, for example, most, if not all, extensible object-oriented frameworks. Of course in such cases the kind of parametrisation can be of a sophisticated nature (e.g. it is typical for frameworks that there is a two way communication between framework

[1] Programming Technology Lab, Computer Science Department
Vije Universiteit Brussel — Pleinlaan 2 — B1050 Brussels BELGIUM
email:prsteyae@vnet3.vub.ac.be
tel. 32 2 641 34 92 or 32 2 641 33 08 fax. 32 2 641 34 95

and parameters; inheritance plays a crucial role also, and we will, for the time being take a simplistic view on inheritance as being a special form of parametrisation). A parametrised system is instantiated by supplying it with a set of chosen parameters. A parametrised system can be considered extensible when there exists a set of default parameters that can be overridden. Major concerns in this kind of endeavours are the determination of the parameters, and how parameters are supplied, but also how a system can be extended in an incremental way.

Programming languages can be parametrised. And in fact this is the basis for much of the reflective architectures that are currently being proposed (or meta-object protocol based architectures [Kickzales,des Rivières,Bobrow91]). Parametrised programming languages are a special form of systems with an open implementation [Rao91]. A system with an open implementation is a system that provides two interfaces to its clients. The base-level interface (evaluating programs) provides the standard functionalities similar to ordinary systems. The meta-level interface (supplying parameters) provides clients with a modular access to the implementation of the system. The meta-level language and base-level language need not be the same.

We propose a two-staged introduction of reflection. Firstly, the system under consideration must be given an open implementation (in which there is not necessarily a relation between meta-level and base-level language) thereby making it extensible, secondly, this system is turned into a reflective one, by making the parametrisation that is the basis for the open implementation, available in the system.

It is our opinion that this approach will lead to better defined reflective systems. This is due to the fact that major concerns about defining an extensible system and the typical concerns of reflection (i.e. avoiding reflective overlap) are separated. Typical for the approach is that there is a stronger basis for defining the meta-theory of a reflective system (it leads to a more formal basis for indicating how a programming language's interpreter is extended than the almost classical: 'adding lines to the interpreter'). Furthermore there is a stronger basis for exploring different variations on a given reflective architecture.

Extensibility is (in this paper) an a priori condition for reflection rather than a consequence of it. Reflection is introduced on top of a system that is already extensible; it is looked upon as a self-contained sort of parametrised system, i.e. the parametrisation is done from within the system itself. We will show, by means of an example, how, and under what conditions, a programming language with an open implementation, and with no relation between meta-level and base-level language, can be transformed into a reflective one. Hereby we effectively show how reflective programming languages are a special form of languages with an open implementation.

2 A Simple Example Language

As example language we will construct a simple extension to lambda calculus, called ASEL. The syntax is given below.

```
;   Expression =
;         (%apply Expression Expression*)
;         (%lambda Identifier* Expression)
;         (%if Expression Expression Expression)
;         (%Y Expression)
;         (%LAZY Expression)
;         (%let Identifier* Expression* Expression)
;         (%? Identifier)
;         (%# Primitive-Value)
```

ASEL supports multiple argument functions (un-curried functions). It extends Lambda calculus with the standard recursion operator[2], an operator that evaluates its argument in a lazy fashion, and an operator to locally bind names[3]. For ease of implementation, an identifier's value is retrieved with an explicit operator (`%?`). In a similar vein primitive (Scheme) values must be indicated by the `%#` operator. All special keywords in ASEL are preceded by a `%`-sign. The reason for this is that, given our goals, we want to have a clear distinction between ASEL-expressions and Scheme-expressions (expression of the underlying implementation language).

Obvious example programs can be written:

```
(%apply (%lambda '(x) (%? 'x)) (%# 3))  ->  3
(%apply (%# car) (%# '(a b c)))  -> a
```

3 A First Implementation

A fairly straightforward implementation of ASEL in Scheme can be given based on evaluation of expressions in contexts[4]. The evaluator function basically is a dispatcher on the type of expression to be evaluated. Lambdas (functions) are represented as closures. Closures are created with the `closure` function (defined locally to the interpreter). A closure contains the name of the formal parameters, the function body, and a reference to the lexically enclosing context of the lambda. It is represented as a Scheme function that applies to the actual arguments of the lambda.

```
--- an ASEL evaluator
(define basic-eval
  (let ((closure                    ---closure representation
         (lambda (ids e c)
           (lambda as
             ((Y (lambda (loop)
              (lambda (ids as c)
                (if (and  (null? ids) (null? as))
                  (basic-eval e c)
```

[2] `(%Y e) = (e (%Y e))`, note also that evaluation of the `%Y` operator proceeds in a lazy manner.

[3] `(%let '(x1 … xn) (e1 … en) b)`
`= (%apply (%lambda '(x1 … xn) b) e1 … en)`

[4] Continuations are omitted for clarity of presentation only. Although continuations are a challenging issue when introducing reflection, the techniques as presented in the paper are applicable also when continuations are involved.

```
                  (loop (cdr ids)
                        (cdr as)
                        (&extend c (&assoc (car ids) (car as))))))))
              ids as c)))))
     (lambda (e c)               ---evaluator dispatcher function
       (if (%apply? e) (apply (basic-eval (%apply-rator e) c)
                        ((Y (lambda (loop)
                              (lambda (es)
                                (if (null? es)
                                  '()
                                  (cons (basic-eval (car es) c)
                                        (loop (cdr es)))))))
                         (%apply-rands e)))
       (if (%lambda? e) (closure (%lambda-id e)
                                 (%lambda-body e) c)
       (if (%if? e) (if (basic-eval (%if-premise e) c)
                              (basic-eval (%if-t e) c)
                              (basic-eval (%if-f e) c))
       (if (%Y? e) (Y (lambda (s) ((basic-eval (%Y-f e) c) s)))
       (if (%LAZY? e) (LAZY (basic-eval (%LAZY-e e) c))
       (if (%let? e) (apply (closure (%let-ids e) (%let-b e) c)
                          ((Y (lambda (loop)
                                (lambda (es)
                                  (if (null? es)
                                    '()
                                    (cons (basic-eval (car es) c)
                                          (loop (cdr es)))))))
                           (%let-vs e)))
       (if (%?? e) (&find (%?-id e) c)
       (if (%#? e) (%#-v e)
       "no such expression"))))))))))
```

Notice that the evaluation function and the closure representation function are mutually recursive functions. The evaluator uses auxiliary functions `&extend`, `&assoc` , and `&find` to manipulate contexts, and auxiliary functions `%…?` (e.g. `%if?`), and `%…-…` (e.g. `%if-premise`), respectively for testing expression types, and accessing sub-expressions.

4 Stage 1: A Meta-Level Architecture for ASEL

The first stage in defining a reflective ASEL consists of building an open implementation. The purpose of this open implementation is that we want to make extensions to ASEL. Each such extension will define a variant to the basic ASEL language. Open implementations, in this context, will be referred to as meta-level architectures[5] since they consist of a base-level and a meta-level interface.

The sort of meta-level architectures that will be considered in this section are of a very specific nature, in the sense

[5] although the term meta-level architecture may have other connotations

that the meta-level interfaces will provide access to the implementation in a very specific way. The considered meta-level interfaces provide access to the implementation by allowing the installation of, so called, customisation parameters. When a meta-level architecture is given a full set of customisation parameters we will say that it is instantiated. We will say that a set of customisation parameters engenders some ASEL variant (the variant that is the result of instantiating the meta-level architecture with this set of customisation parameters).

Furthermore we will take into consideration meta-level architectures where the meta-level language is the underlying implementation language (i.e. Scheme). This is to say that, not only customisation parameters are expressed in the underlying implementation language, but also meta-level architecture instantiation (i.e. applying the meta-level architecture to the customisation parameters).

According to our intuitive idea of a meta-level architecture we first identify the user-level interface. In case of the above ASEL implementation this is a call to the interpreter to evaluate an expression to a value. Values are closures (functions), or primitive values (numbers, lists, …). The user-level interface is characterised by the following function type.

```
--- base-level interface for ASEL open implementations
EvalType = Expression x Context -> Value
```

A function that conforms to this type conforms to the user-level interface. There is one distinguished function that conforms to the user-level interface, called `basic-eval`, that is the standard ASEL implementation.

```
basic-eval: EvalType
```

We will simply model a meta-level architecture for ASEL, as any function that yields a function that conforms to the user-level interface. In general no restrictions are put on the type of customisation parameters with which the meta-function will be instantiated.

```
--- meta-level interface for ASEL open implementations
MetaType = MetaArgumentsType -> EvalType
```

This is a broad model[6], it can be used to construct sophisticated architectures. Different instances of meta-level architectures can be given[7]. First we present two degenerate meta-level architecture that constitute the boundary cases. Then we will present a meta-level architecture that allows clauses to be added to the basic ASEL interpreter. Although this latter is the single practical meta-level architecture that will be presented in detail, other

[6] However, it can be constrained so as to make sure that our initial ASEL language is part of it, by making sure that there exists some customisation parameter that instantiates the meta-level architecture to an evaluator that is equivalent to the basic ASEL evaluator. This constraint allows us to speak of a meta-level architecture “for ASEL”.

```
--- constraint such that ASEL is part of ASEL open implementations
∃ default ∈ MetaArgumentsType: meta(default) ≈ basic-eval
```

[7] The choices of parametrisation that are given are merely illustrative. They do not indicate a standpoint of what are practical or interesting parametrisations.

parametrisations are imaginable; e.g. a meta-level architecture that is the parametrisation of the ASEL evaluator with a closure representation type. A typical usage of this latter would be the tracing of function calls.

The first case is a meta-level architecture that is parametrised with an argument that is not used in an essential way (or not used at all).

```
--- "too much" degenerate meta-level architecture
DegenerateMetaType = String -> EvalType
degenerate-meta: DegenerateMetaType
with default: "no such expression error"

(define degenerate-meta
   (lambda (error-string)
      (lambda (e c)
        (if (%apply? e) …
        …
        (if (%#? e) …
        error-string))))))))
```

The second case is a meta-level architecture that is parametrised with an argument that is supposed to comprise the entire implementation, i.e. the meta-level architecture itself is void of implementation.

```
--- "too little" degenerate meta-level architecture
DegenerateMetaType = EvalType -> EvalType
degenerate-meta: DegenerateMetaType
with default: basic-eval

(define degenerate-meta
   (lambda (an-eval)
      (lambda (e c)
        (an-eval e c)))
```

A more useful example is an instance of a meta-level architecture where the evaluator can be extended by adding clauses to its dispatcher. This is reminiscent of reifier functions [Smith84] in e.g. 3-lisp (the differences will be discussed in section 6). The idea is to extend ASEL with new expression types. Expressions are identified by a tag. The evaluator dispatches over this tag. In the next meta-level architecture the evaluator is parametrised with a function that does additional dispatching whenever the evaluator encounters an unknown expression-type.

```
--- extending the dispatcher meta-level architecture
presume
  DispatcherType = Symbol -> (Expression* x Context) -> Value
then
  DispatcherMetaType = DispatcherType -> EvalType
  dispatcher-meta: DispatcherMetaType
  with default: (lambda (symbol)
                    (lambda (exps c) "no such expression error")
```

```
(define dispatcher-meta
   (lambda (dispatcher)
     (lambda (e c)
       (if (%apply? e) …
       …
       (if (%#? e) …
       ((dispatcher (%reifier-tag e)) (%reifier-args e) c))))))))))
```

ASEL syntax must be adapted so that expressions with an explicit tag can be constructed:

```
Expression =
     (%apply Expression Expression*)
     …
     (%r-apply Identifier Expression*)
```

The dispatcher meta-level architecture can be used, for example, to axiomatise lists in ASEL. Other usages similar to the use of reifier functions are possible.

```
--- using the reifiers meta-level architecture
(define list-reifiers-dispatcher
  (lambda (this-eval)
    (lambda (symb)
      (lambda (args c)
        (if (eq? symb 'list)
           (map (lambda (exp) (this-eval exp c)) args)
        (if (eq? symb 'car)
           (car (this-eval (car args) c))
        (if (eq? symb 'cdr)
           (cdr (this-eval (car args) c))
        "error")))))))

(define eval+list
  (Y (lambda (this-eval)
        (dispatcher-meta (list-reifiers-dispatcher this-eval)))))

(eval+list (%apply  (%lambda '(x) (%r-apply 'list (%? 'x) (%# 4)))
                    (%# 1)) '())
--> (1 4)
```

Notice the way the list-reifiers-dispatcher function is parametrised with an evaluation function. This is necessary for reifiers that need to evaluate their expression arguments. When the meta-level architecture is instantiated, the resulting evaluator is passed as argument to the list-reifiers-dispatcher function. Alternatively each reifier could define its own evaluator (constructed, of course, by application of the meta function) to evaluate its expression arguments. As such reifiers can introduce local ASEL variants for their sub-expressions. This is one of the more interesting aspects of this meta-level architecture. We will come back to this point when the relation with reifier functions is discussed.

5 Stage 2: Circularity Constraints for Reflection

In the previous paragraph we saw how an open ended ASEL was constructed. This was achieved by parametrising the implementation. Such a parametrisation was called a meta-level architecture. A meta-level architecture can be used to construct a set of varying evaluators by installing different customisation parameters.

The resulting architecture is not reflective. All parametrisation is done in the underlying implementation language (i.e. Scheme). In this section we will show how this meta-level architecture is turned into a reflective one. This is achieved by making the meta function, that is responsible for the instantiation of our meta-level architecture, explicit in ASEL.

Here again we are going to proceed in two steps. As a first step towards building a reflective architecture we will investigate what is needed so that the customisation parameters, provided to the meta-level architecture, can be expressed in ASEL (or some variant). Secondly we will investigate how the customisation itself can be expressed in ASEL.

Consider again the dispatcher meta-level architecture. Our aim is to write dispatcher functions and their contained reifiers in ASEL, and to be able to use them as arguments to the dispatcher-meta function; as follows:

```
(set! my-eval
  (dispatcher-meta (TRANSFORM (%lambda '(symb) (%lambda '(e c) …)))))
```

It is obvious that the dispatcher-meta function can not be applied directly to an argument that is an ASEL expression. It expects functions with a certain type as argument. ASEL expressions must first be transformed to appropriate functions before they can serve as customisation parameters. In general this transformation is essentially based on first evaluating its argument (which is an expression), in order to transform it into a function, and then transforming this function to the appropriate type. The following program text reflects this situation. The customisation parameter, expressed in ASEL, is evaluated prior to passing it to some glue function.

```
--- ASEL expression is evaluated prior to passing it to GLUE
(set! my-eval
  (dispatcher-meta
    (GLUE (meta-eval (%lambda '(symb) (%lambda '(e c) …))
                     meta-context))))

--- (trivial) glue to transform an ASEL function into a dispatcher
(define GLUE
   (lambda (ASEL-dispatcher-fun)      ; customisation parameter as
                                        ASEL function
      (lambda (symb)
        (lambda (e c)           ; return a dispatcher as Scheme function
          ((ASEL-dispatcher-fun symb) e c)))))
```

Two remarks must be made here. One remark has to do with the choice of the meta-eval function and the meta-context[8] used to evaluate the customisation parameters. This latter has to do with issues such as reflective overlap and will be considered later. The other remark has to do with under what circumstances a glue function can be constructed and what forms glue functions can take . This will be considered next.

Presume a meta-level architecture that expects customisation parameters of a certain type (e.g. the dispatcher-meta expects functions of type: `Symbol -> (Expression* x Context) -> Value`). A glue function can be constructed when we can find in the value domain (the domain of values to which expressions can evaluate) of our programming language (or system in general) a value type that conforms to or can be made to conform to the expected type of the customisation parameters.

In the case of the given meta-level architecture for ASEL, two aspects of the glue function must be considered. The first is what values in ASEL can be used to represent functions; the second is how symbols, expressions and contexts are passed to these functions (or how will these be represented as ASEL values). The first question has a straightforward answer, since lamda expression in ASEL evaluate to closures and closures are represented directly as functions in the underlying implementation language. For the time being symbols, expressions and contexts will be passed to ASEL programs (i.e. to the customisation parameters that are expressed in ASEL once they are installed) as is, so that the glue function for this architecture must do no transformation at all, and could have simply been omitted.

In general glue functions do need to transform their arguments after evaluation. For example, for a variant of ASEL that has only one-argument (curried) functions, the glue function could look like this (currying the arguments that are passed to the evaluated customisation parameter):

```
--- hypothetical glue for curried ASEL
(define GLUE
   (lambda (Curried-ASEL-dispatcher-fun)     ; customisation parameter
      (lambda (symb)                         ; return a dispatcher
        (lambda (e c)
          (((Curried-ASEL-dispatcher-fun symb) e) c)))))
```

Glue functions are an essential aspect of reflective architectures. They give considerable freedom in the choice of implementation language of the reflective architecture. In a terminology more related to reflective programming languages, glue functions define, and give considerable freedom in, the processes of deification and reification [Friedman&Wand84]. The glue function, in the above case, defines how dispatchers are deified (i.e. installed in the interpreter), and vice versa, it defines how expressions and contexts are reified[9] (i.e. how e.g. contexts are passed to

[8] and the choice of the meta continuation in an architecture where continuations are involved

[9] As an example of the importance of the freedom in the reification process, especially the freedom of choice of how expressions are reified is an important issue. Expressions can either be reified as datastructures that can be navigated (i.e. in which it is possible to get from an expression its sub-expressions), or as an abstract value that can only be used as an argument to the evaluator. This can be an important choice in determining for example whether sub-expression can be compiled (to e.g. a machine format) or not, or for compositionality issues.

dispatcher functions, that are expressed in ASEL, once they are installed).

The second observation that must be made regarding expressing customisation parameters in ASEL has to do with the choice of the (meta-) evaluator with which and the (meta-) context in which customisation parameters are evaluated.

As for the evaluator, three choices can be made to evaluate the customisation parameters. The basic ASEL evaluator can be used, such that customisation parameters are expressed in ASEL without extensions. Or, an evaluator can be used that was constructed earlier. Customisation parameters are then expressed in some extension of ASEL. This is a very similar situation to the first one (customisation parameters are expressed in some variant of ASEL). Finally, and most interestingly, we can recursively use the evaluator that is being engendered to evaluate the customisation parameters that are used to engender this evaluator.

```
--- reflection in the definition of customisation parameters
--- it should be noted that this example can be made to work
--- by lazy evaluating the evaluation of the customisation parameters
(define my-eval
  (Y (lambda (engendered-eval)
       (dispatcher-meta
         (GLUE (engendered-eval
                    (%lambda '(symb) (%lambda '(e c) …))
                    meta-context))))))
```

The above construction is reminiscent of a tower architecture. Indeed, customisation parameters are expressed in the ASEL variant they engender, similar to reifier functions that can recursively (or should we say reflectively) use themselves.

In conclusion we can say that it is possible to categorise meta-level architectures into three different categories that accord to three degrees of reflection. The first category (the category of *languages with an open implementation*) is the category in which customisation parameters are expressed in a totally <u>unrelated language</u> of the language they engender (in most cases the implementation language of the meta-level architecture). The second category (the category of *reflective programs with a statically determined number of reflection levels*) is the category in which customisation parameters are expressed in an <u>unrelated variant</u> of the language they engender (in most cases the basic variant of the programming language under consideration). The third category (the category of *reflective programs with a possibly dynamically determined number of reflection levels*) is the category in which customisation parameters are expressed in the language they engender. Although each of these categories is valuable, it is the possibility to express programs of the last category that is the necessary condition to call a language reflective.

It should be noted that the last step (true reflection) of our previous classification is not achievable for all meta-level architectures. This largely depends on the degree to which the customisation parameters play a crucial role in the meta-level architecture that is customised. Consider for example an ASEL meta-level architecture that is parametrised with how closures are represented. A closure representation is essentially a function (i.e. a lambda that evaluates to a closure). Any attempt to define closure representations in the language they engender immediately leads to infinite meta regression. So, closure representations can only be expressed in a language that uses an unrelated closure representation (e.g. basic ASEL).

Finally the choice must be made in what context the customisation parameters are evaluated (choice of the meta context). This choice is independent of the choice of the evaluator with which customisation parameters are to be evaluated. The choice of the meta context (and that of meta continuations in the case where continuations are relevant) is not unique to our approach. In fact this kind of choice lies at the hart of procedural reflection and tower architectures [Smith84]. We will just point out that two choices can be made regarding the meta context[10]. One choice that mimics the situation of the already existing meta-level architecture (i.e. one in which meta and object level context are separated[11]), and another choice that possibly involves the risk of reflective overlap (i.e. meta and object level context overlap). It is the merit of reflective languages with a tower architecture that they provide the means to avoid reflective overlap.

The above discussion can only be elucidated in a true reflective architecture. Up to now we only discussed how customisation parameters are expressed in ASEL; customisation itself (i.e. applying the meta function to the customisation parameters) is done in the underlying implementation language. Due to this fact there is not much choice for the meta context (unless the underlying implementation language allows reification of contexts there is no possibility to evaluate customisation parameters in the meta context). So, our next step will be to show how the open implementation is used in the construction of reflective ASEL.

ASEL is made reflective by giving ASEL programs access to the open implementation, i.e. more concretely by making the dispatcher meta function available to ASEL programs. This will simply be done by including the meta function in the top level context in which ASEL programs are evaluated. The meta function can be included in the top level context just as is because: 1) Scheme functions and ASEL functions are compatible and 2) due to the nature of our architecture, and the fact that we only consider contexts as arguments to the evaluator, meta and object level contexts do not overlap. In general, however, the meta function has to be transformed and involves calling glue functions.

ASEL is extended with a quoting mechanism so that object level programs can be manipulated. We will presume the existence of an ASEL evaluator that is extended with a quoting expression (with syntax : `(%quote`

[10] For the purpose of determining meta contexts this may seem trivial. In general (especially when continuations are involved) this is less trivial (see [Wand&Friedman88]).

[11] In the case where meta-level language and object level language are unrelated, meta context and object level context are necessarily different. The meta context typically contains Scheme definitions of e.g. the evaluator, the meta function, and the customisation parameters; the object level context typically contains ASEL definitions of user defined functions.

`Expression)`). A typical usage of the meta function in ASEL looks as in the following example. An ASEL program is constructed that defines a reifier to return the current context. An extended evaluator (my-eval) is constructed with this new reifier, and applied to some object level program. Note that the result is an empty context. This is due to the fact that the object level program is explicitly evaluated in the empty context (and not for example in the top-level context in which the dispatcher meta function is defined).

```
--- an ASEL program that defines a current-context reifier
--- presume that the following program is evaluated in the context:
---  (&extend '() (&assoc 'dispatcher-meta dispatcher-meta)))

(%let (my-eval)
       ((%apply (%? 'dispatcher-meta)
                 (%lambda '(symb)
                    (%lambda '(args c)
                       (%if (%apply (%# eq?) (%? 'symb)
                                             (%# 'current-context))
                            (%? 'c)
                            (%# "no such expression error")))))
     (%apply (%? 'my-eval)
             (%quote (%r-apply 'current-context)))
             (%# '())))
evaluates to: '()
```

The effect is that ASEL can be extended from within ASEL programs. The above program is one with a statically determined number of reflection levels. It is possible to construct reflective towers as shown in the appendix.

6 Evaluation of the Reflective Architecture

In lisp like languages, reflection is in most cases introduced by means of reifier functions. Reifier functions are functions that execute at the level of the interpreter, and hence have a similar formal parameter list as the interpreter (typically a list of expressions, the context, and optionally a continuation). Reifier functions are in most cases defined in a way that is similar to ordinary functions.

It is not immediately clear how the above architecture relates to reflective programming languages that employ reifier functions, in some form or another. Although we have shown that with the above architecture ASEL programs can extend the ASEL evaluator, and as such the above architecture has the same goal as the introduction of reifier functions, some objective differences can be listed. The most prominent distinction is the distinction in scope of definition of reifier functions on the one hand and dispatcher functions on the other hand. In contrast with reifier functions, the scope of a dispatcher function need not extend to sub-expressions (does not necessarily obey the lexical scoping, this follows from the remark at the end of section 4). Another difference is of a more intuitive nature. Dispatcher functions have the flavour of being defined ‘before’ the program is evaluated; i.e. they are passed to the meta function and from then on their definition remains fixed in the evaluator. A flavour that is not shared with reifier functions, that are of a more dynamic nature.

We argue that these differences have nothing to do with the way reflection is introduced, but rather has everything to do with the underlying meta-level architecture (even with the initial ASEL implementation).

The above mentioned intuitive difference between dispatcher functions and reifier functions can best be illustrated with an analogy. Consider constructing a reflective object oriented programming language, in which one can have explicit control over the way objects are represented. In analogy to the above this latter can be achieved in two ways.

We can statically define for a program (or sub-program) how objects must be represented. This conforms to using a meta function that has as input the way objects are represented, and as output an evaluator that uses this representation for all the objects that are created (unless an alternative evaluator is used for sub-programs).

Alternatively, a more object-oriented (and a more popular) approach could be followed. For each object a meta-object can be defined that indicates how this object is represented [Maes87]. This conforms to a more dynamic approach.

To support our argument that the underlying meta-level architecture lies at the hart of the difference between dispatcher functions and reifier functions, we will briefly discuss an alternative meta-level architecture for ASEL that is inspired by a more object oriented approach. We will start with a new implementation of the ASEL evaluator.

The essential idea is that expressions are represented as functions. Each expression-type is a function that, given a list of sub-expressions, returns a new expression. This expression is a function that, given a context (and, if relevant, a continuation), evaluates to its result.

```
--- example definitions of expression types
;;; definition of lambda expressions
(define %lambda
  (lambda (ids body)
    (lambda (c) (&closure ids body c))))
;;; definition of identifier look-up expressions
(define %?
  (lambda (id)
    (lambda (c) (&find id c))))
```

We will call, in obvious analogy, functions that define expression types, reifier functions. A program is constructed by applying reifier functions to sub-expressions. A so constructed program is evaluated by applying it to the top level context.

```
--- example program construction and evaluation
((%apply (%lambda '(x) (%? 'x)) (%# 3)) topcontext)
```

This implementation is, by its very nature, an open implementation; albeit one of a more general nature than the one previously discussed. Any function of the appropriate type can be used as a new sort of expression.

```
--- example usage of the meta-level architecture
--- %nop is a new sort of expression that evaluates its sub-expression
--- in the context
(let ((%nop (lambda (e) (lambda (c) (e c)))))
  ((%apply (%lambda '(x) (%nop (%? 'x))) (%# 3)) topcontext))
```

Once again this open implementation can be used as the basis of a reflective architecture, by giving user level programs access to the open implementation. As can be seen in the above example, the process of customisation (extension) is in this case introducing a reifier function in the meta context, and consequently using this newly defined reifier in the construction of a program. A reflective architecture based on making this sort of customisation available to ASEL programs, is very similar in nature to procedural reflective architectures based on reifier functions. Beneath the reader can find an example of how reflective programming looks like in such an architecture. We presume the existence of a %quote reifier that quotes its argument and introduces a new meta context, and the existence of an %r-apply reifier that looks up a user defined reifier in the meta context and applies it to its sub-expressions.

```
--- same example as above, but now the %nop reifier is defined in ASEL
--- reifiers are called with %r-apply
(%let (%nop) ((%lambda '(e) (%lambda '(c) (%apply (%? 'e) (%? 'c)))))
   (%apply (%quote (%apply (%lambda '(x)
                                  ((%r-apply (%? '%nop)) (%? 'x)))
                           (%# 3)))
           (%# topcontext)))
```

7 Related Work

The approach to reflection we propose is, in spirit, most closely related to the reflective architecture of Refci in [Simmons,Jefferson&Friedman92]. Refci is a Scheme dialect with a first class extensible interpreter. Although the end result in [Simmons,Jefferson&Friedman92] and in our work is similar in nature, the former work does not discuss the relation between open implementations and reflection and does not introduce reflection in a two staged way.

In [Simmons&Friedman92] it is argued that : "the extensibility of a reflective system is largely dependent on the choice of its underlying representations", and "requires forethought in the design stage of the kinds of extensions the user might wish to take". It is obvious that our work supports this view on reflection. Even more, in our work, this view is taken as the basic premise in constructing a reflective architecture.

8 Conclusions

We have shown how reflection can be introduced in a programming language in two stages. The first stage consists of making an open implementation. The second stage consists of giving programs, expressed in the programming language, a handle to this open implementation. Intuitively the advantage of this approach lies in the fact that the two essential aspects of reflective programming languages, extensibility and ‘self containdness’, and their respective key issues, providing modular access to the implementation and self reference, are introduced in an independent way. The practical result of this is a separation of design concerns while designing a reflective architecture.

The open implementation defines how and in what way a programming language is made extensible. It defines the sort of extensions that can be made to the programming language at hand.

The essence of reflection is how extensions can be made from within the programming language. The key issues are the choice of the meta evaluator (according to two degrees of reflection), the meta context (avoiding reflective overlap) and the transformations that are used in the deification and reification processes (giving considerable freedom in the implementation of the underlying architecture).

9 References

[des Rivières&Smith84] J. des Rivières and B. C. Smith: **The Implementation of Proceduraly 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)

[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)

10 Appendix

```
--- construction of a tower in reflective ASEL based on dispatchers
--- the result of this program is infinit meta regression
 (%let
   (recursive-reifier)
   ((%quote (%lambda '(symb)
               (%lambda '(args c)
                    (%if (%apply (%# eq?) (%? 'symb) (%# 'rec-reif))
                                 (%r-apply 'rec-reif (%? 'x))
                                 (%# "error"))))))
   (%let (eval)
         ((%Y (%lambda '(this-eval)
                   (%apply (%? 'meta)
                           (%LAZY (%apply (%? 'this-eval)
                                          (%? 'recursive-reifier)
                                          (%# '())))))))
       (%apply (%? 'eval)
               (%r-apply 'quote (%r-apply 'rec-reif (%? 'x)))
               (%# '()))))

--- definitions for defining reflective ASEL based on reifiers
(define %quote
  (lambda (ee)
    (lambda (quote-context)
      (lambda (call-context)
        (ee (shift-up quote-context (current-c call-context)))))))
(define %r-apply
  (lambda (reifier-exp)
    (lambda sub-exps
      (lambda (c)
        ((apply (reifier-exp (shift-down c)) sub-exps) c)))))
;;; override %lambda such that it quotes its body
(define %lambda
  (lambda (ids body)
    (lambda (c) (&closure ids ((%quote body) (shift-down c)) c))))

;;; context handling
(define shift-up
  (lambda (tower-of-c c) (cons c tower-of-c)))
(define shift-down
  (lambda (tower-of-c) (cdr tower-of-c)))
(define current-c
  (lambda (tower-of-c) (car tower-of-c)))
(define set-current-c
  (lambda (tower-of-c c) (cons c (cdr tower-of-c))))
```