Chapter 2

Computational Reflection and Open Systems

2.1 Introduction

*"**Reflection hypothesis:** In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures."*

Smith (1982)

The notion of reflection can be found in disciplines as diverse as philosophy, linguistics, logic and computer science. It is not clear how all these different notions of reflection are connected. Even in computer science the notion of reflection, mainly used in the disciplines of artificial intelligence and programming language design, has different connotations, and especially different motivations. The common theme is that of building computational systems that, in a substantial way, have access to, reason about, and act upon their own computational process.

Before plunging into the technical, or less technical, details of what a reflective system looks like and how to build it, we will first try to answer the question "why ?". Why is it necessary to build reflective systems ? Obviously, there is no need for reasoning about one's self if this doesn't increase one's capabilities for reasoning about one's subject domain. A full analysis of representation, efficiency, and reflection in the most general case is certainly beyond the scope of this dissertation, and would necessarily have to follow the analysis that can be found in [Smith86]. Rather than doing that, we analyse these concepts in the more restricted case of programming languages and computational systems described by programs expressed in a programming language.

For the particular case of programming languages, we will give an in-depth answer to the question why reflection is needed in section 2.4 and section 2.5. The notions of absorption and reification are introduced as two main factors in the need to design systems that open up their implementation. It is shown what it means for one system to access the implementational structures of another system. The notion of systems with an *open implementation*[1] is contrasted with the notion of systems that allow implementational access. *Reflective systems* are then introduced as a specific kind of systems with an open implementation.

While doing so, we will take a less conventional, more constructive, approach to reflective programming languages. It is a constructive introduction of reflection since we introduce reflection almost by saying how to construct a reflective system. It is less conventional because of the firm link that is made between reflection and systems with an open implementation. Rather than directly turning to the question of how a system can have access to, reason about and act upon its own internal structures, we first turn to the question of how one system can have access to, reason about and act upon another system. In particular we use the notion of open implementations where one system can inspect and manipulate the implementation of another system. It is our strong belief that this is an important step in the demystification of reflection.

We conclude this section with a discussion on the difference between systems with an open implementation and systems with an *open design*. We start with some assumptions and some terminology. Note that inevitably the terminology used, can differ from that of other author's.

2.2 Model of Computation

The assumption with which we start is that a *program* expressed in some programming language is turned into a *computational system* by means of another computational system commonly called the *meta-system* [Maes87]. For example, for a program this meta-system can take the form of an evaluator.

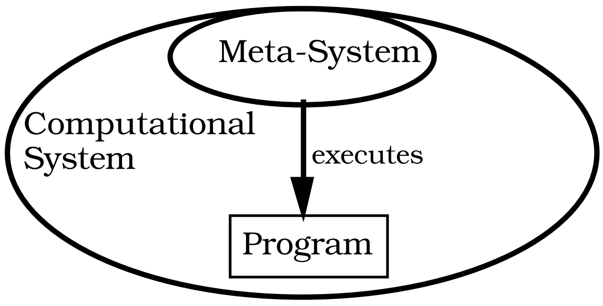

Figure 2.1

[1] It should be stressed that the notion of open implementations must not be confused with that of meta-systems [Maes87a] nor with meta-level architectures. See the section on open implementations for a discussion on the topic.

Although we find that for this special case, the term meta-system is somewhat misleading, it will be adopted in this text. In contrast with what would be expected, the meta-system is not a system that reasons, or acts upon another system, but rather it is a system that reasons about a *program* which is a *description* or *representation* of a computational system[2]. In the section on open implementations examples *will* be given of computational systems that *do* reason about other computational systems.

The meta-system is called a *language processor* in the case where the description of the computational system is a program expressed in some programming language. A language processor can take on different forms. In this text we will focus on evaluators.

A language processor itself can be composed of a program (the processor program) processed by another processor.

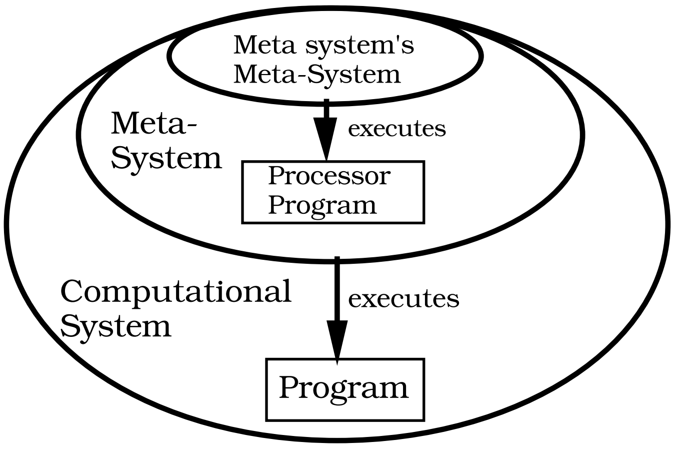

Figure 2.2

In the special case where the language in which the processor program is implemented, is the same as the language implemented by the processor program, the processor program is called *meta-circular* [3] [Abelson&Sussman84].

Programs that describe computational systems that manipulate other programs are often termed *meta-programs*. The program describing a programming environment is an example. The programs manipulated by meta-programs are called *object-level programs*; a relative notion, of course: object-level programs, in their own turn can be meta-programs.

The program of our meta-system is a special sort of meta-program. It is special since it is *the* meta-program that describes how to turn programs into computational systems. The architecture where the program of the meta-system is explicitly available for inspection and modification is a particular instance of a *meta-level architecture*. It has the advantage of being able to modify the meta-system prior to (or even during) the execution of a program.

---

2 It is the author' s conviction that much of the misunderstandings about reflective programming languages comes from a lack of distinction between computational systems and representations or descriptions of computational systems.

3 This process of decomposition can be repeated ad infinitum for meta-circular processor programs. In the literature about reflective systems this ' tower of meta-circular processors' is taken as the basis to introduce reflective programming languages, giving a slightly different view on reflection as it is introduced in this text. See [De Volder&Steyaert94] (also in appendix) for a discussion on the topic.

In general a meta-level architecture is an architecture where the meta-system can be acted upon. In the above case this is done by acting upon the description of the meta-system. Below we will see an example where this is done by acting directly upon the meta-system as is, i.e. acting upon it as a computational system rather than on its description.

Finally, we presume that a computational system has a well-defined interface, called the *base-level interface*, by which its behaviour can be invoked. For example the base-level interface of a language processor comprises the evaluation function.

2.3 Absorption and Reification in Programming Languages

The main reason for constructing reflective systems is efficiency and modularity in the structures used in representing a computational system. Typically only part of the system can be explicitly encoded in such a representation. A substantial part of the system' s behaviour remains implicit in the internal relations between elements of the representation, the process that interprets this representation and the circumstances in the world in which the system is embedded. This is not only an essential characteristic of such representations. It is also an integral part of being able to efficiently express the behaviour of computational systems, if we take a representation in which every aspect of the computational system must be explicitly encoded as an inefficient representation. On the other hand this characteristic puts limits on the generality and power of the underlying representational system. Not all systems can be expressed in an equally efficient way.

This may all seem to be in want of a more thorough explanation, and indeed it is. But, an in-depth analysis of representation, efficiency, and reflection in the most general case is certainly beyond the scope of this dissertation, and would necessarily have to follow the analysis that can be found in [Smith86]. Rather than doing that, we will analyse these concepts in the more restricted case of programming languages and computational systems described by programs expressed in a programming language.

Programming languages are used to describe *implementations* of computational systems. They do so by giving a means to express the internal workings of a computational system that is relatively close to executable code. What differentiates one programming language from another is how the internal workings are expressed. By this we don’t mean the syntactical differences between one programming language and another, but rather the notable differences of how a system is divided into subsystems. How a system is divided into subsystems is determined by the kinds of abstractions (e.g. procedural abstraction, data abstraction) or programming concepts [Maes87a] that are supported by the programming language.

In the discipline of programming language design, one speaks of programming paradigms as those classes of languages that support fundamentally different programming concepts [Wegner90]. The major programming paradigms are: procedural programming, object-oriented programming, concurrent programming, functional programming, logic programming and rule-based programming.

Programs expressed in exemplar programming languages of the different programming paradigms will exhibit different characteristics. They will differ essentially in what aspects of the internal workings of a computational system can be left implicit, and what aspects must be explicitly encoded. For example, it is obvious that for a backtracking problem (e.g. the 8-queens problem) expressed in a procedural programming language, the flow of control that is typical for backtracking must be explicitly encoded, whereas in an implementation in a logic programming language this can be left implicit.

In a program where a certain aspect of the internal workings of the implemented computational system is left implicit, we say that this aspect is *absorbed* (by the programming concepts of the programming language). When it is made explicit we say that it is *reified*. Efficiency, in terms of how concise a computational system can be expressed[4], is defined as the amount of detail that can be absorbed in the implementation language.

Obviously, it is not possible to give a total ordering of programming languages according to this kind of efficiency. Not just because we can only speak about efficiency for implementing a certain system (or a set of systems that belong to a particular problem domain if we are a bit liberal), but also because even within one system, conflicting demands with respect to the programming paradigm can coexist.

Notice that not all aspects of a computational system that can be absorbed in the implementation language also need to be absorbed. This is, for example, the basis on which different language interpreters (in most cases preferably meta-circular interpreters) are compared. A meta-circular interpreter for a Scheme-like language can choose to absorb or make explicit different aspects of the underlying structure of the Scheme language (see also [Abelson&Sussman84], [Maes87a]). The entire evaluation function can be absorbed by falling back on the meta-level programming facilities of Scheme, i.e. by using the explicit evaluation function of Scheme in Scheme. Or, the evaluation function can be implemented in terms of expressions and environments, thereby absorbing continuations and consequently the explicit encoding of the flow of control. Or, the evaluation function can be explicitly encoded with expressions, environments and continuations, but leaving implicit storage handling for lists. Or, an evaluation function can be constructed that makes explicit all machine actions performed by a hypothetical, or real processor. All these differences become relevant in case one wants to reason about or alter this implementation. As we will see in a moment, it is exactly these differences that will determine the theory with which we will be able to reason about our language implementation.

Within one and the same programming paradigm, also, differences exist between programming languages regarding their abilities to absorb implementation aspects of computational systems.

One set of examples are facilities such as garbage collection, persistency aspects of data, scoping issues, modularity etc. (in a mind boggling way, reflection itself can be added to this list, see also [Maes87a]). These facilities are generally considered as programming concepts that can, or should, be added orthogonal to most of the above programming paradigms. Languages that include these facilities have a larger capability to absorb implementation details.

4 We hesitate to use the term expressivity here. It is not clear whether expressivity, in its normal usage of "expressivity of a programming language" applies to the efficiency in expression or generality in expression.

Other more specific examples are (lack of) refinements of existing programming paradigms, or programming languages. We will discuss one example that is by now part of the folklore of object-oriented reflection (example from [Kiczales,des Rivières&Bobrow91]). Consider a computational system in which we need to represent data elements that are composed of other (named) data elements. In the object-oriented paradigm it is customary to implement such a data element as an object. The composition structure is reflected in the instance variables the object has. However, most object-oriented languages only provide facilities for representing objects with a small number of instance variables, all of which typically have a non-default value. Sometimes we need to implement a compound data element that has a possibly large number of components of which a large number has a default value for the major part of the object's life-time. Such a data element must be explicitly encoded as a dictionary for example. Only an object-oriented language that has the facility to represent objects with a large number of instance variables of which only a few have a non-default value, can absorb the implementation of this sort of data elements.

So, we observe that programming languages have different potential to absorb implementation details of a computational system, going from large grained programming paradigms, to more fine grained orthogonal sets of language features, to fine grained specific refinements of certain language features. Whereas the efficiency of programming paradigms is very hard to compare relative to each other, within one paradigm it is possible to compare the absorption capabilities of different language features.

One could be tempted to conclude that the more that can be absorbed by the programming language the better programs can be expressed, and thus that programming language design has as its goal the design of programming languages with ever better absorption capabilities. There is a catch however. It has the form of a trade-off between efficiency in expression, and generality of programs expressed in a programming language. Stated otherwise, the more that can be absorbed by the programming language the less general programs expressed in such a programming language tend to be. This is illustrated by the following example (example due to [Agha90]).

Consider writing a program that calculates the product of values that are stored in the leaf nodes of a tree. When expressed in a programming language that supports recursion, a substantial part of the control flow of this program can be absorbed by the programming language. The return stack of procedure calls can remain implicit. Such an encoding is more efficient than an encoding where we explicitly need to keep track of the visited nodes. It is less general, however, since we are unable to express, in a simple way, the fact that when a leaf node with the value ' 0' is encountered, the entire computation can stop and return the value ' 0' as a result. In an encoding with an explicit return stack, this *can* be encoded simply by emptying the control stack.

It is true that, in the above example, programs are not forced to use all the facilities (i.e. recursion) given by the programming language. Then again, if programs do not use such facilities out of fear of loss of generality, then why provide them ? One could also say that when such features are given, then all the complementary features to recover the loss of generality must be provided as well. For the above example this means that recursion must be complemented by a feature to 'jump out' of recursion. This, however, leads us to a (very old and often held) discussion on efficiency and generality on the level of programming languages, i.e. a small, concise programming language definition for a less general programming language versus a large, less concise programming language definition for a general programming language.

2.4 Open Implemented Computational Systems

So we seem to be stuck with an apparent contradiction between generality and efficiency. This need not be the case. What we truly wanted, in the above example, is a mechanism where the control stack can be left implicit until it is really needed. At that moment the control stack is made explicit, it is emptied and given back to the implementation to be absorbed. In general we need a mechanism where aspects that are absorbed in the underlying structures of the implementation language can, at any point in time, be made explicit, modified, and absorbed back again in the implementation of the programming language.

A mechanism is needed to inspect and alter the implementation structures of a programming language. Rather than tackling the question of how a program can inspect and alter the implementation structure of its own underlying implementation language, we will first tackle the question of what it means for one system to reason about the implementation of another system[5].

Computational systems, either programming language processors or other systems, that give access to their implementational structures are not new. Systems that provide, for example, facilities to test whether some extension of the system is available and how to use it, facilities for testing what version of the system is running, facilities for setting and testing parameters of internal data-structures (e.g. buffer-sizes, block-sizes, … in the area of operating systems) can be found in abundance.

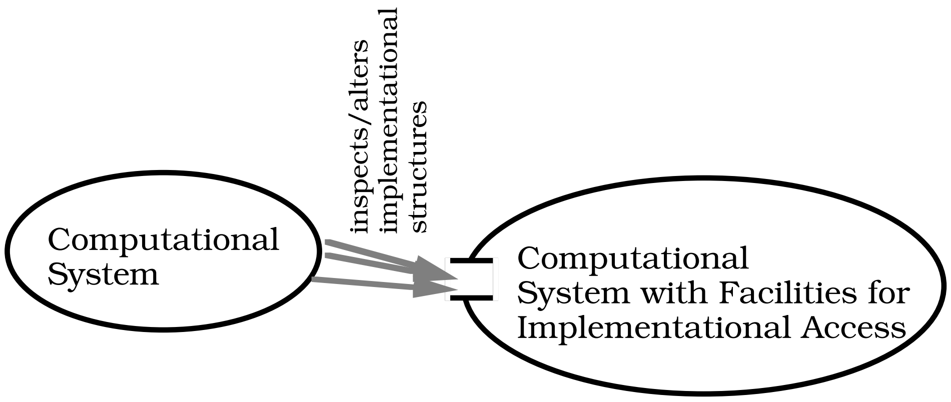
inspects/alters implementational structures
Computational System
Computational System with Facilities for Implementational Access

Figure 2.3

In the case of programming languages, access to implementational structures means that one can, for example, inspect the control stack or the variable binding environment, and that one is able to change these or hand back a modified version, so that the changes are reflected in the further execution of the program (i.e. in a causally connected way). Access takes the form of operations such as ' get-environment' , ' put-environment' that are defined for the language' s evaluator. These facilities are, in most cases, the basis for implementing debugging systems, or can even be put to use to partially solve the problems discussed in the previous section.

5 [Rao91] uses the term "implementational reflection" for inspecting and/or manipulating the implementational structures of other systems used by a program. We prefer to restrict usage of the term reflection to systems that reason about themselves.

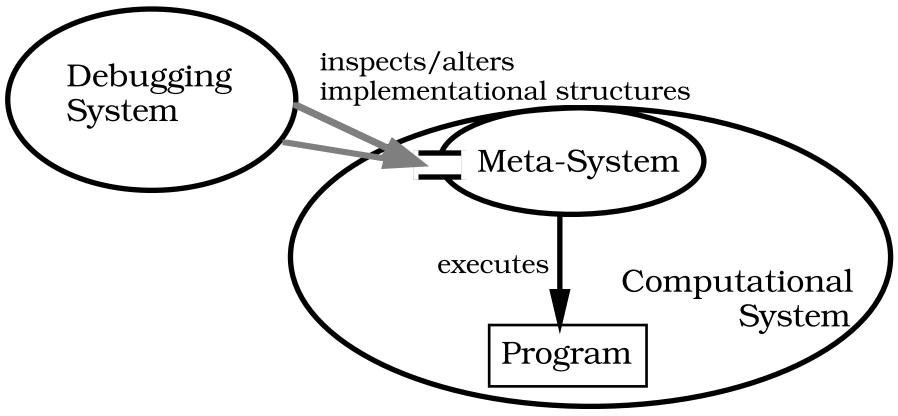

Figure 2.4

All of the above are limited cases of implementational access. First of all, it is not always clear whether such facilities are part of the ' ordinary' usage of the system, or whether they are ' special' in the sense of revealing part of the system' s implementation. Moreover, they do not solve all problems of the above section.

For example, in order to solve our problem of absorbing the representation of composite data elements with a large number of components, it is not sufficient to be able to inspect, and possibly change the implementational structures of an object. Rather, an alternative implementation of objects is needed. It is not sufficient to be able to inspect all the instance variables, nor is it sufficient to be able to add or delete instance variables. An instance variable, even if it has a default value, *is* an instance variable that must be represented in the object. What is needed is that, for this particular kind of objects, we can override the mechanism to look up instance variables.

In the general case, a more structured, ' open-ended' access to a system' s implementation[6] must be provided. Presuming that a computational system has an interface, the base-level interface, that shields its users from the implementation details that are involved in realising the system, and that is used by all users of the system, we can define the following:

> ***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. The notion of open implementations was introduced by Rao in [Rao91], where an open implementation is given of a windowing system that allows the exploration of different window system behaviours and implementations. The base-level interface of the windowing system is, obviously, an interface that allows the opening and closing of windows, dragging, generating pictures in windows, etc. The meta-level interface allows, for example, for the definition of new windowing relationships (such as window, sub-window relations).

[6] The difference between plain implementational access, and structured , open-ended access to a system' s implementation is parallel to the difference between reflective facilities and reflective architectures [Maes88].

A programming language with an open implementation will be called an *open implemented programming language*. A well-designed open implementation of respectively our object-oriented programming language and our recursion supporting language can, in principle, solve the respective problems of object representations and access to the control stack of the previous section.

Consider the problem of representing composite data elements with a large number of components. Any well-designed open implementation of an object-oriented programming language (the CLOS meta-object protocol is such an example [Kiczales,des Rivières&Bobrow91]) will provide a meta-level interface that allows alternative implementations for object representations. An object representation can be implemented in which only the instance variables with non-default values are stored.

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 object-level interface. To illustrate this we will consider two example open implementations.

Example 1: A Meta-function for a Scheme-like Language

An evaluator for a Scheme like language 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. 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, ….

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.

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.

Note that this open implementation implements many different variants of the Scheme programming language. Each particular usage of the meta-level interface *engenders* a different variant.

Example 2: A Class Hierarchy for a Scheme-like Language

Alternatively, a Scheme-like language can be implemented in the form of a class hierarchy in some object-oriented programming language. In this case expressions, lists, closures, and all other components of the evaluator are expressed as objects. To a certain degree the class hierarchy, to which all these objects belong, exposes aspects of the implementation of our evaluator. This has much to do with the often talked about code-reuse facilities that come with object-oriented programming.

To turn this class hierarchy into a true open implementation, however, we need to explicitly identify the base and the meta-level interface, and the link between both. In casu, the base-level interface will have the form of a protocol to which, for example, all objects representing expressions must conform, thereby establishing a contract between implementors of the classes that are used for instantiating ' expression objects' , and users of these objects (e.g. users that invoke the evaluator). The meta-level interface will be expressed as an interface with which new classes that implement expression objects can be added to the class hierarchy, or with which expression objects themselves can be added to a program representation such that they can be used in combination with the already existing expressions.

Not only the protocol of expression objects needs to be specified. All other objects that are part of the implementation may play an important role in the division between base and meta-level interface. The result of such an identification and specification of protocols is called a framework in object-oriented terminology; in the reflection community the term meta-object protocol is used.

This open implementation, just like the previous one, defines many different flavours of the Scheme programming language.

Both of the above open implementations give rise to the meta-level architecture as depicted in figure 2.5. In this meta-level architecture it is possible for a meta-program to act upon the meta-system prior to, or during execution of a program.

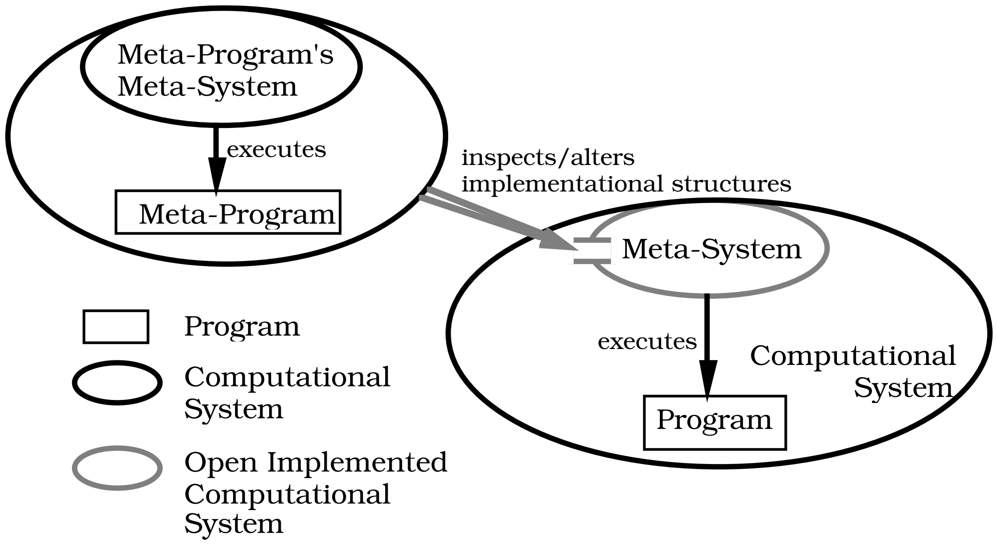
Meta-Program's Meta-System
executes
Meta-Program
inspects/alters implementational structures
Meta-System
executes
Computational System
Program
Program
Computational System
Open Implemented Computational System

Figure 2.5

What differentiates this architecture from 1) the debugging system of above and from 2) a meta-level architecture where *the program* of the meta-system can be acted upon, is the structured access it provides to the meta-system. In the first case access to the meta-system is too limited (as already shown above); in the second case access to the meta-system is too "unlimited"[7]. Especially the difference with this latter is important.

If modifications to the meta-system's program are allowed, then the result can be

[7] We can actually define a continuum with four marker points: a 'plain' evaluator without access to the implementation, an evaluator with implementational access, an open implemented evaluator and an explicitly encoded evaluator.

just about anything. It is the programmer that explicitly modifies the meta-system. And this is a matter of text-editing the meta-system's program-text.

When the meta-level architecture is based upon an open implementation, it actually is another computational system (admittedly, one programmed by the programmer, but still a separately identifiable computational system) that accesses and modifies the meta-system. It does so by using the meta-level interface to extend the meta-system with functions or objects (*not* program text but first class values !). The meta-level interfaces of the meta-system constrains the sort of modifications that can be done.

Finally, note that in the above meta-level architecture, the meta-program explicitly handles implementational structures of the meta-system used to execute a program, i.e. structures that are implicit for that program. So, what is explicit for the meta-program is implicit for the object-level program. As already mentioned before, a system' s implementation itself also absorbs and reifies certain aspects of the implemented system (cf. the different meta-circular interpreters of the previous section). This obviously puts a limit on what is possible with open implementations.

In conclusion we can say that by opening up the implementation of a computational system it is possible to have, to a certain degree, both efficiency in expression and generality. In order to open up the implementation we need to identify a base and a meta-level interface. The meta-level interface is used to alter and/or extend the behaviour of the system. An open implementation can be used to construct a particular kind of meta-level architecture, in which the meta-system can be modified in a controlled manner.

We considered the special case of *open implemented programming languages* i.e. programming languages that have an open implementation. For an open implemented programming language we observed that they implement not one but many different languages, according to how the meta-level interface is used. These are called the languages *engendered* by the open implementation.

2.5 Reflection: Accessing One's Own Meta-system

The above meta-level architectures have in common that one computational system acts upon the meta-system of another, in any other way unrelated, computational system. Meta-level architectures of this kind have their practical applications, even in the area of programming languages. For example in [Kiczales93] a CLOS open implementation for a Scheme compiler is briefly mentioned.

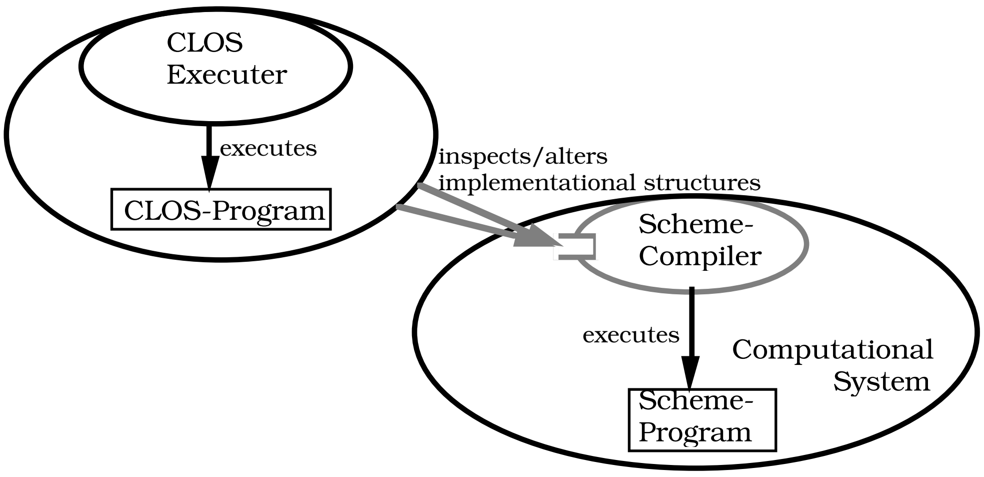

Figure 2.6

A more specific kind of architectures can be studied, i.e. that of *reflective systems*. Since a program is turned into a computational system by a meta-system, we can ask ourselves the question how and under what conditions a computational system described by some program processed by a meta-system can be given access to, use and alter the behaviour of its own meta-system (figure 2.7).

We will consider three forms of access to the meta-system: 1) access to the base-level interface of the meta-system, 2) implementational access to the meta-system, and 3) access to the meta-level interface of an open implemented meta-system. For the special case of programming languages the first will result in a special kind of meta-programming, the second in programming languages with *reflective facilities*, and the third in programming languages with a *reflective architecture*. The last architecture in this list being the most interesting one.

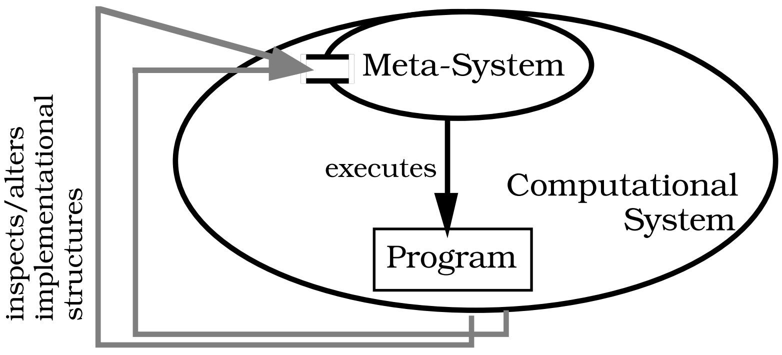

Figure 2.7

Not every open implementation is suitable as the basis for a reflective architecture. We will consider the conditions that must be met. This will lead us to the definition of *open implementations with reflective potential*.

The question how this access can be given can be answered in general. A program can be given access to its meta-system by extending the programming language with the necessary *reflection operators*. Reflection operators are language facilities, offered by the programming language, that allow programs to access the meta-system with which they are executed. A language that is extended with a set of reflection operators, can be called a *reflective programming language*.

A programming language must be *extended* with reflection operators. The special case can be identified where the open implementation of the programming language is powerful enough to formulate this extension. The open implementation of Agora that will be given will be of this kind. In all other cases reflection operators must be added to the programming language in an ad hoc fashion.

A program that uses reflection operators to access its own meta-system can be called a *reflective program.* A reflective program is both meta-program and object-level program at the same time. As mentioned before, the meta-program can explicitly handle implementational structures that are implicit for the object-level program. Collapsing meta-, and object-level program into one reflective program may lead to *reflective overlap*, i.e. implementational structures that are both explicit and implicit in the same expression. Reflective overlap is a phenomenon that can not be observed in ordinary meta-level architectures. Sometimes reflective overlap is undesirable. We will discuss a technique to manage reflective overlap.

An issue related to reflective overlap is that of *meta-regression*. A reflective program is said to *reflect* when it actually uses reflection operators to access its meta-system. The part of the program that reflects must, by definition, be a meta-program. On the other hand in a reflective program, not only the object-level program can reflect, but also the meta-program itself can reflect since it is executed by the same meta-system as the object-level program. This process of reflection can go on ad infinitum. If so, the program is said to *regress infinitely*. We will talk about *static reflection* when the maximum number of levels the program regresses can be statically determined. When the number of times the program regresses is dynamically determined, the program is said to exhibit *dynamic reflection*.

2.5.1 Reflective Architectures

If the meta-system has an open implementation, then a program processed by this meta-system can be given access to the meta-level interface of the meta-system with the intention of altering its behaviour. This gives rise to *reflective architectures*.

For example, in the case of the above "meta-function" open implementation of Scheme, access to the meta-function (i.e. to the meta-level interface) can be given under the form of reflection operators that allow the "installation" of extensions to the dispatcher (see [Simmons II&al.92] for an actual example).

The conditions under which access to the meta-level interface of an open implementation can be given, are reminiscent of, but fundamentally different from, meta-circular language processors.

In an open implemented 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, according to how the meta-level interface is programmed). We will call the former language the meta-level language of the open implementation, and the latter will be called the languages engendered by the open implementation. The meta-level language and the engendered languages need not be related. Even in practice examples can be found where it is advantageous to have a meta-level language that totally differs from the engendered languages. For example in the above CLOS open implementation for a

Scheme compiler these languages are different.

Since programs are processed by the open implementation, they are expressed in one of the engendered languages, and since the meta-level interface is coded from within such a program, a class of ' special' open implemented programming languages needs to be identified. This is the class of open implemented programming languages for which the meta-level can be programmed in *any* language engendered by this same open implementation. This class will be called the class of *open implemented programming languages with reflective potential*.

How can we construct such ' special' open implementations ? Notice that, in contrast with e.g. a plain evaluator, it is not possible to talk about a meta circularly implemented open implementation, exactly because an open implementation can be used to engender many different languages[8] (whereas a plain evaluator engenders only one). A meta circular open implementation would have to pick one engendered language as being preferred, excluding all the rest for programming the meta-level. For a reflective programming language this would mean that only the 'vanilla variant' of the programming language can be used for reflective programming, excluding all languages engendered by reflective programming, themselves, to be used for reflective programming. This latter ability, however, is considered as an essential characteristic of reflection [Smith82].

What is needed to construct an open implemented programming language with reflective potential is that all first class values (primitive values, functions, objects, …) can freely travel between implementation language and engendered language, and that both languages can transparently use each others first class values. Such a construction is called a *linguistic symbiosis* [Ichisugi&al.92] of the implementation language of the open implementation and possible engendered languages. A detailed description of such a construction for Agora will be given in a subsequent section.

2.5.2 Reflective Facilities

In the weaker case where the meta-system only allows access to its implementational structures (and is not a full-fledged open implementation), the system can only be extended with *reflective facilities*.

Reflective facilities usually take the form of two sets of operators. One set of operators to read the implementational structures of the meta-system. And one set of operators to overwrite the implementational structures of the meta-system. In the former case one speaks of *reification* [Friedman&Wand84]; in the latter the term *absorption*, or *deification* is used. Typical examples include reflective facilities to get access to and alter the variable binding environment or the control stack. See [Jagannathan&Agha92] for a fairly complex example of the usage of reflective facilities.

Reflective facilities often give rise to *reflective overlap*. Reflective overlap occurs when a part of the implementational structures of the meta-system is both reified (explicit) and absorbed (implicit) at the same time. Take for example a language with reflection operators ' get-environment' and ' put-environment' to reify and absorb environments (i.e. the reflective variants of the operators for implementational access from the previous section). Here, reflective overlap is most noticeable when environments are reified in a causally connected way, i.e. in a way such that modifications to the reified environment have an effect upon the

8 This is in fact a manifestation of what is called the causal connection requirement [Smith82].

executing program' s implicit environment. Stated otherwise, the environment that is made explicit is shared by the program in which it is made explicit and the meta-system. Environments that are reified are both explicit to the program and implicit in the meta-system. In such a case one speaks of reflective overlap. In our example this is apparent by the fact that the variable that holds the environment is also part of that environment.

This sort of reflective overlap can be disturbing. In the above example the programs that want to manipulate and alter environments must be careful not to destroy or alter their own variables for example. Reflective architectures often provide better mechanisms to control reflective overlap. It must be stressed, however, that not all cases of reflective overlap need to be avoided.

2.5.3 Meta-Programming

In practice programs are not executed by assembling expressions by hand, and then making an explicit call to the evaluator. Rather, an entire set of programs (or to be correct: computational systems) is available to construct and execute programs.

Programs that describe computational systems that manipulate other programs are often termed meta-programs. The program of our meta-system is such a meta-program. It is often desirable to construct one' s own meta-programs. Constructing a programming environment is an example.

To support meta-programming the base-level interface of the meta-system can be made available to programs executed by the meta-system. This enables the construction of meta-programs such that the processing of programs is absorbed, i.e. the language processor is not explicitly encoded. The difference[9] with the case where the language processor is explicitly encoded as a meta-circular processor, is that in that case we are using two *different* meta-systems; even though they have a possibly similar representation (i.e. one is meta-circularly defined in the other).

In our Scheme example access to the base-level interface, means that a Scheme program can explicitly invoke the underlying evaluator. A feature that is available in most Scheme implementations.

Finally note that in a reflective architecture, the code that is used in programming the meta-level interface is typically a meta-program, manipulating pieces of its own program, and applying the evaluator upon these program pieces. Meta-programming and reflective programming often go hand in hand.

[9] In a more general account of reflection this distinction would be made on the basis of causal connection, i.e. the difference would be made on the basis whether programs are executed by an executer program that is causally connected to the meta-system or executer programs that are not causally connected to the meta-system. Given our modest goals, our differentiation between the two is based on our consequent distinction between programs and computations.

2.6 Managing Reflective Overlap and Tower Architectures

What differentiates access to the meta system through reflection operators from 'ordinary' access to the meta-system is its dynamic character. The meta-system is accessed from within an executing program, and therefore meta-program and program possibly execute in the same execution environment. On the other hand, since meta-programs may actually reify and act upon this execution environment this may lead to reflective overlap. It is exactly this aspect that is the most difficult to manage in a practical setting.

A general recipe to avoid reflective overlap is closely connected to the notion of tower architectures [Smith82]. Conventionally, tower architectures are based on the notion of towers of meta-circular processors. Since our discussion on reflection is based on open implementations rather than meta-circular processors, we will briefly discuss the notion of towers of open implementations, and how they can be used to avoid reflective overlap. A more thorough discussion can be found in [De Volder&Steyaert94] (also in appendix).

The general idea is to provide a vantage point on which to stand when reasoning about one's own meta-system. Moreover, a vantage point that is similar in nature to the vantage point one system has when reasoning about another system (as in figure 2.5).

Reflective tower architectures mimic the fact that meta-programs execute in their own execution-environment. Actually, a tower of execution environments is needed rather than a single one, since in a true reflective system, not only is it possible to reflect on the meta-system of 'ordinary' programs, but also is it possible to reflect over the meta-system of meta-programs, and so on. What is actually mimicked is the infinite ascending chain of figure 2.8.

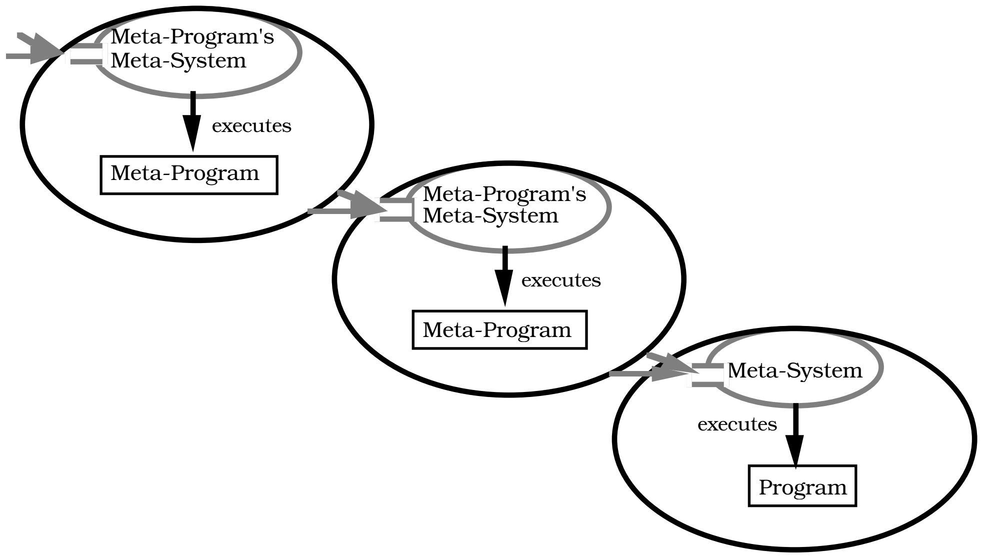

Figure 2.8

A couple of notes should be made here. The first is that in an actual implementation this infinite ascending chain can not be realised literally. In an actual implementation all meta-systems in the chain are one and the same (i.e. the open implementation with reflective potential), only a chain of execution states is realised. This chain of execution states conforms to those parts of the reflective program that can be identified as meta-programs. This leads us to the second remark. What is the nature of meta-programs, and how can they be identified, if they are part of the object-level program ?

In the conventional tower model meta-programs mostly take the form of meta-circular processor programs. In the tower model based on open implementations, meta-programs are generally programs that use the base and meta-level interface of the open implemented meta-system. A typical meta-program is a program that first uses the meta-level interface to alter the behaviour of the meta-system, and then uses the base-level interface of this altered meta-system to execute a program. Whether meta-programs can be identified as such depends on the exact nature of the reflection operators. It should be kept in mind however that in order to avoid reflective overlap, meta-programs should be distinguishable as separate entities in a reflective program. Examples will be given in later sections.

2.7 Computational Systems with an Open Design

In the previous section we saw that open implementations can be used as a basis to construct reflective systems. In this section we argue that for programming languages mere open implementations are not enough. We will analyse what is wrong with open implementations as a basis for safe and fully expressive reflective programming languages and introduce the improved concept of *open designs*. We will analyse the question of *abstract representations* in programming languages. The discussion on how to construct open designs is deferred till the section on object-oriented frameworks.

A computational system with an open implementation does not define a single system but an entire design space of (related) systems. The behaviour of a system with an open implementation can be altered through the meta-level interface. Although the meta-level interface puts constraints on the extent to which the behaviour of the system can be altered, merely opening up a system' s *implementation* gives no guarantee that the so created design space is coherent or is the intended design space.

For an open programming language, for example, we could want such a design space to cover all languages belonging to the same programming paradigm. Merely opening up the implementation of a particular programming language does not guarantee that the intended design space is covered, nor does it guarantee that no languages out of the intended design space can be reached. The former obviously is important, having to do with expressivity, but also the latter, having to do with safety and the ability to reason about programs.

Take for example the design space of pure functional languages. In opening up a particular programming language we need to make sure that we make explicit the *important* language concepts. In case of our functional programming language obviously what needs to be made explicit is the concept of a function. In an actual implementation of a functional programming language this notion need not be explicitly represented. The implementation is not necessarily a good source for finding the important language concepts.

Furthermore it is equally important to identify the *constraints* surrounding the language concepts that are made explicit. Take for example an open implementation of a pure functional programming language, where one can provide one' s own function representation. Here, an important constraint is that functions stay pure, i.e. that one does not introduce function representations that can be used to construct functions that are not referentially transparent. Certainly such constraints will not be found in an implementation, they may not even be enforceable by an implementation.

Finally, one must see to it that the concepts that are made explicit are represented *abstractly* enough. Or vice versa, that they are not represented too operationally. Take the above example again. Functions might be made explicit as closures, i.e. as a record with three fields: formal parameters, body and lexical context. Clearly closures are a representation of functions highly inspired by a particular implementation. Obviously such a representation does not conform to the truly abstract notion of a function, i.e. something that uniquely associates each input argument to an output argument. In case where the important language concepts are too operationally defined, it is possible that not the entire intended design space is covered.

A system with an open design differs exactly in the above points from a system with an open implementation. Rather than elaborating on these issues in general terms, we will explore them in more restricted settings.

The notion of abstractness will be explored in the particular setting of programming languages. We will see that in the area of programming language semantics, these issues have already been investigated, and that a set of objective criteria to test abstractness has already been developed for semantic definitions. These criteria will be adopted.

The questions how to make explicit the important design issues and what forms the constraints can take, will be discussed in the particular area of open object-oriented programs. As already said, this discussion is deferred to the section on object-oriented frameworks.

2.8 Full Abstraction and Compositionality in Programming Languages

"Programs are not text; they are hierarchical compositional structures and should be edited, executed and debugged in an environment that consistently acknowledges and reinforces this viewpoint."

Teitelbaum & Reps (1981)

"The meaning of a sentence must remain unchanged when a part of the sentence is replaced by an expression having the same meaning."

G. Frege (1892)

As said in the previous section, criteria are needed to test whether an implementation or design is defined abstract enough. For programming languages, and in particular in the domain of programming language semantics, such criteria have been defined. They are called full abstraction and compositionality. Full abstraction and compositionality are two complementary constraints. The latter ensures an abstract representation of expressions, the former an abstract representation of the first class values of the programming language. In this section we will discuss how these mechanisms can be adapted to the context of programming language implementations.

2.8.1 Full Abstraction and Compositionality in Semantics of Programming Languages

Although different kinds of semantic definitions exist for programming languages, it is possible to identify a common set of evaluation criteria. Semantic definitions are primarily judged by their soundness and completeness and the possibility to prove this. Any semantical description that lacks one of both is very questionable. Other evaluation criteria for semantic descriptions include formality, mathematical rigour and intelligibility. Although all of the former criteria are important, they are of lesser interest to us since they give no constructive indication on how semantical descriptions should be structured, and these criteria are not directly applicable to the implementation of programming languages. However, we will discuss two other criteria, compositionality [Frege92] [Tennent91] and full abstraction [Tennent91], that can be interpreted in the context of implementation of programming languages and can be used in a constructive way.

Throughout this discussion, and to illustrate it, we will adopt a denotational style of semantics. In general we define the semantics as a function that maps elements of the syntactic domain to the semantic domain. Each expression is mapped to its “meaning”.

```
μ: Syntactic Domain -> Semantic Domain
```

Compositionality
A semantic description is compositional if the meaning of composite expressions is expressed as a function of the meaning of its immediate subexpressions. For a compositionally defined semantics one can say that, in a composite expression, subexpressions can be substituted by semantically equivalent subexpressions without changing the meaning of the composite expression.

> ***Compositionality****: A semantic definition is compositional if two semantically equivalent expressions X and X′ (i.e.* $\mu(X) = \mu(X')$*); in each program context (…_…) where X can be used, X′ can be substituted such that:* $\mu(\ldots X\ldots) = \mu(\ldots X'\ldots)$*. [Tennent91]*

A compositionally defined semantics should not be confused with a semantics that is defined in a recursive compositional style (e.g. [Smith82]). In the latter the meaning of a composite expression is defined as a function of its immediate subexpressions, and not necessarily of the *meaning* of the subexpressions. Although in this latter kind of semantics the compositional nature is still an important issue, it does not have the above discussed property of substitutability of semantically equivalent expressions. An example of a semantics that is defined in a recursive compositional style but is not compositionally defined (example taken from [Smith82]) is the semantics of a Lisp-like language with a (one argument) quote expression, in which the meaning of this quote expression is the quoted expression. Clearly the semantics of the quote expression is not a function of the *meaning* of its subexpression.

Compositionality is crucial in proving properties of programs. It allows inductive reasoning about the structure of programs, i.e. to prove a property one proves the property for all non-composite (primitive) syntactic structures first, then the property can be inductively proven for each composite expression on the hypothesis that the property holds for the immediate subexpressions.

Full Abstraction

Semantic definitions can be classified according to how operational, or vice versa, how abstract they are. A "too operational" semantic definition is one that makes too much distinction in assigning meaning to expressions; abstractly equivalent expressions are assigned different meanings. An abstract semantics maps abstractly equivalent expressions to the same meaning.

> *A* ***fully abstract semantics*** *is a semantics that considers those expressions as equivalent that are indistinguishable in any program context. Consider two expressions X and X′. X and X′ are indistinguishable if for all program contexts (…_…) it can be observed that:* $\mu(\ldots X \ldots) = \mu(\ldots X' \ldots)$*. A semantic description is fully abstract if for all indistinguishable X and X′ it is true that* $\mu(X) = \mu(X')$*. [Tennent91]*

The difference between a semantics that is too operational and an abstract semantics is best illustrated with an example. Consider defining the semantics of an object-oriented programming language that supports strong encapsulation of objects.

On the programming level, strong encapsulation means that if we have two objects with the same behaviour, then these two objects are indistinguishable, according to our definition of indistinguishability above, regardless of how the behaviour of both objects is realised. For example two strongly encapsulated objects can have different private attributes and still be indistinguishable.

The semantic domain will consist mainly of a representation of objects. A semantic description where objects are, for example, represented as a couple (public methods, private instance variables), would be called too operational. Objects in this case can be considered semantically different on the basis of their private attributes. In contrast, a fully abstract semantics must consider all objects with the same behaviour as semantically equivalent.

The question of abstractness of the semantics boils down, in this case, to the question of how well the semantical representation of objects supports the notion of encapsulation. The question of whether such semantics exist remains open, however, and certainly lies beyond the scope of this work.

As Tennent [Tennent91] points out, the practical significance of full abstraction is that: if the semantics unnecessarily distinguishes the meaning of expression P and P′, an axiom asserting the equivalence of P and P′ could not be validated by the semantics, and an axiom asserting that P and P′ are *not* equivalent might incorrectly be regarded as sound. This means, as can also be observed in the above example, that the semantics is too fine-grained in distinguishing values.

Full abstraction and compositionality are two complementary constraints. The latter assures an abstract representation of expressions, the former an abstract representation of the semantic values.

2.8.2 Full Abstraction and Compositionality in Implementation of Programming Languages

The notions of full abstraction and compositionality are informally applicable to the implementation of programming languages. First remark that an implementation of a programming language in general involves more than the simple execution of programs. The different components, i.e. compiler, evaluator, program browser, program debugger, type checker, … of an entire programming environment must be taken into consideration. Although we will focus on the execution of programs, it is important to keep in mind that all that will be discussed is part of a larger whole, i.e. the programming environment.

There are two major mechanisms to execute a program. The first is pure interpretation, i.e. the program is executed by direct inspection of the internal representation of the program. The second is pure compilation, i.e. the program is translated to a form that is directly executable by the hardware. Hybrid forms exist, whereby a program is first translated to some intermediate program code which is then interpreted (by a virtual machine).

In this text we concentrate on pure interpretation. The issue of compilation of open programming languages is outside the scope of this dissertation and is left untouched. We refer the reader to [Asai,Matsuoka&Yonezawa93] [Ruf93] [Kiczales&Paepcke93] for this matter.

Compositionality
A programming language evaluator takes a program representation as input and processes it to generate a result. It is a recursive process over the program representation that generates the result of evaluating the program. In an abstract form, it can be looked upon as a procedure, possibly involving side effects, taking an expression argument and returning a result from some type of answers:

```
Eval: Expression -> Answer
```

The compositionality criterium, as defined for the semantics of programming languages, can be adopted for programming language evaluators, albeit in an informal way.

An evaluator is compositionally implemented if for each composite expression this evaluator is implemented by means of the application of the evaluator to its subexpressions and its result depends only on the result of the application of the evaluator to the subexpressions. The evaluator may not depend on any other properties of the subexpressions.

The role of compositionality in the implementation of evaluators is extensibility. An evaluator can be extended for each new expression type that is added to the programming language in an incremental way.

Full Abstraction
In implementations in general it is hard to devise a criterium for what is a more abstract implementation and what is a more concrete implementation. In the case of an evaluator, however, the notion of full abstraction is such a criterium. In analogy with semantic definitions, the notion of full abstraction can be used as a criterium for the implementation of evaluators. Moreover this notion conforms to the notion of abstractness in implementations in general; i.e. in a fully abstract interpreter implementation details of the values that are manipulated by that interpreter are hidden.

An evaluator is a recursive process over the representation of a program. The result of evaluation is a value of some data type of values. It can be represented abstractly as a function of expressions to values. In analogy with full abstraction in semantic definitions, an interpreter is fully abstract if indistinguishable expressions X1 and X2, evaluate to indistinguishable values. Indistinguishability of expressions has already been discussed. Whether two values are indistinguishable depends, of course, on how these values are expressed (e.g. as data types). In general however this amounts to simple equality or some sort of behavioural equality (all operations that are applicable on the value type give the same results on indistinguishable values).

The importance of full abstraction at the implementation level lies, again, in the incremental extensibility of the implemented system. An example is indicated.

Consider implementing a functional programming language. In the implementation of the evaluator the question arises how functions are going to be implemented. We can make two apparent choices. The first is to implement a function as a record with three fields, i.e. the formal parameter names, the body and the lexical context of the function. Or we can implement our language level functions directly as functions that are available at the implementation level, i.e. as a function that takes a list of actual arguments. The formal parameter names, the body and the lexical context are encapsulated in this function. Notice that no circularity is involved here.

The second implementation is fully abstract: at the programming level functions can only be distinguished by observing their effect on all possible arguments, at the implementation level also functions can only be distinguished by observing their effect on all possible arguments. The first implementation of functions is not fully abstract. Functions can be distinguished at the implementation level by comparing their formal parameter names, whereas at the programming level the formal parameter names have no effect on a function' s input-output behaviour.

The abstract implementation is more suited for extension. Consider adding a function type that is extensionally defined, i.e. a function is explicitly defined as a mapping of input values to output values. In the abstract implementation this function type can be added without modifying the implementation of function calling. In the non-abstract implementation the interpreter must be adapted to take into consideration this new function type.

It should be clear that the potential for constructing an abstractly implemented interpreter largely depends on the abstraction capabilities of the implementation language.

2.9 Conclusion

A program expressed in some programming language is turned into a computational system by means of a language processor (the meta-system). Not all aspects of the resulting computational system are explicit in the program. Some aspects are absorbed by the programming language concepts. We showed that due to this, there is trade-off between generality and efficiency in the description of computational systems.

Computational systems that allow implementational access try to improve on this trade-off by allowing a program to leave certain aspects implicit until they are needed. When needed, these aspects can be made explicit, modified and given back to the system to be absorbed. Computational systems that have an open implementation allow structured access to their implementation. They have an explicit meta-level interface by which a substantial part of the implementation, and accordingly, the behaviour of the system can be influenced. Language processors that have an open implementation can be used to construct a particular form of a meta-level architecture whereby the behaviour of the language processor can be customised prior to executing a program. Programs that alter the behaviour of a meta-system that executes a base-level program, are called meta-level programs.

Architectures in which programs can access their own executing meta-system have been studied. Different flavours of such architectures can be identified according to what aspect of the meta-system can be accessed. The particular kind where the meta-level interface of the meta-system can be accessed was called a reflective architecture. Open implementations with reflective potential allow the construction of reflective architectures.

Finally we discussed the difference between open implementations and open designs. The role of compositionality and full abstraction was discussed in the representation of aspects of programming languages.

In the next two chapters we will develop an open design for object-oriented programming languages. In chapter 5 we will come back to the issues of how to turn an open design into a reflective system.