Chapter 6

Related Work

6.1 Reflection and Open Systems

Open implementations were first discussed in [Rao90] [Rao91]. Rao defines what an open implementation is and describes an open implementation for a windowing system. Although Rao relates his work to what he calls implementational reflection, he does not give a thorough account of the relation between open implementations and reflective systems.

A more general introduction to the intuitions surrounding and motivations for investigating open implementations is given in [Kiczales92].

The most widely accepted account of computational reflection in programming languages was given by Smith in [Smith82] [Smith84]. The intuitions behind and motivations for the introduction of reflection are adopted in this text. Still, in Smith's and later Maes's [Maes87] account of reflection the notion of meta-circular interpreters plays an essential role. We motivated another approach to reflection where meta-circular interpreters are substituted by open implemented language processors.

In [Smith82] the notion of towers of meta-circular interpreters is used, and implementation techniques for such towers are discussed in [des Rivières&Smith84]. A comparison with towers of open implementations is given in the appendix.

6.2 Object-Oriented Reflection

3-KRS, ObjVlisp and Others

3-KRS [Maes87] is probably the first structured account of how to build an object-oriented programming language with a reflective architecture. Although the merits of object-orientation — encapsulation, modularity and abstraction are cited — are acknowledged for reflective programming, and although a serious attempt was made to be as complete and uniform as possible in the self-representation — 3-KRS has for example meta-objects for both data and program objects —, the reflective architecture of 3-KRS is highly inspired by one particular implementation of one particular object-oriented language (KRS).

3-KRS and Agora seem to have some commonalities in the choice of what must be represented in the self-representation, i.e. meta-objects, expressions, slots, etc. The important difference lies in *how* meta-objects, expressions, slots and so on are represented. As has been amply discussed in the previous chapters compositionality and full abstraction play an important role in Agora in the representation of objects and expressions. Equally important is the fact that for Agora the choice of what is made part of the self-representation and how the self-representation is structured is motivated by an in-depth study of the major design issues involved in designing an object-oriented programming language.

It should also be noted that since the connection between open implementations, frameworks and reflection had not been made at the time 3-KRS was being defined, 3-KRS does not have the layered structure of specialisation of language concepts as is the case for Agora.

ObjVlisp as presented in [Cointe87] discusses the introduction of reflection in a class-based language. It is based on a set of postulates, for example stating that the only protocol to activate an object is message passing. Still, it is easily shown that meta-classes alone are not enough to obtain a fully reflective programming language.

Other efforts are worth mentioning in this context. In [Ibrahim&Cummins88] a reflective programming language is developed with a strong emphasis on the representation of expressions. In [Jagannathan&Agha92] a reflective model of inheritance is presented. It is shown how, by reification of name-binding environments, reflection can be used in providing modularity structures in programming languages. In [Ferber89] various approaches to reflection in class-based languages are described. The approaches described vary in the different aspects of the computation that are reified. It should be mentioned that at least the reification of meta-objects and the reification of messages is covered by the reflection operators of Agora. Both [Mulet&Cointe93] and [Malenfant,Cointe&Dony91] study the issue of reflection in a prototype-based programming language. Although in some ways they have similar goals as ours — for example, defining inheritance or delegation as a specialisation —, but they do so by reifying meta-objects in a non abstract manner.

Finally, it should be noted that Agora's reifier expressions (i.e. reifier messages, receiverless reifier messages and reifier aggregates) are based on reifier functions from 3-Lisp [Smith84]. To the author's knowledge no other reflective object-oriented programming languages exist that are so heavily based on reifier expressions as Agora.

Metaobject Protocols

A major thrust in current day research on object-oriented reflection is directed towards metaobject protocols, of which the CLOS metaobject protocol (MOP) [Kiczales,des Rivières&Bobrow91] is the most prominent. Our work is related most to the work on MOPs. It shares the concerns of defining open languages by means of well-documented class hierarchies (e.g. [Kiczales&Lamping92]).

The CLOS MOP defines an open implementation for the Common Lisp Object System. This object system is heavily based on 'generic functions'. Accordingly the object model that is made explicit in the CLOS MOP is a model based on overloaded functions. This results in an entirely different view on object-orientation than the one adopted in this dissertation.

Although object-oriented software engineering practices — protocol design and documentation — play a prominent role in the work on metaobject protocols, it is difficult to say whether the CLOS MOP defines a true open design. Therefore a more thorough account of the design issues of object-oriented languages based on overloaded functions would be needed.

The differences, with our work, in handling reflection must also be pointed out. In [Kiczales,des Rivières&Bobrow91] the MOP is meta-circularly defined. This obviously leads to various circularities. These circularities are resolved in an actual implementation (behind the scenes). Resolving these circularities is an essential step in making a running reflective CLOS MOP. In our work we do not attempt to give a meta-circular definition of our framework. But, on the other hand, reflection is added as a full-fledged specialisation of the framework. The notion of a linguistic symbiosis is an essential ingredient in this process. We do not have a meta-circular definition of for example meta-objects, but what we do have is a way to make meta-objects explicitly referable (exactly this kind of differences was our motivation for making the terminological distinction between explicitly encoded and explicitly referable objects).

Linguistic Symbiosis of Object-Oriented Languages

The notion of a linguistic symbiosis between a programming language and its implementation language and its role in implementing a reflective programming language was first discussed in [Ichisugi&al.92]. There a description is given of a symbiosis, on the level of objects, between RbCl (a concurrent reflective object-oriented programming language) and C++ (the implementation language).

6.3 Object-Oriented Systems

The most important sources of inspiration for our analysis of object-oriented programming language concepts are [Wegner90] [Stein,Lieberman&Ungar89] [Dony,Malenfant&Cointe92] and [Cardelli88] [Cardelli&Wegner86] [Cook89] [Ghelli90] [Lieberman86] [Wegner&Zdonik88] for models of object-orientation and inheritance. For an alternative look on object-orientation based on overloaded functions the reader is referred to [Castagna&al.92] [Chambers92].

6.3.1 Object-Oriented Frameworks

In [Johnson&Foote88] and [Johnson&Russo91] a general description is given of the role of reuse and abstract classes in object-oriented programs. In [Johnson&Foote88] the emphasis is put on how to design (abstract) classes so that they become reusable. We extended the notion of abstract classes to include classes with virtual class attributes. In [Johnson&Russo91] the issue of how entire designs are reused is illustrated by means of a larger example. The distinction we made between a framework's internal and external interface comes originally from [Deutsch87]. In this dissertation object-oriented frameworks were studied in the context of open designs. We emphasised different kinds of specialisations of object-oriented frameworks that preserve the design of the framework. The relation between open implementations and object-oriented frameworks has already been noted in [Holland92]. In [Helm&al90] and [Holland92] a description is given of contracts — high level constructs for the specification of interactions among groups of objects — and how to refine and reuse contracts in a conforming way. We gave a more intuitive explanation of how to specialise a framework entirely based on substitutability [Liskov87] [Wegner&Zdonik88] of objects. Of course this can not substitute a formal description of specifying behaviour compositions in frameworks, but should rather be an intuitive basis for it.

6.3.2 Mixin-Based Inheritance

Our work on mixin-methods is an extension of mixin-based inheritance as was introduced in [Bracha&Cook90]. Mixin-based inheritance is an inheritance mechanism that is directly based on the model of inheritance as an incremental modification mechanism. It makes wrappers and wrapper application explicit.

Our work is based on a generalisation of mixin-based inheritance. Firstly, our mixins are based on a more general form of wrappers, where wrappers can have multiple parents. The notion of wrappers with multiple parents has already been pointed out in [Cook89]. Secondly, we extend the use of mixins to object-based inheritance. This sort of object-based inheritance is similar to implicit anticipated delegation [Stein,Lieberman,Ungar89], the resulting objects are comparable to split objects of [Dony,Malenfant,Cointe92]. Furthermore, we add dynamic application of mixins, mixins as attributes and the resulting scope rules for nested mixins. The extra polymorphism gained by viewing mixins as attributes seems to us an important enhancement to mixin-based inheritance. In contrast with [Bracha&Cook90] the mixin-methods used in Agora remain dynamically typed at the moment.

The relation to nested classes [Buhr&Zarnke88][Madsen87] has been discussed in the dissertation. The correspondence between so called virtual superclasses [Madsen&Møller-Pedersen89] in BETA and mixins has already been noted [Bracha92]. The same remarks as in the previous paragraph apply to the relation between mixin-methods and virtual superclasses.

Having mixins as instance attributes is very similar to “enhancements” described in [Hendler86]. We agree that being able to associate functionality with instances rather than classes has several advantages. The advantages of dynamic classification have also been discussed in the classifier approach of [Hamer92]. Both approaches lack the equivalent of late binding of mixin attributes. Our approach lacks the equivalent of having classifiers as first class values (which would amount to first class mixin “patterns”). Traces [Kiczales93] and first class mixin patterns are comparable mechanisms, they both are a step in solving the "make isn't generic" problem. First class mixin patterns are more general but also more primitive.

Our analysis of the problems involved in multiple inheritance is a résumé of [Snyder87], [Knudsen88] and [Carré,Geib90].

Although it is shown in [Bracha&Cook90] that mixin-based inheritance subsumes different inheritance strategies, no complete analysis is given on its relation with multiple inheritance. Other approaches that share our view of fragmenting the functionality of inheritance, can be found in [Bracha92], [Hauck93], and [Hense92]. Neither of them gives a complete analysis of how the multiple inheritance problems are addressed in it. In [Bracha92] name collisions are resolved with renaming. Neither of them supports object-based inheritance, which is an important part of our solution to multiple inheritance problems.