Chapter 7

Conclusion

In this dissertation we showed what it means for an object-oriented programming language to have an open design. The role of object-oriented frameworks, full abstraction and compositionality in this was shown. We discussed how frameworks can be used in expressing open designs. A framework for an object-based, and later on for an object-oriented programming language was presented. In this framework the major components of an object-oriented programming language were represented as abstract classes. A compositional and extensible representation of expressions and expression evaluation was presented. We showed that reifier expressions are an essential ingredient in making program representations extensible. Furthermore, a fully abstract object representation was discussed. This object representation was shown to represent encapsulated, polymorphic objects with a well-defined behaviour. It was shown that this particular fully abstract representation of objects could be specialised to include an object-based inheritance mechanism based on mixin-methods. This inheritance mechanism was shown to be a particularly useful one in constructing and controlling the evolution of multiple inheritance hierarchies. The resulting framework was used to express two important programming languages. The first language was based on a calculus for objects. The second was a full-fledged programming language with as sole built in construct that of message passing. It was shown that, while remaining in the constraints of the framework, fairly sophisticated extensions to this programming language could be expressed.

In a larger context we showed that reflective programming languages can be defined in a less ad hoc way than is usually the case. The importance of open designs in the context of defining reflective programming languages was made apparent. It was shown that opening up a programming language does not necessarily mean that all control about the language concepts, that are made explicit, is lost. The relation between open designs and reflective systems was discussed in general, and illustrated by making the above discussed framework reflective. In general terms we can say that open designs take up the role meta-circular interpreters conventionally play in reflective programming languages.

7.1 Contributions

The major contribution of this dissertation is showing that merely opening up an implementation of a programming language is not sufficient as a basis for defining expressive and safe reflective programming languages. The concept of open designs was proposed as an alternative. In particular an intuitive explanation of object-oriented frameworks and operations on object-oriented frameworks that preserve the major design issues was given. Furthermore, the importance of full abstraction and compositionality was discussed.

The second contribution is an alternative account of reflection based on open systems. It was shown how a computational system can be made reflective by opening up the meta-system, and by providing effective access to the resulting meta- and object-level interfaces of the meta-system. The properties of open systems with reflective potential have been discussed. A thorough account of a linguistic symbiosis — an essential ingredient in constructing an open system with reflective potential — has been given.

Apart from these general contributions more technical results were obtained.

The definition of a particular open design for object-based and object-oriented programming languages was given. This open design was based on respectively the notion of encapsulated polymorphic objects with a well-defined behaviour and mixin-methods. It was shown how a framework for object-oriented programming languages can be seen as a specialisation of a framework for object-based programming languages. The role of message passing was stressed, not only as a basic control structure, but also for making an extensible expression hierarchy and for making objects extensible. Reifier messages were used to make the expression hierarchy extensible. Mixin messages played an important role in extending objects.

A calculus for objects that features an explicit encapsulation operator and message passing as a primitive, atomic control structure was presented. Although not fully formalised yet, it provided a good basis for defining the basic structures of our framework.

A full fledged object-oriented programming language was presented (Agora). What distinguishes Agora from other object-oriented programming languages with a reflective architecture is its simple design. It features message passing as its only built in language construct. Furthermore a vanilla flavour of Agora was defined in the form of a standard set of reifier expressions. This standard set of reifiers was characterised by its usage of mixin-methods for inheritance.

Mixin-methods were shown to be an object-based inheritance mechanism that combines the advantages of class-based inheritance with the advantages of object-based delegation. It was shown that for an inheritance mechanism based on mixin-methods, inheritance can be entirely encapsulated. Motivated by a thorough analysis of the problems involved in constructing inheritance hierarchies a proposal was made for a generalisation of mixin-methods. It was shown that generalised mixin-methods can be used to express an entire range of multiple inheritance hierarchies in a simple and effective way. Furthermore it was shown how mixin-methods can be used in controlling the evolution of inheritance hierarchies.

7.2 Future work

Expressing Open Designs
In this dissertation we investigated the importance of open designs for reflective programming languages. The converse question, the importance of reflection in expressing open designs has been left open. As we saw in the section on object-oriented frameworks, the major design issues of an object-oriented framework are made explicit in the form of abstract classes. The programming language used to express a framework must support, for this reason, abstract classes as a language concept. In this text we stressed the importance of abstract attributes for this purpose. This is only one aspect of expressing object-oriented frameworks. Other aspects include information about the sharing structure of objects, information about the order of method invocations, information about the possible evolution of the class hierarchy — in the form of classifiers, for example — and many more. Obviously it is impossible to give a complete list of all such language concepts that must be provided, let alone to construct a single monolithic programming language that incorporates this list. Reflection and programming languages with an open design could prove to be an essential ingredient in realising and expressing open designs.

Open implementations, object-oriented frameworks, reflective systems are relatively young research areas. Formalisation of the major issues of each of them is an ongoing research topic. The work presented in this text is no exception to this. Although at some places formal techniques were used and hinted at, the main part of the work is in need of a more formal treatment. Formal semantics for conventional programming languages have been thoroughly studied. To a certain extent the formal semantics of reflective programming languages, also, has been investigated. Still, what is needed for a programming language with an open design is a formal description of a design space of programming languages. To the author's knowledge this issue has been left unexplored in the research community. What can be expected is that formal work on object-oriented frameworks will play an important role in this. However, formal descriptions of object-oriented frameworks too are still a hotly debated research topic.

A Model of Objects Based on Atomic Message Passing
In our analysis of object-orientation we came to the conclusion that a theory of object-orientation can be based upon the notion of encapsulated polymorphic objects with a well-defined behaviour. Subsequently a calculus was presented that featured objects of the above kind. Other such calculi for objects are being proposed in the literature. It remains to be investigated how our calculus relates to the others.

A Model of Inheritance Based on Mixin-Methods
Mixin-methods play an important role in this dissertation. As we already said, in the absence of mixin-methods, one has essentially two choices in picking an inheritance mechanism. Either class-based inheritance or classless delegation can be chosen. The one involves a notion different from objects, the other a notion different from normal message passing. Mixin-methods have been shown to solve this dilemma. Still, some work needs to be done. A more formal treatment of mixin-methods needs to be given. This could, for example, take the form of giving a denotational semantics of mixin-methods. Another remark is that mixin-methods are in some cases too limited. With mixin-methods an object must incorporate, in advance, all its extensions. Sometimes an object must be extended in a way that can not be predicted. Therefore mixin-methods must be generalised.

Implementation Issues

The Agora framework as presented in the text has been given a full implementation. This implementation is based on an interpretative approach. Furthermore efficiency issues are totally neglected. The problems involved in defining efficient implementations of reflective and open programming languages have been studied elsewhere. However, it needs to be investigated to what extent compositionality and full abstraction aid in optimising efficiency of reflective languages.