Chapter 11

Conclusion

The goal of this work has been the design of a "language with class", an object-oriented programming language supported by a formal theory of classification.

A formal account of object-oriented programming has been given, exposing several important misconceptions. The notions of "object", "type" and "class" have been defined. Different mathematical models of classification have been explored, including subtyping, bounded and F-bounded polymorphic inclusion. A theory of classification has been presented and extended to model families of types, type constructors and classes with internal polymorphism. A programming language has been designed to support important properties of objects and classes. Constructs in the language are given unambiguous translations in the formal model. A series of optimisation strategies for a compiler for object-oriented languages have also been discussed.

11.1 Summary of Findings

The work presented here has been an investigation into the nature of object-oriented programming languages. Starting with various informal accounts of objects, classes and types, a formal model of classification has been developed which accounts for the real operational behaviour of object-oriented languages. The model developed here has its forbears [Cook89a, CCHO89a, CCHO89b, CHC90, Harr91] and has been developed further to account for the whole spread of object-oriented concepts, including multiple and higher classification. The model is significant in that it departs largely from current popular understanding. In some points, the model contradicts the formal stance taken in published language descriptions. Previously, it had been contended [Meye89] that practical considerations must inevitably prevent an object-oriented language from being formally consistent. This view has been shown to

be over-pessimistic. A practical programming language has been designed, whose constructs have sound interpretations in the theoretical model presented here. The language covers a wide spectrum of object-oriented concepts, such as identity, encapsulated state, classification, single and multiple inheritance and polymorphism. In particular, it deals properly with the specialisation of recursive types, allowing the kind of flexibility desired by object-oriented programmers, but without making theoretical compromises.

11.1.1 A Survey of Practice and Theory

Object-oriented languages were shown to belong to one of five major programming paradigms; along with the functional languages they were shown to have a claim to preeminence as languages of choice when designing component-based software. Functional and object-oriented design strategies were compared; certain advantages were found for the object-oriented approach, which organises its components so as to encapsulate state and minimize system couplings. A representative selection of current analysis and design methods were evaluated; few were found to offer a convincing development process and only the behavioural approaches offered design techniques to minimize couplings between objects. A historical perspective on current popular object-oriented languages was given, highlighting the many different areas of computer science and software engineering that they have impacted, in both academic and commercial spheres. The overall impression is that object-oriented technology is useful, but still immature.

In particular, the formal understanding of language mechanisms was shown to be inadequate and inconsistent. Different notions of *class* were present in the literature; and technical terms like *polymorphism* were used in an incorrect sense. An exploration of types and subtyping [CW85] showed that, while it would be possible to construct an object-oriented language in which classes had the status of types and subclassing were subtyping, this seriously restricts the expressivity of the language [Cook89b]. In any case, few languages were found to satisfy the subtyping model and operational descriptions of languages tended to contradict this model, suggesting that a different model was required. One such model worthy of exploration was F-bounded quantification [Cook89a, CCHO89a].

11.1.2 New Comparative Techniques

A series of comparative techniques were developed, based on the λ-calculus model of objects as records of functions used by Wand, Cook and others [Wand87, Cook89a]. The subtyping model of Cardelli and Wegner [CW85] was first developed to the level of fully axiomatised types. This allowed the semantic aspects of languages with executable axioms, such as *Eiffel, Sather, OBJ* and *C++* (from version 4.0) to be analysed in the same framework as their syntactic aspects. The subtyping effects of adding or deleting axioms singly from type specifications was explored, in addition to the more conventional observations on adding or overriding functions. A potentially useful finding is that supertype axioms must be deliberately underspecified, if disjoint subtypes

are to be derived subsequently. While the effects of axioms were explored in a simple subtyping framework, they may legitimately be applied in second- and higher-order frameworks, such as those provided by F-bounded quantification.

A λ-calculus framework was developed, within which a naïve model of inheritance as record combination was contrasted with the simple subtyping model and finally with the F-bounded model. The key issue revealed by this comparative technique was the importance of *self*-reference in objects, which are in general recursive records of functions. Whereas the naïve model failed to maintain consistent *self*-reference, the phenomenon of *type-loss* in the subtyping model was shown to be due to the embedding of the supertype's *self* inside the subtype object. Fully consistent self-reference is only exhibited in the F-bounded model. *Self*-reference in the superclass and in the extension are redirected onto the child class. Formally, this is handled through the distribution of parameters standing for the *self* and *self*-type. This framework highlighted the importance of parameterising over *self* and the *self*-type in order to maintain the degree of flexibility required by object-oriented languages.

11.1.3 An Extended Theory of Classification.

This same observation motivated Cook's model of classes, described as second-order generalisations over recursive types, with associated implementations [Cook89a, CCHO89a]. A class is not a type C, but a family of types σ sharing similar recursive structure. The similarity is expressed by an F-bound: $\forall(\sigma \subseteq \Phi C[\sigma]).\sigma$, representing those types possessing a minimum set of functions. ΦC is the generator whose fixpoint is the recursive type C. Cook *et al*. developed the idea of extensible objects [CP89], types [CCHO89b] and object-constructors [CHC90] separately, although a unified treatment was anticipated in the unfinished *Abel* final report [Harr91] and is adopted in our theory here. The underlying theme in this work was that inheritance is best characterised as a transformation applied to *generators*, rather than to recursive types. The simple recursive structure of objects and types is recovered by taking the fixpoints of generators after these generators have been extended through inheritance. Classification is explained as a pointwise inclusion relationship between type generators; this is more satisfying than simple subtyping, since recursive types do not in general enter into proper subtyping relationships.

A serious restriction of Cook's model was that it relied on a simply-typed record combination operator. The use of inheritance was therefore limited to simple extensions to generators whose fixpoints were immediately taken. Such a model could only characterise single inheritance and could only handle polymorphism in the type of *self*. Here, a more general typing for record combination has been devised, based on the notion of *dependent second-order* types. This opens up Cook's model to allow the combination of class generators with extension generators, prior to the taking of fixpoints. The result of record combination is only well-typed if the two generators enter into a dependent second-order type relationship. This scheme extends the power of Cook's model to capture such notions as multiple inheritance and combination

with multiple, free-standing extensions, often known as *mixin*-based inheritance.

This theory was then further extended to support the notion of types with internal polymorphic components; and then classes of such types. Whereas the last unpublished work of the *Abel* project [Harr91] anticipated the use of subtype-bounded polymorphism for polymorphic components, in our approach the use of F-bounds has been systematically generalised. The fact that the whole type σ of *self* is dependent on the polymorphic type τ of a component is expressed using nested quantification: $\forall(\tau \subseteq \Phi P[\tau]).\forall(\sigma \subseteq \Phi W[\tau, \sigma]).\sigma$, where ΦW is a type generator expecting two parameters for the component-type and the *self*-type. Since this order of quantification makes σ dependent on τ, the recursive type W is necessarily homogeneous in whatever type instantiates τ. To create heterogeneous polymorphism requires reversing the order of quantification and this in turn requires a higher-order form of quantification: $\forall(\sigma$ **<:** $\Phi W[\sigma]).\forall(\tau \subseteq \Phi P[\tau]).\sigma[\tau]$. The operator **<:** stands for pointwise inclusion: $\forall(\sigma$ **<:** $\Phi W[\sigma]) \Leftrightarrow \forall(\sigma \mid \forall(\tau \subseteq \Phi P[\tau]).\sigma[\tau] \subseteq \Phi W[\sigma, \tau])$. In the higher-order form of quantification, σ ranges over type-functions rather than over dependent recursive types. This allows the application of σ to types other than τ, permitting the construction of recursive structures with heterogeneous component types.

Record combination with internal polymorphism in types other than the *self*-type requires in general a higher-order form of quantification. Accordingly, higher-order type rules have been provided for record combination with override $\oplus$ and for record merging with conflict-resolution $\otimes$. To distinguish the notion of inheritance, which is understood to mean a shorthand technique for defining an extended class with reference to some existing class, from the ordering relationships among groups of classes, our theory introduced and defined the technical terms: *simple classification*, *multiple classification* and *higher classification*.

11.1.4 A Pure Object-Oriented Language

The theory of classification presented above influenced the design of a practical object-oriented programming language. The aim here was to demonstrate that practical requirements need not mitigate against theoretical purity. The language supports value and reference semantics for objects, classification, single and multiple inheritance and a parametric polymorphism that is resolved either at compile-time or run-time. The language's concrete syntax distinguishes systematically between simple recursive types C, type constructors C[ ] and type generators, #C[ ]. A recursive type C is understood implicitly to be the fixpoint of a type generator #C[ ]. A generator having more than one type parameter yields a type constructor, or recursive type function C[ ] when its fixpoint is taken. Definitions are given in a parameterised polymorphic style, which may be fixed at the point of usage through the propagation of types into parameters, or adapted through the distribution of

new type parameters having different bounds. All this has a well-founded semantics in the theory of classification.

A major innovation in our language is in the treatment of all forms of polymorphism using a single F-bounded parametric scheme. The syntactic expression of F-bounds is implicit: #F[C] defines C as a type parameter constrained by the type generator #F[ ]. Interpreted in the theory of classification, C is understood to range over a class of types satisfying the F-bound $\forall(C \subseteq \Phi F[C])$. Our approach simplifies the expression of polymorphism in two ways. Firstly, the polymorphic typing of *self* and that of internal components is expressed in exactly the same way, using F-bounded parameters. In a class declaration, both *self* and any component may be given parameterised types. Secondly, no distinction is drawn between the syntactic expression of polymorphism that is intended for compile-time or run-time resolution. This means that parameterised structures can be used like type constructors in one context and run-time polymorphic types in other contexts. The difference is determined by the presence or absence of further static type information. It is the task of a compiler to detect the need for static or dynamic binding as a result of propagating static type information into parameterised structures. The parametric scheme is recursive, theoretically allowing an infinite embedding of polymorphic structures. To support the replacement of embedded type parameters, a nested parametric syntax is adopted, in which one polymorphic structure may be matched directly against another, and this is understood to propagate the relevant type information. The formal model translates this into the distribution of types and parameters to type functions expecting multiple arguments.

A second innovation in our language is the clean way in which it distinguishes value and reference semantics when passing objects as arguments. A special syntax marker @ was chosen to identify *alias types* (those passed by reference). Together with rules governing assignment, alias types provide a means of supporting the notion of object identity without violating object encapsulation. Furthermore, the use of alias types largely hides the underlying pointer techniques used to implement object references; this is a great simplification over languages like *C++*. Having both value and reference semantics allows the programmer to extend the basic set of simple values in addition to the larger constructed object types. Due concern was given to efficiency concerns, such that the consequences of a particular declaration style are immediately transparent in the underlying execution model.

A survey of flow-control techniques was also conducted. The chief finding of this investigation was that the acclaimed "pure" object-oriented style of flow-control using dynamic dispatch techniques [GR83] is in fact theoretically very complicated. In effect, by refusing to adopt a selection primitive that branches on simple values, *Smalltalk* passes the buck on to the type system. Since *Smalltalk* is not strongly typed, this aspect has not been noticed or considered before. To promote selection by dynamic binding, methods must be written using a style that introduces many more unresolved polymorphic parametric types than would otherwise be required. Against this, a simple selection

primitive takes the burden of value-based discrimination from the type system and allows recursive types to be constructed in a more natural manner. *Smalltalk*'s iteration strategy, which maps arbitrary closures over collections, was also examined and found to be even more complex. If all closures are properly constituted as methods, writing typed mapping methods proves almost intractable for practical purposes. A simple iteration primitive is to be preferred.

A compiler for the language was discussed. Several important procedures in the analysis and optimisation of a closed set of classes destined for a single application were discussed. These included the efficient processing of required source code files, the analysis and reduction of the inheritance graph, the construction and analysis of a call-graph to aid in the detection of static and dynamic binding and efficient code generation. The algorithms for collapsing the inheritance graph and binding methods were innovations; these were reported earlier in [SLN94]. The former reduces the data segment of programs by around 30% and the latter binds around 80% of methods statically. These figures are for typical medium-sized applications of around 50-100 classes arranged to exploit multiple inheritance.

11.2 In Support of the Thesis

The thesis contends that existing object-oriented languages have developed in advance of a formal theory of classification and, as a result, their treatment of *class* is muddled and ill-founded. The operational behaviour of such languages is often described incorrectly using a terminology of types which strictly does not apply. The type systems affected by such languages are incorrect; furthermore, they may contain redundant mechanisms for handling type polymorphism as a result of the central ambiguity surrounding the notion of *class*. To counter this confusion, the thesis maintains that a mathematically complete and consistent definition of classification is possible, which encompasses other approaches to type abstraction, such as "type constructors", "generic parameters", "classes", "inheritance" and "polymorphism". To demonstrate that such a theory is indeed tractable, it is exemplified in a practical programming language.

11.2.1 Combating Misconceptions

The popular fascination with objects as convenient computer representations of real world concepts diverts attention from the important formal distinction that an object is something with *identity*, *state* and *behaviour*. An object is distinguished from a pure functional value by virtue of its mutable state; and from a relational tuple by virtue of its identity, which is not dependent on its state. An object has an externally observable behaviour, determined by the methods that it supports. Object state is only accessible through its methods. It is these properties that allow objects to be used as components with encapsulated state in modular systems.

If the term *object* is misunderstood, the object-oriented literature is full of contradictions and misconceptions about the notion of *class*. In some treatments, objects are supposed to have *class* and *type* independently [Snyd87, Amer90, OMG91], in others the notions of *class* and *type* are identical [Meye88, SCBK86] or again, either of these views may be chosen at the programmer's discretion [Stro86, Stro91]. The notion of *class* is either relegated to an implementation concept or elevated to a specification concept, confused with *type*. This state of affairs has here been judged unsatisfactory. The type of an object is rightly understood as the set of signatures for its methods. This is consonant with other treatments of data abstraction. Since the behavioural response of an object is typically fixed once it is created (*pace Self* and languages with dynamic reclassification), it is correct to say that an object is an instance of a *type*, rather than an instance of a *class*. Since there exist potentially many other objects supporting more behavioural responses, these are rightly judged instances of different types. If it is intuitively considered that all these objects, by virtue of having a subset of behavioural responses in common, belong in the same *class*, then the notion of *class* is found to be something over and above the notion of *type*.

11.2.2 Defining the Meaning of Classification

This view is further supported by models of classification and inheritance. To investigate whether the notions of *class* and *type* might be considered identical, the Cardelli-Wegner model of types and subtyping [CW85] was explored. It transpires that subtyping imposes severe restrictions on the flexibility of an object-oriented type system. In particular, methods closed over the type of the current object cannot be redefined in subclasses without violating subtyping rules. Recursive types have no proper subtypes if they interact via a binary method with another object of the same type. This effectively fixes the types of methods at the point of declaration. In consequence, languages like *Trellis*, *Sather* and *C++*, which provide a subtyping approach to classification, are subject to type-loss whenever inherited methods are used. This does not promote strong typechecking and severly limits the utility of such an approach.

Operationally, languages like *Smalltalk* and *Eiffel* do not lose type information when the current object is returned (by reference) from an inherited method. Neither does *C++* where the return context is the dispatching site for a dynamically-bound *virtual* function. To capture this formally requires a parametric polymorphic model in which the *self*-type is first abstracted and later replaced by specific types. Cardelli and Wegner's bounded quantification [CW85] quantified over simple subtypes: $\forall(\tau \subseteq C).\tau$ and therefore proved too weak in the presence of type recursion. Canning, Cook, Hill, Olthoff and Mitchell's F-bounded quantification [CCHO89a] supplies exactly the constraint required to recapture the type of *self* in inherited methods. The approach works by virtue of including the type parameter on both sides of the subtype constraint: $\forall(\tau \subseteq \Phi C[\tau]).\tau$. This then defines class membership in terms of those recursive types which share a certain minimum number of operations. What is important is that none of the simply recursive types (*ie* those that are closed over their own *self*) in this family enter into proper subtyping

relationships with any other. The only subtyping relationships which exist are between recursive types τ and truncated versions that are not closed over their own *self*, $\Phi C[\tau]$. The latter appear in F-bounded type expressions and, in practice, only in those programming languages having a properly bound *super* construct.

Eiffel and *Smalltalk* are therefore incorrect to treat classes as types and subclassing as a kind of informal subtyping [Cook89b], since their language definitions follow an operational model which is better explained using F-bounds. *C++* has a triply ambiguous attitude to *class*, which in the context of *public* inheritance is treated as a type, in the context of *virtual* functions as an F-bound and in the context of *private* inheritance as an implementation mechanism only. A class is not a type and subclassing (rather than inheritance [CHC90]) is not subtyping. But neither is a class a mere mechanism for implementing extensible software modules. A class is a typed family of objects whose types and implementations exhibit respective similarities in recursive structure. Our theory of typed classes shows how an object's type is related in a straightforward way to its implementation. This view is more satisfying than existing dual definitions of *object types* and *object interfaces* [OMG91] because it links type and implementation in exactly the way that the conventional computer science literature links abstract and concrete data types.

Classes provide the necessary framework for comparing other kinds of type abstraction. Whereas a type is a monomorphic entity, a class is a polymorphic entity. Traditionally, polymorphism is indicated in *ML*, *Hope* and *Ada* by the presence of type parameters; but whereas in these languages the parameters are typically used to abstract over some internal component of a type, in formal models of object-oriented languages, the parameter abstracts over the whole type. Even if the concrete syntax of an object-oriented language does not have an explicit *self*-type parameter, one is necessary to explain the modified *self*-type in inherited methods. The universal parametric polymorphism used in conventional languages has been shown to be a special case of F-bounded parametric polymorphism. The latter may be used to explain the constraint on the polymorphic type of *self* in classification and to account for *Eiffel*'s *anchored types* and *constrained genericity* [Meye92]. Most recently, *where clauses* [Lisk95] have been rediscovered, expressing a constraint on a type in terms of the operations which it must possess; this is essentially a reworking of the earlier style of *Russell* [DDS78, DD79] and may be considered syntactic sugar for an F-bounded constraint. Type constructors, such as *Pascal*'s *Set of* and *Array of*, may also be described within the same framework. These are essentially recursive type functions with internal polymorphic components, exactly like the parametric-polymorphic lists of *ML* and *Hope*. A *higher class* with internal polymorphism is a generalisation over such type constructors in exactly the same way that a *simple class* is a generalisation over types. Higher classes provide a useful unifying concept, in that fixing their *self*-type yields a type constructor, whereas supplying all component-types yields a simple class. Higher classes permit the definition of extensible type constructors in the same way that simple classes provide for extensible types. Classification is a way of ordering all of these abstractions within the same mathematical framework.

11.2.3 Better Language Support for Classification

The language *Brunel* exemplifies the theory of classification. It is based on an F-bounded interpretation of classes, in which subclassing is not subtyping. Subclass compatibility is judged instead in terms of the replacement of type parameters. This allows polymorphic aliasing [Simo95] but correctly rules out type-unsafe method invocations as static type errors. This is a more habitable alternative than forcing a language to obey subtyping [Cook89b], since it yields a more sensitive analysis of type under inheritance. There is no need for unsafe downcasting techniques to recapture lost type information [Meys92], nor for patches to fix unsound type rules [Meye89, Meye95]. Because *Brunel* is based *wholly* on a different type system than previous object-oriented languages (*pace* the partial, implicit use of F-bounds in [Bruc94, BSG94, ESTZ94, EST95]), it is the first language that has succeeded in abolishing the false opposition between theoretical and practical concerns [Cook89b, Meye89]. In particular, it captures the intuitive notion held by object-oriented programmers that the argument and result types of polymorphic methods are subject to uniform specialisation, covarying with their owning class's type, as the class hierarchy is descended.

Brunel supports the open-ended style of programming desired by object-oriented programmers [Meye88] since definitions may always be given in an adaptable, parameterised way and types are only fixed at the point of use. This answers one of the serious criticisms of *C++*, which is that it forces programmers to anticipate which typed functions are going to be subject to polymorphic generalisation (using the *virtual* directive) and which are fixed in type. *C++* violates Meyer's open/closed principle [Meye88], since it is usually the case that programmers must return to completed library classes to change forms of declaration which were not originally intended for polymorphic use. In *Brunel*, a programmer should always anticipate polymorphic use. A key innovation in the design of *Brunel* is the way in which all remaining polymorphic type parameters may be fixed at their least upper bounds. Fixing is achieved implicitly by using the syntax for a simple type, instead of that for a class, at the point of use. This is better than forcing programmers to make early decisions about mono- and polymorphic types, something perpetuated in *Ada 95*'s class types [ABBD95].

It was the practical pressure to allow any typed declaration to be subsequently extensible that led *Eiffel* to blur the distinction between monomorphic and polymorphic types in its *conformance*-based polymorphism [Meye88, Meye89]. This was initially understood as a form of subtyping, but later proven unsound [Cook89b]. Unlike *Eiffel*, *Brunel* makes a clear distinction between mono- and polymorphic types in its syntax. Because of conformance, *Eiffel*'s ordinary types are all actually polymorphic; this significantly reduces the useful type constraints available in the language. In *Brunel*, it is possible to determine that an object identifier at a particular call-site has a fixed type, at which point all methods invoked on it may be bound statically.

To relieve programmers of the burden of redefining the types of inherited, recursive routines, *Eiffel* also provided *anchored types*, which were viewed

simply as a syntactic abbreviation for type redefinition [Meye89]. In fact, *anchored types*, especially in the form: *like Current*, are better explained in terms of F-bounded polymorphism [Simo95], since they recapture the inherited type of *self* in descendent classes. *Eiffel*'s *generic types* were provided as a way of expressing the polymorphism of type constructors. However, the addition of *constraints* in version 3.0 [Meye92] makes this indistinguishable from F-bounded polymorphism again. With *conformance*, *Eiffel* therefore has three different mechanisms for handling polymorphism where one would suffice. *Brunel* handles polymorphism in a much more elegant manner, providing a single parametric mechanism for both type construction and run-time polymorphism.

11.3 Further Work

No project is ever completely finished. Here, theoretical and practical aspects are highlighted, which merit further exploration and development.

11.3.1 Proof of Soundness, Completeness and Decidability

The theoretical model of classification and the syntax of *Brunel* are present in a completed form. While the concrete syntax is now reasonably stable, there is one area in which further modifications are anticipated; this is discussed below. One of the main outstanding theoretical tasks is to provide a complete set of type rules for all forms of expression *Brunel*, then give a proof of soundness and completeness. Such a task is not undertaken lightly, since the addition or subtraction of single rules must be explored for all possible effects. Languages with subtyping were found to have only semi-decidable type checking algorithms [Pier92b]. It is hoped that the elimination of subsumption in *Brunel*'s type system will enable the design of a completely decidable type checking algorithm.

11.3.2 Acceptance Testing for the Language Syntax

Certain aspects of *Brunel*'s syntax have not been presented widely to programmers before, in particular the nested parametric scheme. The model syntax given here is the result of several evolutions in the design of the language [Simo91, Simo93, Simo94a, SLN94, Simo95], in which the main changes have been to the way in which type information is propagated into parameters. It has been difficult to arrive at a scheme which is both succinct and intuitively clear to its users. In a personal communication, Meyer has indicated that he doubts whether programmers would adapt to a language having a large number of type parameters. With this in mind, other researchers have attempted to eliminate type parameters but obtain some of the same benefits by other means. This is an interesting area to explore; however it does not yield as expressive a type system as the fully parametric one adopted by *Brunel*.

Palsberg and Schwartzbach specify the types of a class's methods as though these were fixed types, but then have a *type substitution* rule in which one type is replaced by another throughout a single structure [PS94]. This provides a form of polymorphism, but has two drawbacks: it tends to confuse mono- and polymorphic types in the mind of the programmer; and it does not easily allow selective substitutions. A special fixing operator is required to specify that certain occurrences of a type are not to be substituted; and it is not possible to perform heterogeneous substitutions, as a fully parametric scheme would allow. Bruce *et al*. hide F-bounds behind *type matching* rules [Bruc94, BSG94]. In the parametric form of the matching rule: $\forall(\tau <\# C).\ x : \tau$, type functions are avoided by implicitly constructing the generator ΦC from the type C. The above expression may be considered equivalent to the F-bounded form: $\forall(\tau \subseteq \Phi C[\tau]).\ x : \tau$; although Bruce *et al*. are currently distancing themselves from F-bounded interpretations, preferring a pure syntactic approach to constructing type soundness and completeness theorems. In a recent personal communication, Bruce has indicated that a non-parametric form is also to be allowed: $x : \#C$, which is interpreted as an abbreviation for $\forall(\tau <\# C).\ x : \tau$. The non-parametric form loses the explicit type parameter and gives *x* the equivalent of *Brunel*'s unresolved polymorphic type, or *Ada 95*'s class type. This restricts the type system in certain ways; for example, arguments in the type #C may not be redefined, since this type is not linked parametrically to any other type about which stronger type assumptions may be made.

There is therefore a trade-off between superficial syntactic simplicity on the one hand and precision and expressiveness in the type system on the other. Bruce's work suggests that a simpler syntax may be possible for expressing unresolved, heterogeneous polymorphic types. *Brunel*'s unresolved polymorphic parameters allow for consistent substitution within single methods; however this extra degree of sensitivity may not be worth the cost of introducing multiple additional parameters for heterogeneous constructions. Apart from this, our current solution of matching one parameterised structure directly against another seems to be the simplest approach possible for a parametric language. Only time will tell whether this will win approval from programmers.

11.3.3 A Revised Compiler

At the time of writing, the full compiler for Brunel is out of step with the latest language definition. The last complete versions with code-generation handled the collapsing of the inheritance graph and detection of early static binding in a simpler type system [Low91, Tse91] and multiple inheritance was added later [Ng92]. A theoretical model which took into account the propagation of types into parameters was explored in full in [SLN94], but this did not generate executable code. Work on the compiler was suspended while alternative ways to express heterogeneous polymorphism were being explored. As another revision is anticipated here, this will delay the first complete parametric version of the compiler.

All versions of the *Brunel* compiler so far have performed a source translation to ANSII *C*, which is then compiled to object code using a standard *C* compiler. This has the advantage of providing the best possible static analysis of bindings for complete delivery systems, but the disadvantage of delaying all compilation tasks to the end of a project. We have indicated the need for a different underlying model for code generation and linking that would allow the early production of object code for individual *Brunel* classes. The key concern here is to provide single object code copies of methods which, by default, are bound dynamically, but where full static type information is available, may be linked statically. This sophisticated kind of linking is not available with current compiler technology; and is a matter for further research.