Chapter 10

Compiling a Language with Class

Here, issues of representation, binding and optimisation are examined with a view to compiling a "language with class".

Earlier chapters have introduced, in stages, the main features of a programming language that properly supports the notion of classification. All these ideas come together in an object-oriented programming language called "Brunel", named after the famous Victorian engineer. The "Brunel" project encompasses work in type theory, interface design and compilation.

Our exposition of classification concludes by examining how an object-oriented compiler generates structures to support run-time objects. Original techniques for optimising closed systems are highlighted, such as the filtering and collapsing of inheritance hierarchies and the automatic detection of safe early binding.

10.1 Introducing the Brunel Project

Much of the work presented here grew out of various designs for a new-generation object-oriented language coupled with a programming support environment, aimed at the software engineering industry [Simo91]. The language is named *Brunel*, after one of Britain's most famous and far-sighted engineers[1,2], many of whose engineering accomplishments from the Victorian era still fulfil their function today[3].

---

[1] Isambard Kingdom Brunel, more influential though sometimes less celebrated than Gustav Eiffel, invented more powerful screw-driven ships than contemporary paddle steamers; and a more stable wide-gauge rail network than that eventually adopted in Britain.

To answer the obvious question: 'Why yet another language?', the arguments of earlier chapters have already exposed sufficient problems and inconsistencies in the type systems of existing languages to warrant a better one. *Brunel* aims not only to improve the correctness and lucidity of object-oriented programming, but also to increase the level of run-time efficiency in object-oriented systems, all without sacrificing the open-ended flexibility that characterises the paradigm. This is a major undertaking, since it requires advances in language design, type checking algorithms, optimisation strategies and user-interfaces.

10.1.1 The Three Pillars

Brunel is designed to overcome perceived limitations in the correctness, efficiency and transparency of current object-oriented languages and development systems. The *Brunel* concept rests on three pillars:

- The type model for Brunel is elaborate, based on a higher-order typed λ-calculus in the tradition of [Cook89a, CCHO89b, CHC90, Harr91a], whose semantics ultimately derives from Category Theory [BW94, SG82, BCGS89]. In this model, classes are considered to be polymorphic types, expressed as the family of types satisfying an F-bound [CCHO89a], a generator which is a generalisation of a recursive type. Polymorphic types are represented using parameterised sets of function signatures and axioms, which may be replaced either at compile-time, in which case classes resemble conventional type-constructors [SC92, Simo93], or at run-time. The language is strongly and statically typed, having a polymorphic type checker which ensures signature and assignment conformity at compile-time and a future version aims to incorporate axiom verification at run-time.

- The compilation model for *Brunel* rests on the principle that a specific *delivery system*, meaning any given application, will typically require many fewer classes and so involve fewer inter-class dependencies than those expressed in the developer's class libraries. It requires a very high standard of system support for monitoring inter-class dependencies, incorporating an automatic loading and configuration tool [Low91, Tse91], an inheritance graph structure optimiser [SLN94], an automatic procedure for determining static or dynamic binding and an optimisation process for replacing unnecessary nested method calls by inline expressions [CUL89, CU90]. One advantage of having a single parametric treatment of polymorphism is that type information is propagated into structures and therefore *Brunel*

---

[2] We have studiously avoided other tower-related acronyms, such as "*Blackpool*", despite the obvious appeal of -*OOL* and a certain tradition started by *Eiffel* and *Sather*.

[3] Such as the breathtakingly beautiful Saltash Bridge over the River Tamar, linking the counties of Devon and Cornwall by rail.

supports a much finer analysis of bindings than is possible with current languages. An important ancilliary requirement is that compilers for different platforms should generate semantically equivalent code: we desire equivalent representations of basic types such as floating-point and integer numbers, supporting an agreed standard of precision across platforms.

- The development environment for *Brunel* provides a graphical interface to class systems and subsystems in the tradition of the best object-oriented browsing environments [Gold85, BS83]. The graphical user interface will conform to industry standards of the day for presentation and portability. An early version [Ong91, Tam92] was implemented in *OpenWindows*, following the then recent trend in standardising different proprietary X-Windows based interfaces. Unlike many CASE tools which are restricted to the generation of design documents and code stubs, the Brunel environment aims to have a full reverse-engineering capability, generating abstract views and interactive diagrams from source [Will95]. At the time of writing, our strategy for storing, parsing and visualising *Brunel* source code is evolving. There is a move away from a form-filling approach [SLN94] with a text preprocessor, towards full syntax-directed editing [MBDF90, Mads93] which saves parse-trees to disk and therefore has to reconstruct the appearance of source code for the programmer.

Earlier chapters have described the progress achieved so far mainly in the first pillar, on which many other aspects of the *Brunel* project will depend. In the rest of this chapter, aspects of the second pillar are described. The storage and binding optimisations described here were previously reported in [SLN94]. The emphasis of the current presentation is to highlight the interactions between types, binding and optimisation.

10.1.2 Development History[4]

The earliest version of *Brunel* was designed mainly as a test-bed to investigate certain compiling and optimisation strategies [Simo91, Low91, Tse91]. This version supported a simple interpretation of classes-as-types and enforced strict subtyping. It offered only single inheritance and had conventional control structures. Two subsequent implementations experimented with the addition of multiple inheritance using an adapted *C++* approach [Ng92, Stro87]; and higher-order functions, to support *Smalltalk*-like control and branching as dynamic dispatching on boolean objects [Blac92, GR83]. The earliest development environment [Ong91] was extended to support the automatic layout of multiple inheritance graphs [Tam92].

Meanwhile, the differences between *type* and *class* described in chapter 4 were first appreciated [SC92] then developed into different forms of concrete syntax supporting a single F-bounded parametric approach to polymorphism [Simo93,

---

[4] I am grateful to the following students who have worked under my supervision on the *Brunel* project: Low Eng-Kwang, Tse Hau-Pui, Ong Pang-Siong, David Black, Ng Yee-Mei, Rex Tam and Mark Williams.

Simo94a]. Type parameters were later added to the *Brunel* compilation model [SLN94], at which point *Brunel*'s finer-grained type analysis started to bring significant gains over the *Eiffel* and *C++* compilation models. A rationale for incorporating axioms into the type model was proposed later [Simo94b].

While the syntax of *Brunel* was evolving, a more advanced environment supporting the layout of lattice structures in both the inheritance and client-supplier dimensions was developed for *Eiffel* [Will95]. This has a full reverse-engineering capability, offering graphical interaction with third-party software libraries, thanks to a parser that builds a dependency model from source files. It offers layered views of a project under development, to control complexity. The analysis of object-oriented polymorphism described in chapters 5, 6 and 8 has also been applied to *Eiffel* and used to suggest several modifications to *Eiffel* syntax [Simo95].

The optimising techniques reported in [SLN94] will be of a wider interest to designers of development systems and compilers for other strongly-typed object-oriented languages, such as *Eiffel*, *Trellis* and *C++*. In many cases, similar advantages may be obtained for these languages. However, the design of the *Brunel* language, particularly its type system, and its close coupling with the development environment bring certain benefits which may not be realised fully in other languages.

10.2 Compiling Object-Oriented Languages

Object-oriented languages present significantly greater challenges to compiler-writers than other structured languages. Developing a compiler which will generate a run-time system with precisely the same semantics as that expressed in the design of the class hierarchy may take up to four or five times the effort invested in developing a standard one-pass compiler for a block-structured language like *Pascal* [Howa93]. Indeed, the designers of *Eiffel* developed their first compiler from a solution to what they had initially supposed were an insoluble set of constraint equations [Meye88].

10.2.1 Building from Source Modules

A conventional compiler for *Pascal* performs the familiar four phases of tokenisation, syntactic parsing, semantic analysis and code generation. Since the syntax rules of *Pascal* ensure that simple declarations are always ordered before more complex, dependent declarations in a file, *Pascal* programs can be processed efficiently in a single pass. Object-oriented languages defeat this simple model of compilation.

Because of the highly modular nature of object-oriented software, the classes required for any one application may be obtained from different source files (and directories). Units of compilation vary widely in current languages. Some, like *C++*, *CLOS* or *Object Pascal*, practise a style whereby small groups of

related classes are usually placed in a single file. Other languages, like *Smalltalk* or *Eiffel*, insist that each class be maintained separately. Clearly, the latter approach is more flexible since it allows classes to be added or deleted singly in applications. Either way, this presents a practical problem in tracing inter-class dependencies at compile time: firstly, can a logical order be found for processing files and secondly, can an optimal order be found for opening files?

The first problem concerns whether it is in fact possible to provide well-defined object-oriented programs. A class declaration depends both on ancestor declarations, from which it may inherit state attributes and methods; and on supplier-classes whose methods it calls. Logically, a class is not well-defined unless all those classes on which it depends are well-defined. However, there exists the possibility of circular dependencies, which must be resolved before the class can be considered defined. Formally, such cycles may be broken by identifying points of recursion and mutual recursion, abstracting over these in order to provide well-founded definitions, then restoring the recursion using the fixpoint finder.

The second problem concerns the fact that, due to the complex nature of inter-class dependencies, it is unlikely that source files can be processed in a single pass. When compiling a complete object-oriented program from source, the aim should be to open each source file as few times as possible, while constructing the dependency information needed for syntactic and semantic checking. Language environments which support the automatic detection of inter-class dependencies, such as *Eiffel*'s, require up to four passes of each source file [Meye88, Meye92], in which the local syntax is checked, inherited material is incorporated, routine calls dependent on other classes are checked and code is finally generated. This can lead to a significant time overhead on some filing systems. Simons *et al.* [SLN94] described a practical procedure for breaking cycles of dependency at optimal points, automatically generating a minimum of forward declarations to ensure processing of files in linear time. A source code editor builds a dependency graph incrementally from the class source files touched during an edit session, bringing forward some of the tasks normally performed at compile-time. The graph is weighted according to degrees of dependency between nodes and cycles are then broken, marking certain nodes for forward consultation. These correspond to files whose data declarations are to be read before the main pass through source. A minimum of one pass and a maximum of two are needed to build systems from source.

10.2.2 Building from Object Code

It is customary to assemble large applications from collections of source- and object-code files. Individual classes (or program modules containing a small group of related classes) are compiled to object-code, supplemented by a header file containing data declarations and method type signatures. When the application is assembled, only the remaining new code elements need be analysed in full; previously compiled code elements are simply linked in the final assembly stage, using the header files to guide syntactic and semantic

checking. This mode of working reduces locally the turnaround-time for recompilation, since only those header files on which a class (or program) is immediately dependent need be consulted.

However, incremental compilation of this kind tends to fix the state of program modules too early in the development of applications. It introduces the need for a monitoring system, whereby all dependent object-code modules invalidated by a late modification to a source file are re-compiled. *Eiffel* manages this process automatically (with the attendant multiple passes through source), whereas *C++* usually leaves this task to the programmer - either on the command line, or through maintenance of a "makefile" in which dependencies are recorded (with the attendant scope for human errors).

Standard object-code generators and linkers do not perform in an ideal way for object-oriented programs, because of the way in which they fix binding decisions too early. A method which contains a degree of internal dynamic dispatching may sometimes be invoked in contexts where more static type information is made available. In these contexts, a smart compiler should be able to remove the need for dynamic dispatch. Standard compilers will allow either the generation of a single object-code version of such a method, in which full dynamic dispatch is obligatory, or else the generation of multiple object-code versions, optimised for static dispatch over certain types. This can greatly increase the size of executable systems. Instead, a smart compiler should generate a single object-code block for each method, which may be optimised at a late stage during linkage at call-sites. This technology is not currently available, mainly because most compilers are designed around statically bound and linked conventional languages, like *C*. Progress in this area to accommodate object-oriented languages would require a general agreement on a new layout for object-code files.

A smart linker is in any case crucial to minimising the size of executable systems. Unfortunately, many compilers, upon encountering an *#include* directive, will link the entire object-code module for each class supplying services. Instead, a smart linker should link only those methods which are actually used by the client class (or program). Smart linkers capable of this task[5] have been around for some time; however they are not universally adopted. This is largely a matter of laziness in a culture where it is easier to provide more resources than it is to consider redesigning standard compilers to optimise the use of fewer resources.

10.2.3 Inheritance and Structural Templates

During inheritance, all object-code modules for ancestor classes are often *#included* indiscriminately in the final program, complete with their structural templates. However, these extra structural templates are only required if, at some point in the program, an instance of a class is treated as though it were

---

[5] Borland introduced smart linkage in products as early as Turbo Pascal 4.0.

an instance of one of its ancestors. If not, a compiler may simply construct a single structural template for a terminal class in the inheritance graph (*ie* one which will be instantiated in the application) from information obtained from many places in the inheritance graph. Although it is desirable to maintain classes at all points in the graph where significant abstraction may be captured, it is not clear, until the application is linked and built, how many of the intermediate classes require a separate representation in the application. A compiler should attempt to minimise the inclusion of such intermediate structures.

When compiling a method defined on a particular structural template, a compiler uses this information to calculate the offsets of attributes in objects of that structural type. Descendent classes will have a different structural template, due to the addition of attributes, which means that applying an inherited method to an object could result in the wrong offsets being accessed and updated - the object's data could become corrupted. Fortunately, in single inheritance schemes the structural template of a child class simply appends data storage blocks monotonically to the parent's template. Methods accessing the parent's attributes will automatically access the same, correct offsets in the child. On the other hand, multiple inheritance schemes present a difficult problem, whereby the offsets of attributes in a child class may be displaced with respect to their declared positions in the parent class. Assuming that the inherited instance variable templates of multiple parents are concatenated in order, any class inherited out of direct line (ie a second, third, ... parent) will suffer from displacement (see also figure 10.1). Many schemes for overcoming this difficulty have been proposed: these include recompiling methods inherited out of direct line, using extra levels of indirection to access sub-parts of classes and pointer arithmetic. The main approaches are reviewed in [Ng92]. A scheme similar to [Stro87, Stro91], which uses pointer arithmetic and instance variable templates, was eventually chosen for *Brunel*. This approach was preferred mainly because it reduces dramatically the number of recompiled versions of the same method in target applications, without significantly compromising access times into structures. The interested reader should consult these sources for further details.

10.2.4 Static and Dynamic Binding

Message polymorphism gives rise to dynamic binding, whereby one or other variant of a method is selected at run-time by discriminating on the type of the object. To do this, a compiler has to construct a *dispatch table* for some methods in an application. Instead of placing a direct call to the compiled method in the object-code, the compiler places an indirect call to a method obtained from the dispatch table, accessed by some index computed from the run-time type of the object and the name of the message. The size of the dispatch table grows in proportion to the product of classes and methods that use dynamic binding. An indirect call also adds an overhead to system execution time. It is therefore important to tailor the amount of dynamic binding used to the needs of the application.

Languages like *Smalltalk* and *Objective C* have a high commitment to dynamic dispatch, so minimising the time and space costs is a paramount concern (see chapter 1). *Smalltalk* builds linked dispatch vectors for each class, hashing on the method selector (message name) and searching up the inheritance hierarchy if no hit is found [GR83]. This is space-efficient, but has a variable time penalty that is on average linear in the depth of the hierarchy. Deutsch's optimisations build local caches for inherited methods as they are found; the average cost is still 1.3 lookups for a cache that is 2/3 full. *Objective C* builds a single dispatch table for the entire system, indexed by class and selector, typically yielding a sparsely-filled two-dimensional matrix of function pointers. This has constant-time access, but is space-inefficient. The sparse property can be exploited by selector index colouring [DMSV89]. Each method selector is assigned a colour, such that no colour is used twice by the same class. This reduces the number of table columns, by minimising the total number of method indices. An alternative technique packs several dispatch table rows into one vector by giving each class a unique starting offset into a vector indexed by method selectors [Dries93]. Rotating this technique through 90° packs several columns by giving each method a unique starting offset into a vector indexed by class [DH95]. For other languages, such as *C++*, the size of the table is kept small and the indexing system correspondingly simple due to the fact that the programmer has to flag dynamically bound methods explicitly in the source code. This compromises flexibility and the freedom to extend class hierarchies. Instead, an automatic approach to the detection of static and dynamic binding is adopted here. Although every method should be written as though it could be selected dynamically at run-time, a compiler should be able to determine, at the time of building the application, a large set of calls for which only one method can be invoked and replace the dynamic lookup by a static method call, or even by an inline expression. Our algorithm, described below, performs a global optimisation for a given application and relies to a certain extent on *Brunel*'s novel type system.

The policy of encapsulation protects the internal state of objects. This gives rise to a proliferation of methods whose sole purpose is to provide controlled access to state variables. In certain circumstances, chains of access methods may be required to pass requests for data on to deeply nested objects, reducing the speed and time efficiency of programs. A natural solution is to request inline expansion for these, and other, short methods. This brings an immediate speed advantage. However, inlining is not always without hazards - it prevents redefinition of the inlined method and, without peephole optimisation, may give rise to the multiple evaluation of sub-expressions. Furthermore, an aggressive inlining strategy stands to increase the size of the application code. A scheme is suggested in [SLN94] for detecting automatically safe cases when access methods can be inlined, with a guaranteed reduction in code size.

Our inlining strategies are inspired by *Self*'s *automatic message inlining* [CUL89, CU90]. As in *Self*, the emphasis is on the automatic detection of safe cases for inlining and does not depend on a special *inline* directive in the language, as found in *C++*. A compiler may choose to replace reader/writer

methods by direct access into object structures. Chains of method calls, which essentially pass external requests on to the most deeply nested object, are also detected. A compiler may recognise the syntactic similarity in delegated messages and replace the outer call by its inlined body [SLN94].

10.3 Global Optimisation of Inheritance

Inheritance is expressive as a means of sharing type and implementation. In languages like *POOL-I* and *CommonObjects* [Amer90, Snyd87], much was made of the independence of *class* and *type*. Here, classes were viewed simply as units of convenience for bequeathing implementation details to their descendants. A subtyping hierarchy was maintained separately from the "class" hierarchy (sometimes linking nodes in completely different orders; see chapter 2). In contrast to this fragmentary and sometimes flawed[6] approach, *Brunel* supports an F-bounded type inheritance and, dependent on this, the inheritance of implementations.

10.3.1 Type and Implementation Inheritance

Brunel has a single syntactic *class* construct, which has three related semantic interpretations: *class*, *type* and *template*:

- A *class* in *Brunel* is a higher-order type, described using F-bounds. It describes the common pattern of type and implementation for a polymorphic family of objects, using type parameters that are recursively instantiated in descendent classes to derive new type-bounds for inherited methods.
- A *type* in *Brunel* is generated by replacing all remaining parameters in a polymorphic structure; in particular by taking fixpoints. It describes a family of objects having the same fixed interface and implementation. Method implementations may be shared with other types, but are implicitly retyped.
- A *template* in *Brunel* is the concrete implementation schema for objects belonging to a type, namely the collection of hidden state variables for that type. A template is partitioned into state variable records, which may be recombined in different orders under multiple inheritance.

A syntactic class definition in *Brunel* may provide anything from a completely abstract specification (cf a fully *deferred* class in *Eiffel*) in terms of method stubs, to a concrete specification detailing attribute storage and complete implementations of methods. Descending an inheritance graph in *Brunel* typically describes a process of type restriction, whereby inherited methods are progressively retyped, and eventually reification[7]. Concrete attributes

---

[6] Type hierarchies were supposed to conform to subtyping, but often did not stand up to detailed inspection, especially where recursive types were concerned.

[7] Introducing a particular concrete representation is just one kind of restriction.

introduced at points in this graph must always be pertinent to the class and all its descendants.

10.3.2 Type Factorisation and Templates

Inheritance rightly encourages the creation of a great number of classes, many of which are only incrementally different from their ancestors. A multiple inheritance graph may be seen as mapping out focal points in a space of overlapping type descriptions [SC92]. However, generating a type and a template for each node in the hierarchy presents a problem from the space-efficiency viewpoint. Application programs will, in general, only need to generate types and templates from specific terminal class descriptions in the network. These classes will have inherited most of their specification and implementation from intermediate classes in the network. Many of these intermediate classes will not require a separate representation in the target code, since their only purpose is to bequeath inherited material.

A survey examining the impact of different versions of *C++* and *Eiffel* in an industrial context [Quin90] reported how programmers, having been trained to factor out the functionality of a system over some set of classes in an inheritance graph, were then surprised to find that applications were too large to load onto standard processors. A particular set of bad experiences was obtained when converting from the older single-inheritance version of C++ to the multiple-inheritance version [Stro91] and finding that compiled applications would no longer fit on 80386 processors, despite the apparent reduction in source text. The problem was eventually traced to the overhead in maintaining many finely-factored object templates in the runtime system [Quin90].

One imperfect solution is to try to increase the grain size of objects. This leads to a style where there are fewer objects, which are more multi-functional. The pressure is on the programmer to produce, every time, a class which maximises operational code at the expense of conforming to proper notions of type abstraction and factorisation. One example of the undesirable effect of this pressure is *Eiffel*'s POLYGON class in [Meye88] which, in addition to being the abstract ancestor of all SQUARE, RECTANGLE, ... classes, is also used as the concrete class to create N-vertex polygons. A discussion of how this also invalidated the type-status of the class was presented in chapter 2.

10.3.3 Collapsing the Inheritance Graph

Brunel's type system supports and requires[8] the exploitation of overlapping type spaces; therefore we should expect to see more classes and more finely factored classes in *Brunel* than in some other languages. Accordingly, a smart compiler must be able to reduce automatically the number of types and templates generated in the target language to the essential minimum for

---

[8] As a consequence of the rule for introducing methods with a distinct semantic functionality at single points in the inheritance graph.

particular applications. In [SLN94], an algorithm was published for pruning and collapsing the class graph for closed application systems. The algorithm is based on the idea that source code extracted from class libraries can be translated into a more compact form before object code is generated for the application. The process is described as an ordered sequence of transformations:

a) The compiler is invoked on the class which encapsulates the whole application. Dependencies are traced recursively from this to all other classes in the application. A dependency is either an *inherits from* or *client of* relation. The former produces a transitive closure of *ancestor* classes, whereas the latter produces a transitive closure of *supplier* classes.

b) A subgraph of the application system is then constructed. This graph contains the application class and all its (recursive) ancestors and suppliers.

c) Abstract classes (ie fully deferred classes), whose sole purpose is to provide specifications, are eliminated after static type checking. However, partially deferred classes which are the branch-points for dynamic dispatching are treated in (e) below.

d) Intermediate classes, which are not instantiated in the application, but which share physical structure among terminal classes in more than one descendent branch, generate a single global data structure, or table. References to shared[9] attributes, once their scope has been checked, are compiled out to offsets into this table.

e) Intermediate classes, which are not instantiated in the application, but which share private attribute declarations and methods among terminal classes in more than one descendent branch, generate a single state record, used to describe state variable offsets in all descendants, a method type schema and one set of methods. The templates for terminal classes are constructed later from collections of state records. *Brunel* employs a multiple inheritance scheme similar to *C*++ v2.0 [Stro87, Stro91] in which such records are applied to different offsets in objects prior to accessing attributes. Method type schemas are used during the automatic detection of static and dynamic binding (see below).

f) Terminal classes and any intermediate classes which are instantiated in the application must be fully represented in the target language. In addition to providing a state record for any new private attributes, a method type schema and a set of methods, these generate an object template, used to create instances. Object templates are partitioned into state records, which are retained separately for the sake of multiple inheritance.

---

[9] The keyword *shared* declares a state variable that is allocated once for all instances of a class, like Smalltalk's class variables. A semantics for this was given in chapter 6.

g) The inheritance graph is now collapsed. All other declarations in intermediate classes are brought down to the nearest class represented in the application. We describe this process as "flattening", whereby a chain of classes inheriting from each other is collapsed down to a single datatype which declares in one place all the inherited features.

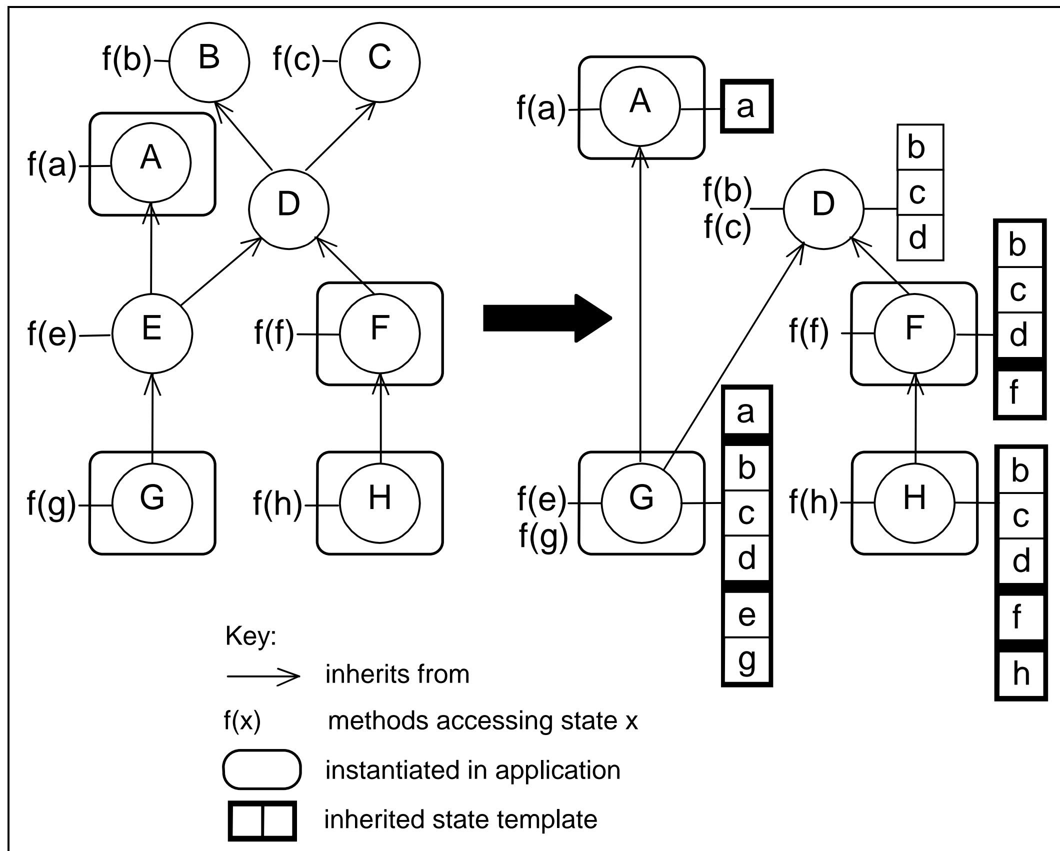

Figure 10.1: Collapsing the Inheritance Graph

Figure 10.1 illustrates the transformation of an example class graph. Each class A - H in the class library is assumed to provide a set of methods *f(a)* - *f(h)* and a block of attribute declarations *a* - *h*. Only classes A, F, G and H are instantiated in the application, so these must appear in the collapsed graph and object templates are generated for these classes. Class D collects inherited material from B and C which have been "flattened"; however D must generate a state record so that the single copies of methods in *f(b), f(c)* may access the correct offsets in instances of F, G and H. The importance of this record is seen especially in G where the attributes collected in D are inherited out of direct line, due to E inheriting multiply from A and D. D's state record template, incorporating the block of attributes *b, c* and *d*, is applied to an offset in G before any method in *f(b), f(c)* is called on an instance of G. Classes B, C and E are eliminated altogether in the collapsed graph.

From the small-scale modelling tests [Low91] conducted so far on simulated run-time systems having 10 - 100 classes, it appears that a reduction may be

obtained of between 32% and 50% on the number of object templates and state records generated at compile time. Counting each instance variable declaration as a uniform cost, this represents a total reduction of between 20% and 38% in data declarations for the application system. All the test examples were systems of finely-factored classes using multiple inheritance. The classes instantiated in the application were situated between depth 3 and 5 in the inheritance graph. The improvement was calculated with respect to the application subgraph extracted from the class library prior to collapsing. There is not yet enough data to give a statistical indication of how effective this technique is over large, actual production systems.

10.3.4 Optimising Inheritance in Other Languages

This algorithm can be viewed as a way of automatically extracting larger, more multi-functional classes from a finely-factored class library. It gives the developer the freedom to engage in abstract design, without the burden of implementation considerations. The best results are obtained where the developer's class library is quite large, with many intermediate nodes corresponding to branch points for other class variants not used in the current application. These nodes would have a separate representation in *Eiffel* and *C++* systems, but are collapsed into their descendants in our system.

Our technique could be incorporated into a *C++* to *C* translator, but not into current *C++* native code compilers. This is due to the early fixing of classes in object-code. Methods compiled over a *C++* base class automatically need that class's template, even if the application only uses direct-line descendants of that class. Our technique might be incorporated into *Eiffel* in the following way. The *Eiffel* source for a class is typically translated to *C* and compiled to one or more object-code files. These, and other files which record dependency information, are hidden in a *<name>.E* directory, corresponding to the *<name>.e* source file. *Eiffel* is able to check whether dependent files need recompilation. This approach could be adapted to generate alternative collapsed *C* sources and object-code files for classes used in particular applications projects. The idea is that *Eiffel* could check for changes in different subsystems of classes being developed for a particular application. If the inheritance structure of the subsystem changed, then a different set of intermediate *C* sources would be generated. While developing a given application, this would limit the number of re-generations of *C* sources and compilations to object-code. When switching to a new project, system-wide recompilation would occur.

10.4 Global Optimisation of Bindings

Conceptually, whenever an object receives a message to perform some action, it responds by finding the nearest appropriate method in the inheritance graph. The task of selecting which method to use in response to a given call is known as *binding* (a function to a symbolic name). If binding happens at compile-time, this is known as *static binding*. If method lookup is delayed until run-time, this

is known as *dynamic binding*. If a compiler can determine that only one type of object will ever be present at a given call-site, then it will bind that method statically. Otherwise, if it detects that some small set of object types will be present, then it will bind that method dynamically. Dynamic binding typically arises as a result of a method being *deferred*, *ie* specified in an abstract class and implemented multiple times in the concrete descendants of that class; or else when a method is redefined multiple times in descendent classes for other reasons, such as for efficiency's sake, or in order to add functionality to the inherited version.

10.4.1 Exploring Static and Dynamic Binding

In a strongly-typed language, the great majority of methods to call can be determined statically at compile-time. This is because the actual type of the objects to be encountered at the call-site is known and therefore the compiler may replace call expressions by function pointers. It has been estimated [Booc91, Simo92] that around 80% of object-oriented code can be bound statically. The time and space savings of static over dynamic binding are considerable. Looking up a method may have a constant, or linear time penalty and requires at least one extra pointer dereference, compared with placing a direct call to the method. Any methods which are never invoked dynamically can be removed from run-time dispatch tables (see above). For some 20% of object-oriented code, it is essential to have dynamic binding. This is because the most specific type of the objects encountered at the call-site is not known.

Untyped languages like *Smalltalk* [GR83] adopt dynamic binding universally. This is not appropriate for *Brunel*, from either the security or efficiency viewpoints. *Objective C* [CN91] has a type-free style of messaging with dynamic binding (such objects are of the type *id*) and an alternative typed style with static binding. *C++* and *Eiffel* have strongly-typed static and dynamic binding, which we adopt as an appropriate starting point.

The approach taken in *C++* [Stro91] is not as flexible or open-ended as it should be. The language forces the programmer to determine in advance whether a method is to be invoked statically or dynamically (dynamic methods are declared as *virtual functions*). This is wrong because it breaks Meyer's open/closed principle: if you later wanted to extend the class hierarchy and re-implement a static method for dynamic invocation, you would have to return to the original class and change the declaration style of the original method. The modification now provides the wrong solution for the majority of existing applications that do not use the additional redefined method and which therefore do not need dynamic binding.

It is insufficient to rely on some labelling scheme (such as *virtual* or *deferred*) to trigger the appropriate choice of dynamic over static binding, since merely re-implementing a method in a descendant may entail a need for dynamic binding. *Eiffel* [Meye92] provides full dynamic binding by default, with a compiler switch to optimise bindings for a given assembly of classes in a post-processing stage. In this strategy, the set of classes required for an application is extracted from

libraries and scanned for the presence of redefined methods. Unique methods are linked statically and methods that are subject to redefinition are linked dynamically [Towe94]. The post-process takes time, since it must regenerate the object-code for all methods used in the system, in case they make calls that can be rebound statically.

A different approach relies on a tighter analysis of polymorphism. *Brunel*'s type system removes ambiguities present in other strongly-typed object-oriented languages. In *Eiffel*, a compiler encountering a variable with the type annotation GRAPHIC cannot tell whether this refers to the precise *type* of object to be stored there, or a polymorphic *class* of object-types to be stored there. This is immediately apparent in *Brunel*, which makes the distinction between GRAPHIC, an albeit rather general *type*, and #GRAPHIC[G], a polymorphic *class*. Annotations that resolve to types can be bound statically without further consideration. Polymorphic annotations like #GRAPHIC[G] may or may not result in dynamic binding. This is because type constraints can be propagated through the call graph, especially where function results become targets for further calls. An expression containing a polymorphic call to *draw* might already have replaced a parameter tied to G by an actual type, say CIRCLE, for that invocation of the function. As a result, that call could be bound statically, although in general *draw* was designed for polymorphic invocation.

10.4.2 Global Binding Analysis

In [SLN94], an algorithm was published for analysing global bindings for closed systems. Like the algorithm for collapsing inheritance structures, this approach assumes a complete analysis of source code and late generation of object code. Our binding optimiser performs the following steps:

a) A complete subgraph representing the portion of the developer's class graph used in the application is constructed and collapsed using the optimisations outlined above. This eliminates spurious sibling and cousin classes which are not used in the application, flattens chains of intermediate ancestors down to shared branch-points, retaining these and all classes which are eventually instantiated in the application. Each class in the collapsed graph obtains a type signature for every method that it understands, including inherited methods with re-bound type parameters.

b) The method which initiates the application becomes the root of a call-graph which is constructed by tracing all nested invocations of methods in the application program. Each call is a node in this graph, labelled with a most specific type, obtained by consulting the type schema of the receiver object (the target variable of the call). Two invocations of the same method may therefore have different types, depending on the type of the receiver at the call-site. Any two such distinct invocations become separate nodes in the graph. On the other hand, nodes generated at different places, but which are labelled with identical method names and types, become merged.

Recursive methods therefore generate self-transitions and mutually recursive methods generate closed loops.

c) For each type-distinct method invocation, the inheritance graph is searched upwards from the type of the target object until an implementation or a declaration for the method is found. The most specific implementation (if any) is logged and marked for inclusion. If only a declaration is found, the method is considered deferred and the branch is marked as *unsafe* (see (d) below). If no declaration or implementation is found, a method invocation error is reported.

d) For each polymorphic method invocation, the inheritance graph is now searched from the class of the target object downwards towards its leaves (*ie* in the reverse direction), in order to determine if there are any re-implementations of the method in any of the application classes. The number of re-definitions (if any) is logged and the methods are marked for inclusion. While descending the graph, the algorithm marks each branch as *safe* when an implementation is detected. If any application class is touched in an *unsafe* branch, a deferred method error is reported.

e) Each type-distinct method invocation is now marked for static or dynamic binding. All static calls and those polymorphic calls having a single implementation are bound statically. Any polymorphic call having more than one associated implementation is bound dynamically. All these are included in a dispatch table and a lookup instruction is generated at the call-site. Dynamic binding results either from a method being deferred with multiple definitions, or implemented but having at least one re-definition.

f) Any methods which are not marked in the class hierarchy during the traversal of the call graph are never used in the application; they can be eliminated from the target program. This task should really be delegated to a smart linker (see above); in the absence of such, and since the algorithm effectively performs a source code cross-translation, unused methods are eliminated here.

This algorithm can be applied as part of a global binding optimisation strategy in any strongly-typed object-oriented language. Figure 10.2, adapted from [SLN94], illustrates the distinct cases in determining static and dynamic binding. The first example illustrates the case where only one static method *f(a)* can be found for a call. The second illustrates the case where a deferred method *f()* has two implementations *f(b), f(c)* in different branches and therefore is bound dynamically. The third illustrates the case where an implemented method f(a) is redefined as *f(c)* in a descendent class, requiring dynamic binding again. *Brunel* offers the advantage of discriminating monomorphic and polymorphic call-sites. In other languages, *all* call sites would have to be processed in step (d) above to search for possible method redefinitions, whereas in *Brunel*, only the polymorphic call-sites have to be processed.

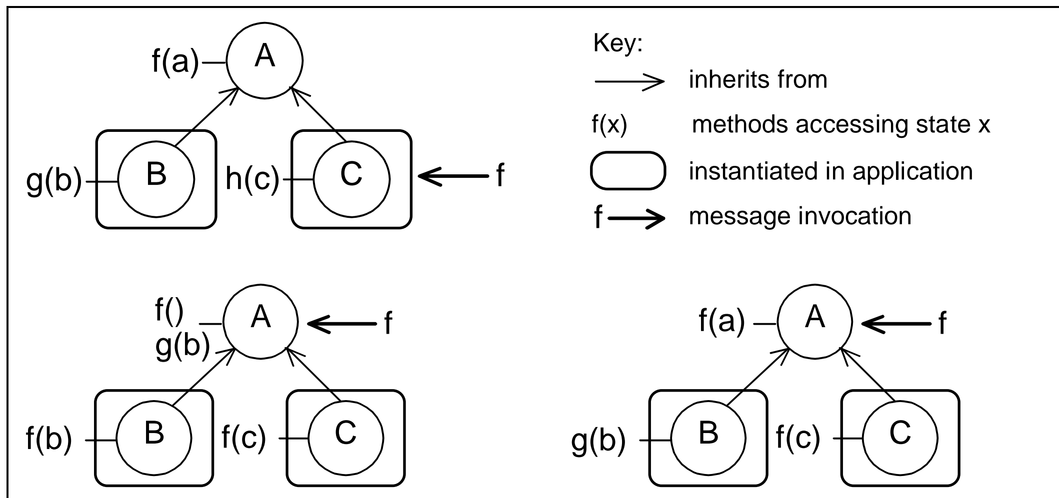

Figure 10.2: Static and Dynamic Binding

Because our algorithm is expressed in terms of traversing a call-graph, this ensures that different invocations of the same method are not lumped together in the analysis of binding. A more approximate local analysis [Towe94] would only consider each method once for possible re-definitions in the class hierarchy, whereas our approach takes into account each call and the most specific type of the target variable at the call-site. In consequence, this scheme allows the same method to be statically bound over some variables but dynamically bound over others, where it is subject to re-implementation. It all depends on the branch of the inheritance graph in which the target of the invocation is typed. Note also how our scheme allows binding to depend on which classes are needed for an application, such that incremental extensions to the developer's class library which would require dynamic binding do not force this upon existing applications.

10.4.3 Optimising Bindings in Other Languages

It is clear that this optimisation is a global post-process for specific applications. Like the inheritance optimisations described earlier, it could not easily be applied in an incrementally compiled regime. For the moment, the language definition of *C++* rules out the automatic determination of binding anyway. It is possible to improve on the late post-processing performed by *Eiffel*, using a scheme similar to the one described above for the early detection of stable inheritance subsystems. If the dependency information recorded for a class under its *<name>.E* directory were to include counts of method redefinitions and were to log the types that were the targets of creation instructions below it in the inheritance hierarchy, then stable binding subsystems could be established during the development of an application. This would allow the early generation of object-code in which bindings were optimised. Local changes to classes would be propagated upwards through the dependency files of ancestor classes. A sensitive change would result in the recompilation of any client invoking a method whose binding status had altered, the next time the system was built.

Curiously, hardly any reported research seems to have adopted our emphasis of detecting the need for dynamic binding in a statically-typed regime. By contrast, a lot of recent work has focussed on opportunities for detecting early static binding in a dynamically-typed regime. *TS* (standing for Typed Smalltalk) [JGZ88, GJ90] is geared towards retrofitting a full static type system for Smalltalk, with optimisation in mind. The type system for *TS* is different from *Brunel*, being independent of the class hierarchy and based on discriminated unions and signature types. Like the separate *protocols* of *Objective C* [NeXT93], this is necessary to capture the common type of features introduced at disjoint parts of the (single) inheritance hierarchy. Type annotations are provided and some type inference is performed, permitting the early binding of many methods.

10.5 Code Generation Strategies

Perhaps the most interesting sequence of work on binding optimisation however has been carried out by the designers of *Self* [CUL89, CU90]. *Self* has no classes (it is prototype-based) and no type annotations; furthermore it has user definable control structures and dynamic inheritance, making the optimisation task even more challenging. To compensate for the lack of class types, the *Self* compiler builds implementation-level maps to group objects cloned from the same prototype. Then, methods are dynamically compiled as they are first invoked. Through the techniques of *type prediction* and *message splitting*, multiple versions of a method are compiled, some versions optimised for particular common types (eg integer, boolean objects). Within each version, the type of the receiver is fixed and this enables the static analysis of further nested calls. The effect of this is to reduce dynamic dispatching considerably; many nested calls are simply inlined.

10.5.1 An Explosion of Object Code

Our approach shares the goals of these examples, in that nowhere in the language syntax should the full potential for dynamic use be compromised. However, it is also important to control the size of the object-code. An unbridled use of *Self*'s message-splitting technique stands to increase the number of recompilations of methods, and this is exacerbated in a finely-factored multiple inheritance regime. The trade-off in *Self* is between losing dispatching code and gaining multiple copies of other operational code. In *Brunel*, a much tighter static analysis of type may be performed than in other languages, since actual type information is propagated into polymorphic type parameters. This often delivers full monomorphic type information at call-sites for methods that were designed to be used in a polymorphic way. Without imposing some restrictions, we might obtain geometric increases in object-code size, as a result of recompiling methods at each type-distinct call-site.

To see how this might happen, consider Smalltalk's deferred class *Collection*, whose methods *collect*: and *select*: capture the general notions of mapping and filtering over all collections; or *addAll*: which appends all the elements of one

collection to another. These methods are implemented in terms of the more primitive iteration method, *do*:, and the method to add single elements, *add*:, which are deferred and implemented multiple times, once for each kind of collection. Calls to *do*: and *add*: are dynamically bound and dispatched inside these methods. Now, if there were five significantly different subclasses of *Collection* and it was desired to optimise the bindings of *do*: and *add*: such that all calls to these methods were bound statically, this would yield a five-fold increase in *Collection*'s methods, in terms of recompiled versions of *collect*:, *select*: and *addAll*:. Because of the code size trade-off, the *Brunel* compiler currently chooses to ignore some of the opportunities for early static binding. Instead, it monitors the number of redefined versions of methods at polymorphic call-sites and sets a threshold on the number of recompilations, reverting to dynamic binding if this is exceeded.

10.5.2 Conventional Code Generation

Object-oriented languages need a different strategy for code generation and linkage from that currently available. Current techniques either force the late compilation of systems by requiring a global analysis of system bindings, in which case they generate multiple object-code versions of each method used in a different context; or else they allow incremental compilation at the expense of forcing early decisions on binding. The chief difficulty with the latter lies in not knowing how a given piece of code should be bound at the time of compiling the source, so usually the most flexible (and inefficient) solution is chosen.

```
class #DRAWING [D[G]] uses #GRAPHIC [G]
(self : @D[G]) is (#OBJECT [D[G]]) with
{  private
    pen : PEN;
    element : G;
   public
    element : G { element }
    setElement (g : G) { element := g; }
    setStyle(line, colour : INTEGER) { pen.setStyle(line, colour); }
    draw { element.drawUsing(pen); }
...}


square_drawing : DRAWING [SQUARE];
square_drawing.draw;                          ...static binding


... uses #GRAPHIC [T] ...                     ...unresolved in this scope
general_drawing : DRAWING [T];
general_drawing.draw;                         ...dynamic binding
```

Figure 10.3: Binding Type Variables

Figure 10.3 illustrates a #DRAWING[D[G]] class which contains a polymorphic #GRAPHIC[G] picture element. Since the parameter G is present in the external type interface of #DRAWING[ ], it is clear that this class is intended to

be used like a type constructor. The variable *square_drawing* is declared in the parametric-polymorphc style; it has the type: DRAWING[SQUARE]. This provides full type information for the internal binding of *draw()*'s call to the method *drawUsing()*, whose target object type is now fixed at SQUARE. The *draw()* method may invoke #SQUARE[ ]'s *drawUsing()* method statically. However, the variable *general_drawing* is declared to have the type: DRAWING[T], where T is an unresolved polymorphic type. Assuming that different #GRAPHIC[ ] objects have different *drawUsing()* methods, the *draw()* method must here dispatch the internal call to *drawUsing()* dynamically.

Global static analysis and late compilation techniques may be used to obtain two object-code versions of *draw()* in this scenario. This may be tolerable; however in the general case many copies of near-identical object code will be generated, whose internal bindings are optimised over different types in the #GRAPHIC[ ] class. By contrast, in an incrementally compiled regime, *draw()* must be compiled without foreknowledge of how it will be used. This means that early static binding cannot be assumed for internal calls bound over polymorphic components such as *element* : G above. The compiler is forced to generate object-code for *draw()* with internal dynamic dispatch by default. No further advantage may be taken in a situation where the actual type of G is known at call-sites invoking *draw()*. However, only a single object-code copy of the method is generated.

10.5.3 Smart Code Generation

A compilation technique is required that supports the early generation of object-code, but late linkage to take into account the inheritance- and call-graph optimisations described above. A prime requirement is for the compiler to generate single copies of object-code for each equivalent source-code block, but to provide flexible linkage of code at each call-site. Dynamic binds should be generated in a special way, to allow automatic bypassing of the dispatch-table lookup in cases where type analysis permits early static binding.

A new approach to code generation is proposed in outline. Every method like *draw()* above can be considered parameterised over those of its internal method invocations that are polymorphic at the time the class #DRAWING[ ] is compiled. This is equivalent to extending the argument list of *draw()* to accept pointers to functions at run-time. By default, the values supplied for these arguments are expressions corresponding to the dispatch table lookup, which deliver a pointer to the intended method at run-time. However, in cases where type analysis permits early binding, the lookup expression is replaced inline at the call-site by a static pointer to the method that is dispatched inside *draw()*.

Figure 10.4 illustrates the way in which object-code could be laid out to accommodate this idea. In this scheme, code blocks reserve a special area, apart from the main body of the method, for handling dynamic dispatch calls. Instead of embedding the dispatch call in the method body, a local pointer is used at the dynamically bound call-site in the method body to redirect control to the dispatching area. In the non-optimised default case, the dispatching area

contains the addresses of lookup routines, which are compiled separately from the body of the method and linked when it is known that the method is invoked at least once in a context requiring dynamic binding. In the optimised case, the dispatching area contains the addresses of statically bound methods as a result of the linker overwriting the pointer to the lookup routine with a direct method pointer. In other words, the dispatching area is a read/writeable space for function pointers, which contains pointers to lookup routines by default. A linker must be capable of substituting the addresses of statically bound methods into the dispatching area at system assembly time. This should be no more difficult than binding functions to arguments in a conventional execution model.

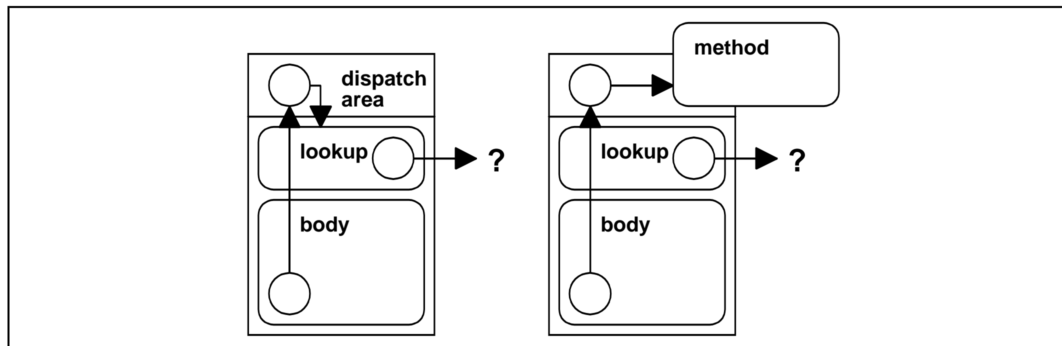

Figure 10.4: Object Code Layout

The cost of this approach is reasonable. Each method is compiled to one object-code, instead of many. The execution stack reserves additional space for each call, corresponding to the size of the dispatch area required for that method. Static binding information is passed, rather like existing value parameters, at each call-site, so a small increase in the storage required to hold method invocation expressions is to be expected. This overhead may be calculated as a fixed number of function pointers, one for each internally dispatched method whose binding may be optimised on some invocations. There is a time penalty of one extra local pointer dereference for internally dispatched methods; this is incurred when the code body redirects control to the dispatching area. Dynamic method lookup instructions are always included for each internally dispatched method in the caller's object code, but are not used in cases where the method can be bound statically.

10.6 Summary of Optimisation Techniques

We have presented several strategies for analysing and optimising object-oriented software in *Brunel*. Such strategies may be useful in any object-oriented language with strong static typing, but yield their fullest benefits in a language like *Brunel*. *Brunel* offers a more accurate type analysis than other languages, by distinguishing monomorphic types from polymorphic classes; this permits a tighter analysis of bindings, through the static propagation of type information into type parameters. An optimising compiler for *Brunel* may detect

more opportunities for early static binding than can be expected in other languages. The optimisations are specific to given delivery systems and are aimed at reducing the size of the object-code module, while achieving all speed performance advantages which do not mitigate against this.

10.6.1 Compiler Optimiser Architecture

Figure 10.5 illustrates the current overall architecture of the *Brunel* compiler/optimiser. From the diagram, it is clear that the optimisations presented here constitute a linear sequence of transformations on the global structure of some chosen target system of classes and methods. The optimisation processes are time-ordered and generate intermediate memory structures used by subsequent processes.

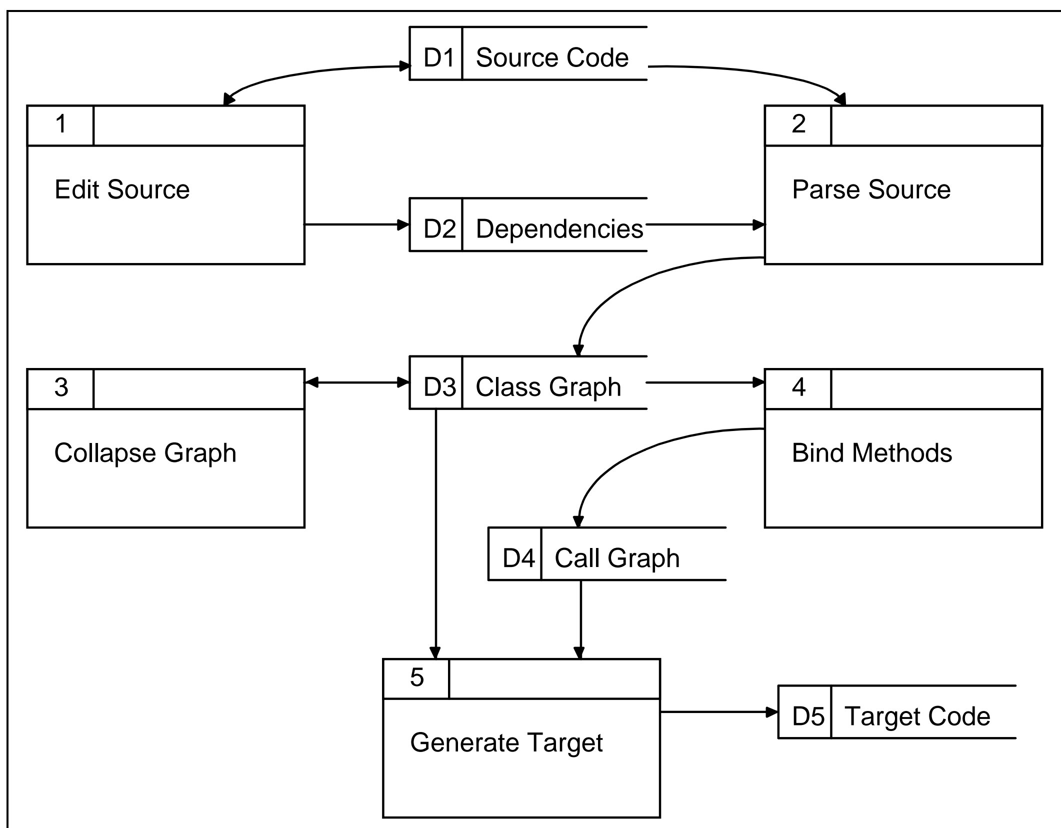

Figure 10.5: Compiler/Optimiser Architecture

The collapsing of the inheritance graph is an innovation in object-oriented optimisers and contributes to a reduction in the size of the object-code data segment of up to 30%. Programmers may freely exploit finely-factored multiple inheritance without penalty. The technique transforms a graph of classes extracted from the class library into a semantically equivalent collection of encapsulated data types. It achieves its effect by chunking data descriptions and pulling inherited method definitions down to the lowest shared point. The automatic detection of static and dynamic binding removes the need for *virtual* directives and allows methods which are bound statically in one application to be bound dynamically in another, according to need. Furthermore, the same

method may be bound statically or dynamically at different call-sites in a single application. While these binding techniques are based on previous work [CUL89, CU90] they are more sophisticated than currently used commercial techniques [Towe94] because each call-site is analysed separately; and in *Brunel* a more efficient type analysis is performed, as a result of the propagation of simple types into type parameters. All the techniques described here require a global analysis of source code for closed systems. The prototype optimiser consults files in a dependency directed linear order and performs a translation from *Brunel* source to ANSII *C* [Low91, Ong91, Tse91, Black92, Ng92]. Different techniques are necessary to support incremental compilation.

10.6.2 Future Directions

A direction has been indicated for the developers of future object-oriented compilers. In particular, it is desirable to modify the way in which object code is generated, to allow for late static binding during the linking phase. This would remove the need to compile multiple copies of object-code, which is the current strategy used when eliminating dynamic dispatch calls. The development of a new layout for object code was discussed, introducing the concept of a dispatching area.