FLEXIBLE ENCAPSULATION

Christian S. Collberg

Department of Computer Science
Lund University

FLEXIBLE ENCAPSULATION

Christian S. Collberg

Department of Computer Science
Lund University

CODEN: LUNFD6/(NFCS-1006)/1-278/(1992)

Department of Computer Science
Lund University
Box 118
S-221 00 Lund
Sweden

E-mail: Christian.Collberg@dna.lth.se

Abstract

Most modular programming languages provide an *encapsulation* concept. Such concepts are used to protect the representational details of the implementation of an abstraction from abuse by its clients. Unfortunately, *strict* encapsulation is hindered by the separate compilation facilities provided by modern languages. The goal of the work presented here is to introduce techniques which allow modular languages to support both separate compilation and strict encapsulation without undue translation-time or execution-time cost.

The thesis is divided into four parts. The first part surveys existing modular languages and modular language translation techniques. The second part presents the experimental modular imperative and object-oriented language Zuse, which supports strict and flexible encapsulation facilities. The third part describes four translation system designs for Zuse based on high-level module binding. The fourth part evaluates these systems.

The Zuse design applies the *principle of orthogonality* to encapsulation: every aspect of every exported item may be either hidden or revealed. Specifically, Zuse supports three types of exported items: *abstract*, *semi-abstract*, and *concrete*. These differ in the amount of implementation information revealed to client modules: concrete items reveal all representational details, semi-abstract items some, and abstract items none. Exported items can also be paired with a *protection clause* that restricts the ways in which they may be manipulated by particular clients.

The four translation system designs presented in the third part of the thesis assure – through the use of intermediate code module binding – that the cost (in terms of execution-time and storage) of using an abstract or semi-abstract item will be no greater than if the same item had been concrete. In addition to performing the tasks of traditional system link editors the *sequential binder* checks *deferred context conditions*, performs inter-modular optimizations, and generates code for *deferred procedures*. A deferred procedure is one for which the compiler is unable to generate code because of references to imported abstract items. Similarly, a deferred context condition is a static semantic check

which could not be performed at compile-time. The other translation systems discussed in the thesis perform the same actions as the sequential binder, but apply different techniques to improve performance: the *distributed binder* distributes its actions over the sites of a distributed system such as a network of workstations, the *incremental binder* inserts the code of modified modules in-place in the executable program, and the *hierarchical binder* binds collections of modules into *libraries* which themselves can take part in later binds.

The sequential and distributed binders have been implemented, and their performance is evaluated in the last part of the thesis. To examine the behavior of the binders, a *model* of modular programs has been developed which allows programs with widely differing qualities to be generated. Results indicate that the distributed binder runs between 2 and 3.5 times as fast as the sequential binder; for some programs it is even faster than traditional link editors.

Acknowledgments

Many people have been instrumental in bringing this work to a completion. I thank all of you collectively, but there are some who I wish to mention specially. First of all I want to thank Magnus Krampell, with whom I started out as a graduate student, and who should have shared credit for some of the results in this thesis, particularly those of Sections **??** and 5.6.[1] I am also indebted to my advisor Anders Edenbrandt, who should be awarded a special medal of honor (a purple heart?) for enduring my endless questions, rantings, and ravings. I owe both of you a lot.

I have also been fortunate to have two supportive people serving on my committee: Ferenc Belik and Svante Carlsson. By allowing me to participate in his project, Ferenc is indirectly responsible for the most interesting results in this thesis, namely those presented in Chapter 6. Svante has always found the time to talk and listen to my problems, and has the gift of offering the right kind of encouragement at those moments when nothing is going right. Svante also put me in touch with Prof. Dr. Thomas Strothotte, who read and commented on an early draft of the thesis. It was his go-ahead that finally made me believe that I might actually graduate some day. Rolf Karlsson has over the last few months helped me cut through the administrative red tape necessary to graduate. I thank you all.

I furthermore wish to thank my (one-time) fellow graduate students Arne Andersson, Mats Bengtsson, Anders Gustavsson, Christer Mattsson, and Ola Petersson, who have made the last few years bearable.

A special "Thank you, Bud!" to Bud Grace for the *excellent* cover art.

I owe everything (and then some) to my wife Sheila for our partnership in love, child-rearing, and late-night scientific, theological, and amorous jaw sessions. Sheila's belligerent use of red ink has furthermore relieved this manuscript of 243 extraneous commas. It is a comforting thought that out of the four gestation periods and eventual births that we have endured over the last

[1] This work was also presented in Magnus' licentiate thesis [130].

six years (two kids and two PhD theses), the kids' were the shortest and turned out the best. I am certainly indebted to Louise and Andrew for keeping me sane during the conception of this thesis by being living proofs of the tenet that no scientific discovery can match a good cuddle.

Finally, I share with millions of other sons the belief that I was born to the greatest person on earth. The difference between me and everyone else is that *I am right*. This work is therefore lovingly dedicated to you, my mother.

Contents

Chapter 1

Introduction

It is well known that ninety-nine percent of the world's problems are not susceptible to solution by scientific research. It is widely believed that ninety-nine percent of scientific research is not relevant to the problems of the real world. Yet the whole achievement and promise of modern technological society rests on the minute fraction of those scientific discoveries which are both useful and true.

C.A.R Hoare in Jones [112]

1.1 Background

Computer programming is set apart from other engineering disciplines by the complexity of the artifacts produced: not only may a computer program contain literally millions of components – each the result of distinct design decision – but it is also the case that the correctness and robustness of the program may depend on the behavior of each and every one of these individual components. In contrast to disciplines such as civil engineering, where an artifact is made robust by duplicating components likely to fail or by increasing the strength of each component, a computer program to which more components are added or in which the complexity of a component is increased will have greater potential for failure.

A wide variety of research directions have been pursued with the goal of better managing this inherent complexity in computer programming. Developers of languages, tools, and methodologies for the *formal specification and verification* of programs hold the view that programs should be considered pieces of

mathematical text, and as such should be subjected to formal analysis and proof of correctness. Complexity, it is argued, arises because insufficient attention is given to the separation of *what* a program should do (its *specification*, which is a statement best expressed mathematically), *how* it should be done (the program's *implementation*), and the way a specification is best transformed into a correct implementation (a process best carried out using mathematical argumentation).

Programming language researchers, on the other hand, maintain that the best way of furthering the science of computer programming is to build on the concepts and notations already familiar to programmers; i.e. to further develop programming languages to allow the clear and precise expression of designs and ideas. Programs are complex, it is argued, because the programming languages in which they are written do not have the appropriate facilities for organizing the program and expressing the necessary concepts in a natural way.

Programming environment designers, finally, argue that complex computer programs become manageable when maintained by other programs. A programming environment can help a programmer to keep track of the interrelationships of the many components of his program, can increase his productivity by making it easy to add, change, and test components, and can help him towards a full understanding of the program by allowing him to view it from different perspectives.

These approaches are of course not mutually exclusive and we feel all of them to be worthwhile fields of study. However, in this thesis we will mostly side with the programming language design community. We will present refinements to the abstraction primitives found in many systems programming languages and argue that these refinements are essential to harnessing the complexity of programming. We will furthermore argue that our language design – and its accompanying translating system designs – have secondary benefits such as the production of efficient code and reductions in overall translation time.

1.2 Programming by Abstractions

The paradigm that pervades most of the field of systems programming is *programming by abstractions*. The idea that distinguishes this paradigm from others is the following three-step approach to problem-solving:

1. Identify basic concepts inherent in the problem.
2. Isolate these concepts from details regarding their representation.
3. Determine the relationships between the concepts.

This thesis will be concerned with programming language designs for the programming by abstractions paradigm, particularly with linguistic support for the second item on this list, a principle known as *information hiding* or *encapsulation*.

The principle of information hiding may also be paraphrased such that *modules*[1] should be manipulated only through well-defined interfaces. Such interfaces should be *complete* and *minimal*; i.e. contain enough information for programmers to use and implement the modules, but no more information than that. The main advantages of this are that a user of a module need only consult its interface in order to determine the services it provides and how these services should be used, and that an implementer of a module can determine from the interface exactly which services need to be implemented. Furthermore, since the particulars of a module's implementation are not evident from the interface, there is no risk of a user inadvertently making use of such information, a dangerous situation in the event of an implementation change.

Many languages with a module concept also support *separate compilation* of module interfaces and implementations. In addition to reducing the cost of compilation after small changes, separate compilation is also essential for team programming, since it allows many programmers to work independently on different parts of the same program. Unfortunately, however, encapsulation and separate compilation are in one sense conflicting concepts, encapsulation aiming to *minimize* and separate compilation needing to *maximize* the information present in interfaces. This inherent conflict and the way it affects programming language and translation system design lies at the heart of this thesis.

1.3 Contributions and Their Relation to Previous Work

This thesis is – at its very basic level – concerned with modular language translation techniques which increase the flow of information between separately compiled modules. We will show how such techniques allow the design of programming languages with strict encapsulation and support the implementation of inter-modular program optimizers more efficient and effective than the ones currently available. More concretely, we will propose a new modular object-oriented programming language with more flexible and orthogonal encapsulation facilities than other similar languages, discuss the problems this language poses for its translating system, and based on this discussion propose and eval-

[1] The *module* (also known as *class*, *package*, or *cluster*) is the prime programming language construct for representing the implementation of abstractions.

uate four concrete translating system designs. All four designs (classified as *sequential*, *hierarchical*, *distributed*, and *incremental*) are based on intermediate code module binding.

No research is performed in a vacuum, and the contributions of this thesis naturally extend the work of others. As is customary we will indicate our more important sources of inspiration,[2] and at the same time give a brief exposé of the structure and contributions of the thesis. Numbers in parentheses refer to sections where a particular work receives more thorough attention.

Chapter 2 We present a comparative study of module concepts as they appear in a variety of programming languages and survey some existing translating system designs for modular languages.

Chapter 3 We present a new modular object-oriented language with *flexible encapsulation* facilities. The design of the language is based on the principle that any aspect of a program which can be revealed should also be able to be hidden. Briefly, the language supports fully and partially hidden (abstract and semi-abstract) types, constants, and (inline) procedures.

Precursory forms of flexible encapsulation have been incorporated into a small number of languages: Milano-Pascal [45, 46] (2.8.6) supports hierarchies of modules and a form of abstract types and constants; Mesa [235] (2.8.5) and Oberon-1 [225] use compiler hints to support abstract types and semi-abstract records, respectively; and Modula-3 [41] (2.8.9) employs run-time processing in order to allow partial revelation of object types. Sale [183] also proposes a language similar to ours with abstract types, constants, and inline procedures.

Chapter 4 We discuss how the language of Chapter 3 differs semantically from other similar languages.

Chapter 5 We present a novel translating system specifically designed to support the flexible encapsulation facilities of Chapter 3. The translating system – which is based on the binding of intermediate code – achieves execution-time efficiency at the expense of translation-time overhead. We also present a hierarchical version of this translating system, which is efficient when some parts of a program are rarely changed.

[2] Strictly speaking, the work of Sale and the MIPS group discussed below appeared in print around the same time as the first published account of the work reported in this thesis [52], and should hence be considered independent work.

Sale [183] proposes a translation scheme similar to ours with the intent to support abstract types and inter-modular inlining. It was never implemented. The MIPS compiler suite [48, 94] (5.6) supports inter-modular inlining by concatenating the intermediate code files of all modules prior to optimization and code generation. Horowitz [104] discusses the flexible use of binding-time in programming language translators.

Chapter 6 We present a distributed version of the translating system described in Chapter 5. The system achieves execution-time efficiency without any translation-time penalty and furthermore shows promise as a base for efficient inter-modular optimizations.

Chapter 7 We present an incremental version of the translating system in Chapter 5, which is likely to achieve translation-time efficiency when small local changes are made.

Incremental translating techniques have been studied extensively in the literature, but the system which has influenced the work in this chapter the most is Linton and Quong's [139, 172] (7.2) incremental linker. The R^n incremental scientific programming environment [44, 58, 57] (7.2) which supports inter-modular optimizations has also been an influence.

Chapter 8 We present a model of modular programs with flexible encapsulation facilities and examine how some of the previously presented translating systems behave when subjected to various types of programs generated by the model.

Chapter 9 We conclude the thesis with a discussion of the results achieved and an indication of directions we intend to pursue in the future.

Chapter 2

Modular Languages and Their Processors

> *It would take quite a University to accept a doctoral thesis in which the candidate refused to define his central term.*
>
> *Robert M Pirsig [167]*

> LAMBDAMAN: *Forget about these unprincipled Modula opaque types. What we need are* real *abstract types.*
> PLUCKLESS: *What are real abstract types?*
> LAMBDAMAN: *It's an abstract type whose corresponding concrete type is guaranteed to be hidden at runtime as well as at compile time.*
>
> *Anonymous in Nelson [160]*

2.1 Introduction

The central theme of this thesis is the *module*, the cornerstone on which many programming languages build their concepts of abstraction and encapsulation. This chapter seeks to review the various module concepts that have been proposed, examine their strengths and weaknesses, especially in regard to their facilities for encapsulation and separate compilation, and further to survey the processors (translators and executors) which have been designed with modular languages in mind. We will use the definitions and examples presented here to motivate our programming language design in Chapter 3, and to compare our translating system designs (Chapters 5 through 7) to the previous designs of others.

In Section 2.2 we will start by reviewing the concepts of *abstraction* and *encapsulation*. Section 2.3 will introduce some basic definitions related to programming language concepts for abstraction and modularity. Section 2.4 will introduce concepts which may serve as a basis for the classification of different forms of modularity. Here we will also discuss how separate compilation facilities blend with the module systems found in different languages. Section 2.5 is of utmost significance to the work presented in this thesis: it describes how different modular language designs have approached encapsulation, and how the design choices affect the execution efficiency and translation cost of these languages. Section 2.6 presents the *pragmatics* of modules; i.e. how modules may be used. Section 2.7 surveys translation system designs for modular languages, with an emphasis on *binding-time* operations. In an attempt to visualize the classification concepts introduced in the chapter, Section 2.8 gives programming examples from a total of ten languages and discusses how each language design has approached the problems inherent in abstraction, encapsulation, and separate compilation. Finally, Section 2.9 summarizes our findings.

The content of this chapter is not original but rather summarizes original work by others, and furthermore draws heavily on previous surveys. The more important such influences include Ambler [9] (protection), Calliss [39] (modular languages), Blaschek [29] (object-oriented languages), Conradi [54] (separate compilation), Cardelli [43] and Meyer [150] (polymorphism), and Wulf [234] (abstraction).

2.2 Abstraction and Encapsulation

For several decades the most important programming methodology has been based on the principle of "problem decomposition based on the recognition of useful abstractions (Liskov [141])." Here, *to abstract* means "to ignore certain details in an effort to convert the original problem to a simpler one (ibid)." The rationale of this tenet is that programmers can only keep a limited amount of information active at any one time, and dividing a large and complex program into smaller and simpler units will make the programming process more manageable. Programming using this methodology becomes the process of identifying the abstractions in the problem, turning each abstraction into a self-contained unit, and then finally combining the units into a functioning program.

An important concept which often goes hand-in-hand with the abstraction methodology is *information hiding* or *encapsulation*. The essence of this principle is that each abstraction should only be manipulated through a well-defined interface. The client of such an abstraction will have no need to know just how

the abstraction is realized, only how it should be used. There are many compelling reasons for hiding the implementation details of an abstraction from the scrutiny of clients. First of all, a client that has no knowledge of the implementation details of an abstraction will never be tempted to use such knowledge. Hence, if the implementation should ever change it will have no effect on the client, provided the interface remains the same. A second reason is that an implementer of an abstraction will, when given its well-defined interface, know exactly *what* to implement. The implementer may, furthermore, rest assured that the integrity of the internal data structures used to implement the abstraction cannot be compromised by the client.

Parnas [166] originally formulated these principles in the following two rules of information hiding:

1. The specification must provide to the intended user *all* the information that he will need to use the program, *and nothing more*.

2. The specification must provide to the implementer *all* the information about the intended use that he needs to complete the program, and *no additional information*; $[\cdots]$.

Abstraction and encapsulation have many applications outside the area of computer programming. In fact, information hiding has been a central part of most engineering disciplines for a long time. For example, while to the materials scientist a girder is essentially a particular alloy molded into a particular shape, to the civil engineer it is a building block (a *module*) with certain properties such as dimension and durability. The civil engineer (the client of the girder abstraction) is not required to know the details of the manufacture of the particular alloy or of its behavior at the molecular level, only the properties which are of interest in his design of bridges and houses. Similarly, the materials scientist (the implementer or supplier of the girder abstraction) will not want to keep in mind all the potential uses of the alloy he is developing, only the required properties of the new material. The essence of encapsulation and abstraction is thus to look at problems from the right perspective: too much detail only gets in the way and not enough detail hinders us from finding the right solution to the problem at hand.

2.2.1 Programming Language Abstraction and Encapsulation Concepts

Many different concepts have been introduced in programming languages in order to support the programming-by-abstractions methodology. The most im-

portant such concepts are *modules* and *classes*[1] (used for *data abstraction*) and *iterators* and *processes* (used for *control abstraction*). Data abstractions describe the data objects which will be useful in a program and the operations needed to operate on these objects. Control abstractions are used for the ordering of the events in the program. Often, there is a one-to-one correspondence between the modules in a program and the data abstractions that the program embodies, but as we shall see later there are other uses for modules as well.

The module may be the principal abstraction primitive in many languages, but it often also serves to enforce encapsulation. A popular design – and the one which will be of most interest to us in this thesis – is to explicitly divide a module into two parts: an interface (or specification) part and an implementation part. In the spirit of the Parnas quote above the interface describes the module so that clients can find out exactly what abstraction the module implements, and so that implemementers can know exactly what needs to be implemented. Ideally, and in keeping with Parnas' intentions, the specification should contain *only* this information, and nothing more. As we shall see in the next few sections, however, module interfaces in most languages leak information which really ought to be confined to the implementation. To counter this some languages have introduced an alternative to the aforementioned concept (*encapsulation by hiding*), namely *encapsulation by protection*. Language designs which embrace a protection approach to encapsulation allow implementation details to be revealed in the module interface, but protect these revelations so that clients cannot easily make use of them.

In this thesis we will argue that encapsulation is a fundamental and important concept and that programming languages should be designed so as to provide maximum control over encapsulation. In particular, programming languages should have facilities for *strict* encapsulation; i.e. it should be possible to specify interfaces to abstractions such that no information regarding the implementation of the abstraction is leaked to clients. It should be clear, however, that not everyone shares this view. Many designers of object oriented-languages – which have traditionally relied on protection rather than hiding concepts – in particular seem to hold the view that encapsulation is more of a hindrance than a boon. Stroustroup [203], for example, notes that "Hiding is for the prevention of accidents, not the prevention of fraud." In the same vein Meyer [151, p. 23] says, "Client designers may well be permitted to read all the details they want;

[1] Modules and classes are very similar concepts, distinguished by the fact that modules are *static* entities while classes are *dynamic*, and that classes *are* types while modules may *contain* types. Thus, a program can contain only one copy of each module but several instances of each class. In the following we shall mostly limit our discussion to modules, although it should be clear that much of what we have to say applies to classes as well.

but they should be unable to write client modules whose correct functioning depends on private information." Our experience is that even accidental breaches of encapsulation may have consequences resembling a fraudulent attack, and that it is therefore important for modules to hide as much implementation information as possible from the scrutiny of clients. The conclusion must be that programming languages should provide as secure encapsulation primitives as possible, and that these should first and foremost be based on hiding rather than protection.

2.2.2 The Structure of a Specification

So far, all we have said about the specification of a module is that it should contain a minimum of information; only enough for a client to use it and an implementer to implement it. But what kind of information is actually needed? Hoffman [95] divides a specification into four parts: The *INTRODUCTION* section gives a short natural language description of the module and the services it provides, the *SYNTAX* section describes how a client should syntactically use the operations provided by the module, the *EFFECTS* section describes the module's behavior (or semantics), i.e. what happens when the operations are used, and the *EXCEPTIONS* section describes the conditions under which the operations listed in the *SYNTAX* section will fail.

Few programming languages allow all four sections to appear in a module specification. The *EFFECTS* section in particular is often missing. However, almost all languages we will consider allow or require specifications to contain the declarations of data types and procedure headings, i.e. enough information to allow cross-module type checking. We will discuss this more thoroughly in Section 2.3.

A specification as we have described it here is related to Meyer's *software contracting* [151]. The specification of a module should, in Meyer's opinion, be a contract between a client and a supplier of an abstraction. The contract states that provided the client uses the abstraction in accordance with the rules stated in the *SYNTAX* and *EXCEPTIONS* sections, the behavior of the abstraction will be as described in the *EFFECTS* section.

2.2.3 Relationships Between Abstractions

The abstractions (and hence the modules) in a program can be related to one another in different ways. Two modules may, for example, be involved in an *import-export* relationship, meaning that one of the modules (the importer or client) directly uses some of the features defined by the other (the exporter or

supplier). If we are dealing with *data abstractions* the relationship will often be such that the importer uses the exporter's data objects as *a part of* its own data, in which case the relationship is one of *aggregation*. All languages with a module concept provide these kinds of import-export mechanisms.

A fundamentally different kind of relationship is *inheritance*, which organizes abstractions hierarchically, much like animals and plants are organized in families, genera, and species. We say that if an abstraction $\mathcal{B}$ inherits from an abstraction $\mathcal{A}$, then $\mathcal{B}$ has all of $\mathcal{A}$'s features, and possibly some additional features. $\mathcal{B}$ is then said to be a *specialization* of $\mathcal{A}$, much like a lion is a special kind of cat which is a special kind of mammal, etc. We also say that $\mathcal{B}$ is an *extension* to $\mathcal{A}$, since it may have extra features in addition to the ones it has inherited from $\mathcal{A}$.

Wegner [220] provides a convenient classification of languages based on whether they provide modules, classes, and inheritance: A language where classes can be related through the inheritance relation is said to be *object-oriented*, a language with classes but not inheritance is *class-based*, and a language which has some linguistic concept such as modules for data abstraction but not classes or inheritance is *object-based*. Thus the three terms form a hierarchy such that object-based languages form a proper subset of class-based languages, which form a proper subset of object-oriented languages.

2.3 Basic Definitions

In the previous section we introduced – in general terms – the notions of abstraction, module, and encapsulation. Now that the scene of discourse has been set, it is appropriate to stop and regroup and provide some solid terminology to help us talk about how these terms relate to programming language design. There is a proliferation of terminology associated with programming languages and programming paradigms, and much of it is confusing, redundant, and arbitrary. For the benefit of the discussions in this and the following chapters we will adopt our own terminology, adapted from a variety of sources. This section will only give definitions of the most fundamental terms; more specialized terms will be defined as needed.

2.3.1 Items and Modules

A *declarative item* (or *item* for short) in a language is any syntactic category which may be declared (i.e. given a name) and manipulated within the realms of the language. Typical items are types, variables, constants, and procedures, but may also include macros, exceptions, modules, processes, classes, etc.

A *module* encapsulates a collection of items and restricts the ways in which they may be manipulated by other modules.[2] In order to be able to distinguish modules from procedures we require that a module be able to *export* the items which it wants to make available to other modules and selectively *import* the items needed from other modules. We furthermore require that a module should not be callable.

A *class* is a *typed* module. Modules and classes are therefore distinguished by the fact that the former are *static* (there can be only one instance per program) and the latter *dynamic* (a program can create many instances of a class at run-time).

A module may be made up of several parts, or *units*. Often these come in two flavors, *specification* and *implementation* units. The specification units define the capabilities of the module and list the items the module exports to its *clients*. The implementation units contain the *realizations* of the items listed in the specification units. Most languages make some restrictions on the maximum number of specification and implementation units per module, and these restrictions vary from language to language. Restrictions are also often imposed on the kinds of items which a module may export and how these items may be divided between the different units.

2.3.2 Definitions and Realizations

The *definition* and *realization* of an item may mean different things to different kinds of items. As a general rule, however, the definition of an item contains all the information necessary in order for it to be used in a correct way, according to the static semantic rules of the language. The realization of an item, on the other hand, contains the information necessary for the item to be correctly represented in the memory of an executing program. As a concrete example we will consider named (or *manifest*) constants. In a statically typed language the definition of a constant must at least include its name (so that the constant may be referred to) and its type (so that the static semantic correctness of a use of the constant may be determined):

```
C : INTEGER =;
```

[2] Providing an all-encompassing definition of the term *module* has apparently been a problem to many authors, and many fail to define the term at all. Even David Parnas, the early champion of modularity and information hiding, decides to use the term "without a precise definition [165]."

The realization of a named constant, on the other hand, must at least contain its value:

```
C : INTEGER += 10;
```

Similarly, the definition of a subprogram must contain its name, the types, modes, and order of its formal parameters, and the return type in case of a function. This information, sometimes referred to as the *subprogram header*, is sufficient and necessary in order to perform strict static semantic analysis of a call to the subprogram. The subprogram realization is usually its code together with the realizations of any items defined locally.

2.3.3 Abstract and Concrete Items

In the same vein, we will consider *abstract* and *concrete* items and modules. Intuitively, an item is concrete if its realization is visible to a user, and abstract if the realization is invisible. More formally, we say that an item is abstract if its specification unit definition contains *only* the information necessary to perform strict static semantic analysis of any use of the item, and no more information. Similarly, an item is concrete if its definition *as well as* its realization is given in the specification unit. Some languages allow part of an item's realization to be given in a specification unit and the remainder in the implementation. We will call such items *semi-abstract* and denote the specification and implementation unit parts the *visible* and *hidden* parts, respectively. Further, an abstract (concrete) module is one which contains only abstract (concrete) items.

2.3.4 Other Terminology

To allow for some variation of expression in this very restricted domain of discourse the terms *interface* and *specification* will be treated as synonyms, as will *representation* and *realization*, *module* and *package*, and *(module) part* and *(module) unit*. By abuse of terminology we may also refer to the realization of a module, thereby meaning its implementation unit.

2.4 Modules and Separate Compilation

In this section we will examine and classify some of the module systems which have been proposed and realized in concrete language designs. We will pay particular attention to facilities for separate compilation and the separation of specification and implementation units, since this is of special interest to us in this thesis. The classification schemes proposed here will be put to use in Section 2.8 where some real language designs will be surveyed.

2.4.1 Specifications and Implementations

As has already been mentioned, different languages impose different constraints on the way modules may be divided into specification and implementation units. The following list categorizes languages according to the correspondence between the number of specifications and implementations:

zero-to-one Several languages, among them Euclid, Oberon-2, and Eiffel, do not provide separate module specifications. Instead, the text of a module's implementation unit contains an indication of the items which the module exports.

one-to-one A Modula-2 module has one specification and one implementation unit.[3]

one-to-at-most-one An Ada module has one specification and zero or one implementation units. A module without an implementation unit is restricted to exporting types, variables, and constants.

one-to-at-least-one Collberg [53] suggests a module system with linguistic support for multiple implementations of the same specification.

many-to-many The relationship between Modula-3's *interfaces* and *modules* and Mesa's *definitions* and *program modules* (specification and implementation units in our terminology) is more general than the *n-to-m* module systems discussed above. A procedure defined by a particular specification unit can, for example, be realized by any one of the implementation units in the program, and a particular implementation unit can realize procedures defined by several different specifications.

It should be kept in mind that the list above is concerned only with linguistic mechanisms for modularity. A programming environment for a particular programming language may provide facilities which make it possible to simulate the existence of separate specification units or handle several alternative implementations of the same specification, although the language proper does not provide linguistic mechanisms for these kinds of module concepts. The MIT CLU [142, 13, 140] programming environment [64], for example, handles module specifications as well as implementations, although the language does not provide any explicit syntax for specification units. The MIT environment can also handle several implementations of the same specification, even though

[3] We ignore the fact that the Modula-2 main program module does not have a separate specification.

the language definition does not state how an application may choose between the various implementations. Similarly, the Eiffel programming environment provides a special operation which extracts a module's specification from its implementation. This specification, however, is only for programmer convenience and does not take part in the processing of the module.

Although their module systems are much less restrictive than those of other languages, it is common for actual Mesa and Modula-3 modules to exhibit a one-to-one relationship between specifications and implementations. A two-to-one correspondence is also often used in Modula-3 to give a module one abstract specification (for use by client modules which do not need to access the representation of the module) and one concrete specification (for client modules which modify the module's behavior in some way). A one-to-many correspondence is common in Mesa as a means of dividing a large module into several smaller, separately compilable pieces.

2.4.2 Import-by-Name or Import-by-Structure

In most languages a module which needs to refer to an item exported by another module *imports* that module, and then simply uses the name of the desired item. Several varieties of this scheme have been devised:

qualified import The construction **IMPORT** *M* in Modula-2 is known as *qualified* import. The importer can refer to *M*'s items by use of the dot-notation: *M.i* means item *i* in module *M*.

unqualified import Modula-2 also permits the construction **FROM** *M* **IMPORT** *i* which allows importers to refer to *i* directly without qualification.

qualified and unqualified import The two previous forms of importation can be combined in Modula-2: **IMPORT** *M*; **FROM** *M* **IMPORT** *i*. This allows both qualified (*M.i*) and unqualified (*i*) references.

overloaded import Ada allows qualified import (**with** *M*) but also permits a client to import *all* the exported items of a module: **with** *M*; **use** *M*. Ada's rules for overloading resolution are used to disambiguate references to names defined by several modules imported this way.

Another interesting, though rarely used, approach to referencing external items is *import-by-structure*. The basic idea is that rather than specifying the *name* of the module from which a particular item should be imported, a client

specifies the item's *structure*, and the translating system is responsible for finding a module which provides a matching item. One can imagine a language in which a client in need of a stack abstraction gives the behavior specification (using pre- and post-conditions, for example) of Push and Pop, and the translating system searches all modules for one which exports these operations with the matching behavior. To the best of our knowledge no such language is available today; the one closest in spirit is Milano-Pascal, although this language does not support specification of behavior, only syntactic specification (see section 2.8.6).

2.4.3 Modules and Polymorphism

A further distinction between different module concepts is to what extent a module may be *parameterized*. A parametric module is a template representing a family of modules with similar behavior. *Generic modules* represent the most common form of module parameterization, and are present in one form or another in languages such as Ada, CLU, Eiffel, and Modula-3. Usually, a generic module is allowed to take one or more type parameters, but some languages (most notably Ada) also permit constant and procedure parameters. Generic modules are most often used to implement *container* abstractions (lists, sets, bags, and maps) which have identical behavior regardless of the type of data stored.

Meyer [151] distinguishes between two kinds of genericity: *constrained* and *unconstrained*. An unconstrained generic module $\mathcal{M}$ with a formal type parameter $\mathcal{T}$ may pose no restrictions on the kinds of types a client may use in an instantiation of $\mathcal{M}$. A *constrained* generic module, on the other hand, may require the instantiating type to fulfill certain requirements, such as having an equality operator defined, being enumerable, etc.

Parameterization is one form of *polymorphism*, but there are many others. Languages such as Eiffel whose module systems are based on classes with inheritance support a form of polymorphism known as *inclusion polymorphism* [43]. The basic idea is that a subclass can appear wherever the class itself can appear. Some object-oriented languages (Object Pascal, Modula-3) define classes and inheritance at the type system level rather than at the module system level. Naturally, these languages, too, support inclusion polymorphism.

2.4.4 Specification of Behavior

In Section 2.2.2 we noted that the interface to an abstraction should contain four kinds of information, one of which should describe the behavior or semantics

of the abstraction. A few modular languages have direct linguistic support for semantic specification (Euclid, Alphard, Eiffel), while others have defined ancillary specification languages (Anna for Ada, traces for C [96, 95], Larch for Modula-3 [114]). This is still an important area of research, and no consensus has been reached on whether specifications should be language-independent or defined within the language proper, or on what type of specification paradigm should be supported. A more detailed analysis of the problems involved is outside the scope of this thesis.

2.4.5 Separate Compilation

Compilation units are language constructs which may be submitted for compilation in source files textually separate from the rest of the program. There is some variety among languages as to what constitutes a compilation unit, but most modular languages at least allow modules to be separately compiled. In fact, in many languages the module system is so tightly coupled with the separate compilation facilities that programmers often think of them as one and the same. An Oberon-2 module, for example, is the only construct in that language which may (indeed must) be compiled separately. At the other end of the spectrum is Alphard [194],[4] which allows any kind of declarative item to be separately compiled.

A language definition must settle two issues in regard to the language's separate compilation facilities. First of all, the definition needs to state *what* language constructs may serve as compilation units. Secondly, it must state to what extent a translating system may impose an *order of compilation* between different types of units. Languages with explicit module interfaces usually impose an order of compilation between the exporting module's interface and the importing unit in order for the translating system to be able to enforce strict inter-modular static semantic correctness. Thus, if a unit $\mathcal{B}$ imports from a unit $\mathcal{A}$, such languages require the interface of $\mathcal{A}$ to be compiled prior to $\mathcal{B}$. Languages which do not require strict type checking between separately compiled modules (FORTRAN 77 and C, for example) do not need to impose an order of compilation between modules. Separate compilation in these languages is known as *independent compilation* [192].

To exemplify we will consider Ada and Modula-2, which have slightly different compilation dependency rules. Modula-2 allows compilation dependencies to exist between two specification units and between a specification unit and an implementation unit, but not between two implementation units. This is

[4] It should be noted that Alphard has never been completely implemented.

the preferred compilation dependency rule, since it guarantees that once a program's interface units have been compiled the implementation units may be compiled in an arbitrary order. Consequentially, future changes to implementation units will only require the recompilation of the affected units themselves. Ada, on the other hand, has more complex rules which permit dependencies between the implementation units of two modules involved in an import-export relationship if the exporting module is generic or exports inline procedures. In both these cases a compiler needs access to the code in the exporter's implementation unit when compiling the importer's implementation in order to be able to perform generic instantiations or procedure integrations.

Languages with a separate compilation facility have two important advantages over monolithic languages: compilation time is reduced since only affected modules need to be recompiled after a change, and several programmers can work concurrently and independently on different parts of the same program. However, very complex and restrictive compilation dependency rules may nullify all of the advantages of separate compilation. One may imagine, for example, an Ada program in which all modules depend on a particular module which exports an inline procedure. According to Ada's compilation rules the entire program will have to be recompiled whenever the body of the inline procedure is changed. If this is a frequently occurring phenomenon separate compilation may turn out to be *slower* than monolithic compilation. Furthermore, these kinds of restrictive compilation rules also make it more difficult for programmers to work independently: a programmer who makes a seemingly innocuous change to one of his own implementation units may force all other programmers to recompile all of their modules.

Trickle-down recompilation, or the "transitive propagation of possible interface changes [54]," is a related phenomenon which also negatively affects the usefulness of separate compilation. Since the compilation rules of languages with explicit interfaces require the recompilation of *all* units which transitively depend on an altered interface, a small change (such as the addition of a new operation or even the correction of a spelling mistake in a comment) to the interface of a basic module may in the worst case cause the recompilation of the entire program. The more information contained in the interface units of a program the more acute the risk that the program will suffer from many trickle-down recompilations during its lifetime. As we shall see in Sections 2.5 and 2.8, certain languages (most notably Ada) *require* interfaces to contain extraneous information. Programs written in these languages are thus more prone to suffer from trickle-down recompilations than programs written in other languages. Adams [5] proves this point in his analysis of the change history of a particular Ada program: 37% of all changed units were specifications.

2.5 Approaches to Encapsulation

Many of the problems regarding abstraction and encapsulation that designers of recent imperative and object-oriented modular languages have been wrestling with originate in a desire to satisfy four requirements:

- to allow fully abstract module specification units
- to disallow compilation dependencies between module implementation units
- to disallow language features which would entail undue execution-time overhead
- to disallow language features would entail undue translation-time overhead or which would require a non-standard translation system.

It has turned out to be a difficult task to accommodate all four of these goals in the same design. The reason is that a fully abstract separately compiled module interface – while containing enough information for a client to use the module and the implementer to implement it – does not contain enough information for a standard module compiler to generate efficient code for the module's clients. In Section 2.8 we will examine a number of concrete language designs which have selected slightly different compromises in the attempt to accommodate as many as possible of the four conflicting goals. In this section we will discuss the seven basic approaches employed by these languages, each striking a different balance between level of interface abstraction, complexity of compilation rules, and translation-time or execution-time overhead. We label them as follows:

realization restriction An exported item may be fully abstract, but the language restricts the range of possible realizations.

realization revelation The realization of an abstract item is revealed in the module specification.

realization protection Similar to **realization revelation**, but the realization is coupled with a restriction on the kinds of operations an importer may perform on the exported item.

partial revelation The realization of an abstract item is deferred to the module implementation; but, in order to facilitate translation, some aspects of the representation of the item are revealed in the module specification.

implementation ordering There is an order of compilation imposed on implementation units.

binding-time support Inter-modular information needed in order to implement abstract items is assumed to be exchanged when modules are joined together to form an executable program.

run-time support The use of an abstract item is more expensive (in terms of the memory or time required at run-time) than the use of the corresponding concrete item.

Modula-2 and Modula-3's abstract types, known as *opaque* types, are the prime example of *realization restriction*, since they may only be realized as pointer types.[5] This rule ensures that a Modula-2 or Modula-3 compiler, which only has access to the information given in specification units, will always know the size of an imported abstract type.

Inline procedures in Mesa and C++ fall in the *realization revelation* category. The bodies of inline procedures are revealed in the specification unit in order to give the compiler access to the procedure's code when processing importing modules.

Ada's abstract types, *private* and *limited private* types, fall in the *realization protection* category. An Ada private type reveals its realization in the module specification unit, but the compiler refuses a client access to the internals of the realization. The reason for revealing the realization is to make the size of the type available to the compiler: Ada favors static allocation over dynamic allocation, and without compile-time knowledge of the sizes of all types static allocation would not be possible.

Mesa's opaque type, which in regard to degree of encapsulation falls in between Ada's private type and Modula-2's opaque type, is an example of *partial revelation*. Much like in Modula-2, the realization of a Mesa opaque type is deferred to the module implementation, but unlike Modula-2 there is no restriction imposed on the realization. Instead, the module specification must reveal the maximum *size* of the realization. A similar approach is used for the implementation of Oberon-1 *public projection* types, record types which may reveal some fields in the specification unit and defer some to the implementation unit.

One advantage that Mesa's opaque type and Ada's private types have over Modula-2 and Modula-3's opaque types is that they permit static allocation. Modula-2 and Modula-3's opaque types, on the other hand, must be allocated

[5] This is the rule imposed by the Modula-2 standard. Actual implementations often relax the rule to include any type with the same size as a pointer.

on the heap. Thus, in Ada and Mesa there is never any need for explicit deallocation of variables of abstract types, since these are automatically removed from the execution stack when the flow of control reaches the end of the scope in which they were declared. In the case of Modula-2, variables of opaque types have to be explicitly deallocated, while in Modula-3 a garbage collector removes unreferenced dynamically allocated variables. In one respect, however, Modula-2 and Modula-3's opaque types have the edge on Mesa's opaque and Ada's private types: changes to the realizations of the latter may trigger trickle-down recompilations, while changes to the former only affect the module in which they are declared.

Ada generic modules and Ada modules that export inline procedures may fall in the *implementation ordering* category, since it is legal for the compiler to require that the implementation units of such modules be compiled prior to any client implementation units. Languages without explicit module specification units, such as Eiffel or Oberon-2, also fall in this category.

The only programming language which falls in the *binding-time support* category is Milano-Pascal, which requires that inter-modular information regarding its abstract types and constants be exchanged at module binding-time. Inline procedures in the MIPS compiler suite also fall in this category. The problem with this category is, of course, that extensive binding-time processing can cause excessively long turn-around-times.

The Albericht (an Oberon descendant) *public projection* type and Modula-3's opaque object type (at least as implemented in the currently available translating system) require extra work at run-time, and hence fall in the *run-time support* category. This extra work, as we shall see in Section 2.8.9, involves the calculation of offsets of fields and methods and the construction of object type templates, work that other translating systems perform at compile-time. Languages which fall in this category may thus have problems with execution efficiency.

2.6 The Pragmatics of Modules

Abstract data types may be the most frequently occurring abstractions, but they are certainly not the only ones. There have been several attempts at classifying modules according to the way they are used and the abstractions they implement. We will call this the *pragmatics* of modules. Booch [31] gives four uses of Ada modules (see Ross [181] for a slightly different classification):

1. Named collections of declarations.

2. Groups of related program units.

3. Abstract data types.

4. Abstract state machines.

A typical example from the first category is a module which exports only manifest constants. These may be related physical or mathematical constants or constants which describe the hard-wired limits of a particular program. The module in Figure 2.1 is an example of such a module taken from the translating systems described in Chapters 5 through 7. The programming language notation used is that which will be introduced in Chapter 3.

```
SPECIFICATION Configuration;
  TYPE
    String == ARRAY [RANGE CARDINAL [0 .. 4]] [CHAR];
  CONSTANT
    MaxNrOfModulesPerProgram : CARDINAL =;
    MaxNumberOfSites         : CARDINAL =;
    ImplementationExtension  : String   =;
    SpecificationExtension   : String   =;
END Configuration.

IMPLEMENTATION Configuration;
  CONSTANT
    MaxNrOfModulesPerProgram : CARDINAL += 1024;
    MaxNumberOfSites         : CARDINAL += 20;
    ImplementationExtension  : String   += ".imp";
    SpecificationExtension   : String   += ".spe";
END Configuration.
```

Figure 2.1: Example of a module from Booch's category *Named collections of declarations.*

A module in the second category (*Groups of related program units*) typically exports no types or constants, only procedures. Examples include library modules for trigonometric or other mathematical functions, modules containing procedures for converting between data formats, etc. Again we provide an example from the implementation of the translating systems in Chapters 5 through 7. The module Bits in Figure 2.2 provides operations for the low-level manipulation of machine words not provided in the language in which the translating systems were written. Again we use the language to be described in Chapter 3.

```
SPECIFICATION Bits;
  PROCEDURE ShiftLeft : (
     REF B : BITSET;
     N     : CARDINAL) =;
  PROCEDURE ShiftRight : (
     REF B : BITSET;
     N     : CARDINAL) =;

  PROCEDURE Insert : (
     Source     : BITSET;
     Low, High : CARDINAL;
     REF Dest   : BITSET) =;
  PROCEDURE Extract : (
     Source     : BITSET;
     Low, High : CARDINAL;
     REF Dest   : BITSET) =;
END Bits.
```

```
IMPLEMENTATION Bits;
  INLINE PROCEDURE ShiftLeft : (
     REF B : BITSET;
     N     : CARDINAL) +=
  BEGIN ··· END ShiftLeft;

  INLINE PROCEDURE ShiftLeft : (
     REF B : BITSET;
     N     : CARDINAL) +=
  BEGIN ··· END ShiftRight;

  ......
END Bits.
```

Figure 2.2: Example of a module from Booch's category *Groups of related program units*.

A module implementing an *Abstract data type* will at least export a type and a number of procedures which operate on the type. Often the module will also export some auxiliary type and maybe a constant providing a default or *null* value for the type. In class-based and object-oriented languages the class will not explicitly have to name and export a type, since the class itself *is* the type. We will not provide any abstract data type examples here, since there will be plenty of examples in Section 2.8 of this chapter and Section 3.6 of the next chapter.

An *abstract state machine* module, finally, differs from an abstract data type module by having an internal state which is operated on by clients through the exported procedures. The module may of course also export types and constants, but the procedures are the main objects. An abstract state machine is essentially a "black box": a box with dials (operations which change the state of the box) and readout indicators (operations which examine the current state). The difference between an abstract state machine module and an abstract data type module is that while there can only exist one copy of the former, the latter can create many instances of its exported data type such that each instance is itself an abstract state machine. Figure 2.3 gives a simple example of an abstract state machine module.

```
SPECIFICATION Elevators;
  PROCEDURE Init : (NrOfElevators : CARDINAL) =;
  PROCEDURE GoTo : (Floor : CARDINAL) =;
  PROCEDURE CallFrom : (Floor : CARDINAL) =;
  PROCEDURE AtFloor : () RETURN CARDINAL =;
END Elevators.

IMPLEMENTATION Elevators;
  IMPORT Threads, CardToCardMap;
  VARIABLE NowAt : CardToCardMap`T;
  ......
END Elevators.
```

Figure 2.3: Example of a module from Booch's category *Abstract state machines*. The example is adapted from Ross [181]. Init, GoTo, and CallFrom change the state of the machine, whereas AtFloor queries the current state of the machine.

2.7 Translating and Executing Modular Programs

A separately compiled modular program goes through several processing stages on its way from infancy (its source code form) to adulthood (its executing image in the memory of a computer). We identify four such stages:

Compilation Check the static semantic correctness of and generate code for each module.

Binding Combine the code generated for each module into one program. Resolve inter-modular references.

Loading Load the program generated during binding into the memory of the computer.

Execution Run the loaded program.

Depending on the type of translating system used, any one of these stages (except the execution stage) may be missing or may be designed to do more or less processing than indicated here. In this section we shall examine some of the translating systems which have been proposed for modular programming languages and compare their strengths and weaknesses. Of special interest to us in this thesis will be module binding, and we will therefore pay special attention to this stage of translation.

2.7.1 Compiling Modular Programs

Compiling *monolithic* modular programming languages – languages with a module concept but without a separate compilation facility – is not much different from compiling programming languages without a module concept. Euclid (see Section 2.8.4) falls in this category. Languages with separately compiled modules are more challenging, particularly if modules may be divided into textually separate specification and implementation units.[6] The main challenge is to provide compile time type checking across module boundaries. This means that there has to be some way for the compiler to determine the names and types of the items exported by the imported modules. A common design is to associate with each module a *symbol* file which describes the module's exported items. This symbol file is read by the compiler when processing the module's clients. Gutknecht [87] identifies the following classes of symbol files:

class A The symbol file for a module $\mathcal{M}$ contains only the items found in $\mathcal{M}$'s specification unit.

class B The symbol file for a module $\mathcal{M}$ contains the items found in $\mathcal{M}$'s specification unit as well as any imported items on which these transitively depend.

class α The symbol file data is in a source code or a slightly processed source code format.

class β The symbol file data is essentially an encoding of the compiler's symbol table.

The difference between class **A** and **B** is one of compilation efficiency. A class **B** compiler only needs to load the symbol files of directly imported modules, whereas a class **A** compiler must load all the symbol files which transitively depend on the directly imported modules. Class α and β also differ in regard to efficiency: compiled declarations can be loaded faster than declarations still in their source code format. The **A**α approach (used by Foster [76]) is somewhat similar to the `include`-file mechanism used for separate compilation in the `C` and `C++` languages, except that each imported module opens up a new scope for its items. Most often α approaches are implemented by letting symbol files be the specification units themselves. This has the advantage that specification units do not need to be compiled, and that there does not need to exist any explicit symbol files. The disadvantage is that errors in a specification unit

[6] Units are *textually separate* if each unit resides in its own file.

are not caught until a client is compiled. See Gutknecht [87] for a detailed description of the **B**β method.

Once a modular language compiler has loaded the symbol files of imported modules, compilation can proceed in much the same way as in a compiler for a monolithic language. One complication is name lookup for imported items, and the methods employed must depend largely on the language compiled: Ada uses overloading to resolve name clashes, Modula-2 allows both qualified and unqualified references to imported items, Eiffel allows only qualified reference, etc. See Calliss [39] for a more detailed classification. In languages which take a protection approach to encapsulation, the compiler must of course also check that client modules do not make illegal references to imported protected items. A final problem for compilers using the class **B** approach described above involves consistency checking between imported modules. Consider, for example, the case where module $\mathcal{M}$'s implementation unit imports modules $\mathcal{A}$ and $\mathcal{B}$, both of whose specification units import module $\mathcal{C}$. Assume that after a change to $\mathcal{C}$'s specification unit $\mathcal{C}$ and $\mathcal{A}$ are recompiled, but not $\mathcal{B}$. When $\mathcal{M}$ is recompiled it will read two incompatible versions of $\mathcal{C}$'s symbol file: one from $\mathcal{A}$'s symbol file and one from $\mathcal{B}$'s. Most compilers counter this problem by including in each symbol file the time when the corresponding specification unit was compiled. The compiler can then easily check these *time-stamps* to assure that the imported modules have been compiled in the correct order.

2.7.2 Maintaining the Correct Order of Compilation

In the discussion above and in our description of the order-of-compilation rules imposed by languages with a separate compilation concept (Section 2.4.5), we assumed that the rules were enforced simply by the detection of the modules which have changed since the last compilation. This is indeed often the case, and in most systems it is accomplished by checking the *file modification date* of the files in which the module source and object code are stored. The process is most often automated through the use of programs such as the `Unix make` [73, 213, 14, 15, 16] facility, which recompiles affected modules after editing changes. This is a very coarse-grained approach since, as we have already noted, simple formatting changes can lead to massive recompilations. To counter this, more fine-grained approaches have been developed. Known as *smart recompilation* [209, 190], these methods take into account not only which *modules* have been changed but also which *items* in the modules have been changed. Unfortunately, smart recompilation methods have met with limited success since they require compiler cooperation and since the overhead of checking whether a particular module needs to be recompiled sometimes outweighs the cost of

actual recompilation.

2.7.3 Binding Modular Programs

Most compilers for separately compiled modular languages produce one *object code file* per source code module. This file contains the code generated for the subprograms exported by the module, space for the static variables declared in the module, and the values of the module's structured manifest constants. In addition to this information the object code file often contains an encoding of the module's symbol table, to be used by debuggers and module binders.

A *module binder* is a program which combines the object code files produced by compilers or assemblers into a file which can be directly loaded and executed. Most operating systems contain a simple module binder (often called a *link(age)-editor* or *linker*) which handles the binding requirements of assemblers and languages with rudimentary module systems such as C and FORTRAN. The main task of these system linkers is to gather together the code and data produced by the compiler for the modules which are to take part in the resulting program, to resolve references between these modules (this is known as *relocation*), and to include whatever other system-specific routines the program will need at run-time. The link editor often also serves as the main instrument for inter-language programming; i.e. the linker allows modules written in one language to be bound together with modules produced by compilers for other languages.

Special Linking Techniques

Languages with module concepts more advanced than C and FORTRAN often require more complex link-time processing than what is offered by ordinary system link editors. Often this problem is handled through the use of a *pre-linker*, a program which is designed to take care of the special link-time requirements of a particular language and which as a part of its processing calls the system linker to handle the relocation and concatenation of code and data. Typical tasks performed by pre-linkers include: checking time-stamps (similar to the compile-time checks discussed previously), finding the modules that should be part of the final program, and determining the order of execution between module bodies.

In operating systems where several processes can execute concurrently it is often the case that certain modules or groups of modules are used by several programs simultaneously. Examples include modules for buffered I/O, mathematical routines, and graphics primitives. To avoid having several copies of

the same modules in core at the same time, some operating systems employ a technique known as *dynamic linking*. A module is linked dynamically if several concurrently executing programs can share the same copy of the module. If a particular module is to be linked dynamically (rather than statically) into a certain program, the linker refrains from including its code and data into the resulting executable file. Rather, when the program is loaded or when a routine in the dynamic module is called for the first time, the module's object code file is loaded into main memory and cross-module references are relocated. If, however, at this point the object code file is already present in main memory (due to the fact that it is being used by another concurrently executing program), the loading can be dispensed with.

Another linking technique, sometimes called *smart linking*, is supported by some linkers to reduce the size of the executable file. The idea is to try to avoid including code and data in the resulting executable file which could never be referred to at run-time. The linker accomplishes this by constructing the call-graph of the procedures in the program and traversing the graph (starting at the main module) to determine the procedures which might be called. The result is, of course, never more than a conservative estimate of the procedures which are *actually* called at run-time.

Two techniques have been employed in attempts to speed up the linking process: *incremental* and *hierarchical* linking. An incremental linker saves the resulting executable file and data structures used during linking between sessions and uses this information to incrementally update the executable file in response to changes in the object code files. See Section 7.2 for a more thorough discussion. A hierarchical linker speeds up linking by binding together the object code files of groups of modules to form new object code files (or *libraries*). Since all cross-module references within the modules in such a library are resolved once and for all when the library is created, future links which use the library will be faster. See also Section 5.11.

We refer to Presser [170] and Beck [20] for early and modern views of linking and loading.

Code Generation and Optimization at Module Binding Time

It has often been observed that due to a lack of inter-modular information many code optimization techniques cannot be successfully applied at compile-time. Such inter-modular (or inter-procedural) optimization techniques include inter-procedural register allocation, inter-procedural constant propagation, and inter-modular inlining. It is natural to investigate whether these techniques can instead be applied at module binding-time when all the necessary inter-modular

information is available. Indeed, this is exactly what we will do in Chapters 5 through 7.

Previous work in this area includes the MIPS [48, 94] compiler suite which performs procedure integration and other inter-procedural optimizations at link-time, Wall's [214, 218, 216, 217] and Santhanam's [186] inter-procedural register allocators, and Benitez's [21] optimizing linker which performs procedure call optimizations at link-time. In the MIPS system the linker simply concatenates the intermediate code files produced by various compilers into one large file. Other tools then perform optimization and code generation on this file, and the resulting code is run through the assembler and system link editor to form the executable file. Wall's linker works on object code files where the relocation information has been extended to include information regarding variable usage. Based on this information, the program's call-graph, and collected profiling information, the linker allocates frequently used variables to global registers and restructures the code generated by the compiler accordingly. Benitez's optimizing linker works on machine code in a *register transfer list* (RTL) representation. The linker (which is designed to restructure procedure calls to pass arguments in registers and to remove unnecessary test instructions after procedure calls) emits assembly code which is run through the system assembler and linker to produce the executable file. Santhanam also performs optimizations at link-time, although his translating system does not contain an optimizing linker. Rather, each module is run through the compiler *twice*, the first time to gather inter-modular information, and the second time to perform the actual optimizations. The object code files are then linked as usual with the system link editor.

Naturally, linkers which have been extended with functions for optimization are likely to be much slower than traditional link editors. Benitez reports that his optimizing linker is three to six times slower than the system linker. Most of the time is spent in the assembly phase. Wall's link-time optimizations slow down the linker by a factor of 2 to 4. In Chapters 6 and 7 we will propose techniques for link-time optimization which have the potential for making optimizing linkers as fast as (and sometimes faster than) traditional link editors.

Language-specific Module Binders

So far, we have only discussed language-independent module binders, i.e. binders which are designed to handle modules produced by compilers for many different languages. In this section, however, we will give a brief overview of some translating systems where the module binder has been designed to work with only one particular language or to handle certain language-specific problems.

The module binder for the Mesa language can combine modules in many interesting ways. However, Sweet reports [204] that most binding requests are simply of the kind "bind them [the modules] together in the only way possible." For more complicated situations Mesa provides a special configuration language, C/Mesa [204, 235]. C/Mesa, for example, allows one to describe how groups of modules should be joined into libraries (hierarchical linking).

Ancona [11] describes a system whereby Pascal modules are bound at the source code level. After binding, the output of the binder is submitted to a compiler for monolithic Pascal. The binder is able to support a simple form of generic modules.

Some special binders are also designed to perform inter-modular semantic checking. Kieburtz [124] and Celentano [45, 46] both describe binders which perform inter-modular type checking for modular Pascal dialects. For a discussion of the latter system see Section 2.8.6.

Thorelli [206, 207, 208] describes a binder for modules in the language *PL*. PL modules fall under the *import-by-structure* (see Section 2.4.2) and *zero-to-one* headings (Section 2.4.1) and are, unlike modules in most languages other than Milano-Pascal (Section 2.8.6), completely autonomous. The binder is designed to combine separately compiled modules according to instructions given to it in the linking language *LL*, which can describe module interconnections in a variety of ways. Since no inter-modular type checking can be performed at compile-time, this task has to be deferred to the linker.

2.7.4 Loading and Executing Modular Programs

Loading and execution are the last translation phases of any modular program, and one might expect these to be uncomplicated. For the most part this is true, but we will touch on some of the more salient issues.

An important question is what happens when a program *starts* and *finishes* executing. Many languages allow the programmer to specify actions to be taken at one or both of these points. Each Modula-2 module, for example, has a (possibly empty) *module body* which is executed after the program has been loaded. The actions performed in module bodies are usually limited to the initialization of static variables, and if this is the case the order in which the module bodies are executed is most often immaterial (the main module body must, of course, be executed last). However, if a module body makes use of operations defined in imported modules, it is essential that these modules be initialized first. The Modula-2 standard therefore states that the body of a client module should be executed *before* the bodies of the modules it imports. In case of circular dependencies the order is left unspecified.

Similarly, Euclid, whose modules are typed, allows *initial* and *final* actions to be associated with each module. The initial action is executed when a module variable is created, and the final action is executed when the variable is destroyed.

Object-oriented languages (class-based languages with inheritance) have a somewhat more complicated run-time behavior than object-based languages. There are two main sources of complexity: *dynamic binding* and run-time type checking. As we have already explained, object-oriented languages allow a certain degree of polymorphism – descendants of a class can be used wherever the class itself can be used (they essentially *are* the class since they fulfill the *is-a* relationship) – and in some cases it is not possible to determine at compile-time whether the assignment of one class instance to another is legal. This check therefore has to be deferred till run-time where it is performed based on a type-tag stored with each instance. Object-oriented languages also support dynamically bound procedures (also known as *virtual* procedures or *virtual methods*). A virtual procedure declared in a class can be overridden by descendant classes, i.e. replaced by another procedure whose behavior is more suited to the descendant's needs. Dynamic binding is usually implemented (see for example Harbison [89, pp. 199–204]) by storing with each class instance a pointer to a *method template*, which is a list of addresses constructed at compile-time of the class' virtual procedures. All method calls are done indirectly through this template.

The final point we will consider is run-time code generation and optimization. We have already seen that optimization can be done at compile-time as well as link-time, and it is therefore natural to ask whether there are some types of optimizations which are best applied during execution. The answer to this question (at least for some types of languages) seems to be 'yes', but unfortunately there has yet to be much work done in this area. Chambers [47] describes a translating system for the SELF language which performs run-time code generation in order to produce customized code for different combinations of run-time types. See also Keppel [122] for a thorough discussion of run-time code generation.

2.7.5 Integrating Translation Phases

A obvious method for achieving faster turn-around times in program development environments is to integrate two or more of the translation phases. *Load-and-go* [20] assemblers and compilers, for example, dispense with the linking phase and load the generated code directly into main memory for execution.

The Oberon system [228] also dispenses with linking altogether. Instead

modules are loaded and bound together as needed during execution. This is somewhat similar to *dynamic linking* as described earlier in this section. The advantage is, of course, faster turn-around time for the programmer: not only is one translation phase completely eliminated, but it may also be possible to change a running program simply by reloading a new version of a changed module. The disadvantage, at least for the Oberon system, is a "noticeable [228, p. 893]" procedure call overhead. The reason is that an inter-modular procedure call must be preceded by a table lookup to find the procedure's address.

Integrated and *incremental* programming environments such as the *Rational* [222, 12, 71] Ada environment and *Gandalf* [161] integrate editing and all or most of the different stages of translation. The result is a translation system which continuously updates the executable code in response to editing changes. See Section 7.2 for a more detailed discussion.

2.8 A Survey of Programming Language Module Concepts

We will next attempt to illustrate the concepts discussed previously in the chapter by reproducing the ubiquitous stack abstraction in ten model languages: Ada, Beta, Eiffel, Euclid, Mesa, Milano-Pascal, Standard ML, Modula-2, Modula-3, and Zuse. The languages have been chosen so as to illustrate the trade-offs between abstraction, encapsulation, modularity, and separate compilation in programming language design, and each example has been coded in a way that we believe best illuminates the language's strengths and weaknesses.

2.8.1 Ada

Ada's module system is characterized by three things: use of *realization protection* as the prime encapsulation mechanism (although Modula-2 style pointer hiding – *realization restriction* – is also supported), the availability of constrained as well as unconstrained generics, and *implementation ordering* for generic modules and packages exporting inline routines.

The generic stack definition of Figure 2.4 illustrates these points. The realization of the stack abstraction is given in the module specification unit under the **private** heading. The only operations available to clients on variables of type GENERIC_STACK.STACK are PUSH and POP; the declaration **limited private** explicitly outlaws assignments and equality-tests which would have been legal if STACK had just been declared **private**.

In general, Ada specification and implementation units are compiled sepa-

```
generic
   type ITEM is private;
package GENERIC_STACK is
   type STACK (SIZE : POSITIVE) is limited private;
   procedure PUSH (S : in out STACK; E : in ITEM);
   procedure POP (S : in out STACK; E : out ITEM);
   pragma INLINE (PUSH, POP);
private
   type VECTOR is array (POSITIVE range < >) of ITEM;
   type STACK (SIZE : POSITIVE) is record
         SPACE : VECTOR (1 .. SIZE);
         INDEX : NATURAL := 0;
      end record;
end GENERIC_STACK;

package body GENERIC_STACK is
   -- Implementations of PUSH and POP ...
end GENERIC_STACK;

with GENERIC_STACK;
procedure MAIN is
   package STACK_INT is new GENERIC_STACK (INTEGER);
   S : STACK_INT.STACK (100);
begin
   STACK_INT.PUSH (S, 314);
end MAIN;
```

Figure 2.4: Ada generic stack example, adapted from Rogers [179].

rately, and changes to implementation units do not affect importing modules. The Ada reference manual however allows programming environments to require implementation units of modules containing exported inline routines to be compiled prior to the module's clients. Similarly, programming environments may disallow the separate compilation of specification and implementation units of generic modules.

2.8.2 Beta

The language Beta generalizes programming language concepts such as procedure, function, class, and process into one concept, the *pattern*. It differs significantly from the other languages described in this section, particularly when it comes to separate compilation and modularization which is not defined within the language proper. Instead, a language-independent mechanism

– called a *fragment system* – forms the basis of the separate compilation system and is used to split programs into modules, modules into specification and implementation units, and implementation units into variants. A fragment is any legal sequence of terminal and nonterminal symbols which can be generated from a non-terminal in the language's context-free grammar. Fragments can be textually separate or part of fragment groups, and can contain links to other fragments.

```
ORIGIN 'betaenv';
BODY 'stackbody'
-- LIB: Attributes --
Stack: (#
    Private: @ <<SLOT Private: descriptor>>;
    Push:    (# e: @Integer enter e do <<SLOT PushBody: descriptor>> #);
    Pop:     (# e: @Integer do <<SLOT PopBody: descriptor>> exit e #);
    New:     (# do <<SLOT NewBody: descriptor>> #);
 #)

ORIGIN 'stack' [[
    -- Private:     descriptor -- (# A: [100] @Integer; Top : @Integer #)
    -- PushBody: descriptor -- (# do (* Code for Push *) #)
    -- PopBody:   descriptor -- (# do (* Code for Pop *) #)
    -- NewBody:  descriptor -- (# do 0 -> Private.top #)
--]]

ORIGIN 'betaenv' [[--
INCLUDE 'Stack'
    -- Program:descriptor -- (# S: @Stack do S.New; 314 -> S.Push #)
--]]
```

Figure 2.5: Beta stack example, adapted from Knudsen [127].

The Beta program in Figure 2.5 is split into three textually separate fragments, each starting with the keyword **ORIGIN**. The first fragment corresponds to the specification unit of the Stack module, the second is a fragment group corresponding to the Stack's implementation unit, and the third fragment is the main program itself. The syntax "<<SLOT PushBody: descriptor>>" in the first fragment serves as a place-holder for a fragment with the name *PushBody* of the syntactic category *descriptor*. *PushBody* is defined in the "implementation" fragment.

2.8.3 Eiffel

Eiffel's module system is built on the *class* concept. An Eiffel class is separately compiled, dynamically or statically allocated, and may inherit as well as import from other classes. Constrained as well as unconstrained generic classes are supported. Behavior is specified in the form of pre- and post-conditions and invariants, and is tightly coupled with the exception mechanism.

```
deferred class STACK [T] export depth, empty, full, push, pop
feature
    depth : INTEGER      is deferred end;
    empty : BOOLEAN      do Result := (depth = 0); ensure (depth = 0); end;
    full : BOOLEAN       is deferred end;
    pop : T              is require not empty deferred
                         ensure not full; depth = old depth - 1 end;
    push (x : T)         is require not full deferred
                         ensure not empty; depth = old depth + 1 end;
invariant depth >= 0
end -- STACK

class FIXED_STACK [T] export depth, empty, full, push, pop
inherit STACK [T]; ARRAY [T]
feature
    Create (n : INTEGER) is do ... end;
    -- redefinitions of depth, full, pop, and push using ARRAY.
end -- FIXED_STACK;

class ROOT feature
    Create is local S : FIXED_STACK [INTEGER] do S.Create (100); S.Push (314)
    end
end -- ROOT;
```

Figure 2.6: Eiffel generic stack example, adapted from Meyer [151].

Eiffel classes are of the zero-to-one kind; i.e. there are no explicit specification units. The Eiffel programming environment, however, provides a special operation (**short**) which extracts a class' specification from its implementation. Furthermore, *deferred classes*, classes which contain some features which are not fully realized, can be used to simulate specification units. This is an attractive approach for two reasons: (1) a deferred class can be the ancestor of several different realizations of the same abstraction and (2) the behavioral specification in the deferred class applies to descendant classes also.

Figure 2.6 gives a deferred generic stack definition *STACK*, followed by an implementation *FIXED_STACK*. An alternative implementation

(*LINKED_STACK*, for example) would also inherit from *STACK*. *FIXED_STACK* uses an array realization and therefore inherits from the *ARRAY* class as well as from *STACK*. The pre-conditions (the **require** clauses), post-conditions (**ensure** clauses), and invariants of the features of *STACK* apply to the corresponding features in *FIXED_STACK* as well.

2.8.4 Euclid

Euclid was designed with verification in mind, and the language includes assertions, pre- and post-conditions, and invariants, along the same lines as Eiffel. Euclid is class-based (modules are typed) and modules can be parameterized with compile-time expressions.

```
type IntStack (size : unsignedInt) = module exports (Push, Pop)
    type range : 1 .. size
    var store : array range of signedInt
    var index : range
    inline procedure Push (E : signedInt) = pre (index < size) begin ... end Push
    inline procedure Pop (var E : signedInt) = pre (index > 1) begin ... end Pop
    initially index := 0;
    invariant ((index >= 0) and (index <= size));
end IntStack.

include IntStack;
type Main = module var S : IntStack (100);
    initially begin S.Push (314) end
end Main.
```

Figure 2.7: Euclid parameterized stack example. Post-conditions have been omitted.

Euclid modules are of the zero-to-one kind and may, but are not required by the Euclid report [135] to be, separately compiled. Figure 2.7 gives a Euclid stack module of client-defined size. As in the Modula-2 example (Figure 2.11), generic modules may be constructed using low-level facilities.

2.8.5 Mesa

The Mesa module system is of the *many-to-many*-kind; i.e. the items defined by a particular specification unit can be realized by any one of the implementation units in the program. A special module binding language – C/Mesa – is used to describe the connections between the modules in a program.

```
Stack: DEFINITIONS =
BEGIN
    T : TYPE [202];
    PT : TYPE = LONG POINTER TO T;
    Init : PROC [S : PT];
    Push : PROC [S : PT; E : INTEGER];
    Pop : PROC [S : PT] RETURNS INTEGER;
END.

StackImpl: PROGRAM EXPORTS Stack =
    BEGIN
    T : PUBLIC TYPE = RECORD [
                space : ARRAY [1 .. 100] OF INTEGER;
                index : [0 .. 100]];
    (* Implementations of Init, Push, and Pop ... *)
    END.

Main: PROGRAM IMPORTS Stack =
    BEGIN S : Stack.T; Stack.Init [@S]; Stack.Push [@S, 314]; END.
```

Figure 2.8: Mesa stack example. "[202]" in the definition of T refers to T's size. "Stack.Init [@S]" passes the address of S to Init. This construction must be used since Mesa only has pass-by-value parameters.

Two kinds of hidden (*opaque*) types are provided: a Modula-2-style hidden pointer type and a hidden type whose size is revealed in the specification unit. In addition to this, types (or parts of structured types) can be protected by a **PRIVATE** clause which prevents clients from accessing the types' internal structure. The **SHARED** clause can be used by clients to override such access restrictions. Inline procedures are also available in Mesa, but the code of an exported inline procedure must be revealed in the specification unit. Mesa thus supports four different types of encapsulation: *realization restriction*, *realization protection*, and *partial revelation* for abstract types, and *realization revelation* for inline procedures.

The stack definition in Figure 2.8 shows the use of opaque types with revealed size. The main advantage of revealing the size is that variables of the type can be statically allocated and hence do not tax the dynamic allocation system.

2.8.6 Milano-Pascal

The Pascal dialect developed at the *Politechnico di Milano* (henceforth Milano-Pascal) is interesting for several reasons: it is one of a few modular languages which explicitly requires a non-standard translation system, it supports fully abstract types and constants, it provides completely autonomous modules, and it allows nesting of separately compiled modules. The main aspect that separates Milano-Pascal from the other languages in this survey, however, is that clients specify the *structure* of the items they need to import rather than, as is more common, the name of the module which should provide these items.

```
module Stack;
    const stack_size = 100;
    type int_stack = record space : array [1 .. stack_size] of integer;
                index : integer;
            end;
    procedure init (var S : int_stack); begin ... end;
    procedure push (var S : int_stack; E : integer); begin ... end;
    procedure pop (var S : int_stack; var E : integer); begin ... end;
endmodule.

module Main;
    interface spec
        const stack_size = ?;
        type int_stack = ?;
        procedure init (var S : int_stack);
        procedure push (var S : int_stack; E : integer);
        procedure pop (var S : int_stack; var E : integer);
    end interface spec;

var S : int_stack;
begin
    init (S); push (S, 314);
endmodule.
```

Figure 2.9: Milano-Pascal stack example, adapted from Celentano [46].

Consider the program in Figure 2.9. The module Main needs a stack and therefore indicates in its *interface spec* that it must be bound in an environment which contains a constant *stack_size*, a type *int_stack*, and three procedures *init*, *push*, and *pop* with formal parameters of the specified types and modes. In other words, *Main* does not import a *Stack* module, but rather indicates that there must exist in its environment items which satisfy the properties of a stack.

It follows that modules *Stack* and *Main* are completely independent of each other, and can be compiled in an arbitrary order. Inter-modular type checking is done at link-time rather than compile-time by a language-specific linker. The input to the linker is a parenthesized expression of module names describing the nesting of the modules. In addition to static checking, the linker has to allocate variables whose size cannot be determined at compile-time, for example S in module *Main* (Figure 2.9).

2.8.7 Standard ML

ML is – in contrast to the other languages studied here – a *functional* language. ML is strongly typed and has a polymorphic type system. Most ML implementations are interactive in nature; i.e. the system accepts an expression as input, evaluates it, and prints the result. In the ML version of our stack example (Figure 2.10) the `signature` serves as the stack's specification, introducing the stack type `T`, an exception `E` which is raised by `pop` and `top` when the stack is empty, and functions operating on `T`. The `'a T` idiom indicates that we are dealing with a polymorphic stack, one which can operate on any type of element. The implementation of the stack is given in the `abstraction`, where each of the items introduced in the `signature` receives a representation. The use of `abstraction` indicates that a client of `Stack` has no access to its representation. Exchanging `structure` for `abstraction` would make the representation available for clients to manipulate. Other aspects of the ML module system will have to be omitted for the sake of brevity.

2.8.8 Modula-2

Modula-2 modules are of the *one-to-one* kind, and type abstraction is through *realization restriction*: *opaque* types are restricted to pointers. There is no linguistic support for generic modules, but untyped generics can be programmed by use of low-level facilities. An example of this is shown in Figure 2.11. It is also interesting to note the difference between this example and the previous one in Mesa given in Section 2.8.5. The Mesa stack was an opaque type with revealed size and could therefore be allocated statically, whereas the Modula-2 stack is an opaque pointer type and therefore needs explicit routines for dynamic creation and destruction.

2.8.9 Modula-3

Modula-3 extends Modula-2 and Mesa with features for object-oriented programming. It is an interesting language since it breaks with many of the

```
signature STACK = sig
  type 'a T
  exception E
  val empty:  'a T
  val push:  'a T * 'a -> 'a T
  val pop:  'a T -> 'a T
  val top:  'a T -> 'a
end;
```

```
structure Main = struct
  local open Stack in
      val S = push (empty, 314);
      val r = top (S);
      val S = pop (S);
  end;
end;
```

```
abstraction Stack :  STACK = struct
  datatype 'a T = Stack of 'a list;
  exception E;
  val empty = Stack([]);
  fun push(Stack(S), x) = Stack(x::S);
  fun pop(Stack(x::S)) = Stack(S) | pop(Stack([])) = raise E;
  fun top(Stack(x::S)) = x | top(Stack([])) = raise E;
end;
```

Figure 2.10: Standard ML stack example.

traditions kept by the other languages in the family of Pascal descendants: Type compatibility is *by structure* rather than *by name*, dynamic storage is reclaimed automatically, the module system is of the many-to-many variety, and the encapsulation concept is such that it requires extensive post-compilation processing. The last point is of particular interest to us in this thesis, since the language which we will describe in Chapter 3 has similar problems, only more so. We will therefore study in some detail the requirements which Modula-3 imposes on its translating systems.

Consider the Modula-3 stack abstraction in Figure 2.12. It consists of the two interfaces *Stack* and *StackRep*, an implementation unit *StackImpl*, and a client module *StackUse*. The *Stack* interface states that the Stack type *Stack.T* is a subtype of **ROOT**, which reveals that *Stack.T* must be realized as a an *object type*. Object types form Modula-3's basis for object-oriented programming (they can be related by aggregation as well as by inheritance), and are usually implemented as a pointer to a record containing the object type's data fields as well as a pointer to its *template*. The template is essentially a (constant) record of pointers to the object type's procedures. The *StackRep* interface reveals further information about *Stack.T*, namely that it is a subtype of *StackRep.Link*. In other words, the *Stack.T* realization must at least have two fields: *next*

```
DEFINITION MODULE GenericStack;
   IMPORT SYSTEM;
   TYPE Stack;
   PROCEDURE Create () : Stack;
   PROCEDURE Destroy (VAR S : Stack);
   PROCEDURE Push (S : Stack; E : SYSTEM.ADDRESS);
   PROCEDURE Pop (S : Stack; VAR E : SYSTEM.ADDRESS);
END GenericStack.

IMPLEMENTATION MODULE GenericStack;
   IMPORT SYSTEM, Storage;
   TYPE Stack = POINTER TO RECORD
              space : ARRAY [1 .. 100] OF SYSTEM.ADDRESS;
              index : CARDINAL;
            END;
   (* Implementations of Create, Destroy, Push, and Pop ... *)
END GenericStack.

MODULE Main;
   IMPORT GenericStack, Storage;
   VAR S : GenericStack.Stack;
         E : POINTER TO INTEGER;
BEGIN
   S := GenericStack.Create ();
   NEW (E); E^ := 314; GenericStack.Push (S, E);
   GenericStack.Destroy (S);
END Main.
```

Figure 2.11: Modula-2 generic stack example.

and *item*. Finally, *StackImpl* reveals that *Stack.T* is a direct subtype of *StackRep.Link*, and that in addition to the fields *Stack.T* receives from *StackRep.Link* the complete realization also includes a field *depth*.

We see from the example in Figure 2.12 that Modula-3 supports *partial revelation* and *multiple views* of object types. A client of the *Stack* abstraction only needs to see the abstract *Stack* interface, whereas an implementer also needs the information in *StackRep*.

What is particularly interesting in the light of this thesis is that partial revelation makes problems for translating systems for Modula-3. Consider, for example, a module M which declares a subtype $M.T$ of $Stack.T$. Furthermore, assume that $M.T$ also declares a data field f. Since the most efficient implementation of single inheritance requires that an object type's local data immediately follow the data of its supertypes, the Modula-3 compiler needs

```
INTERFACE Stack;
   TYPE T <: ROOT;
   PROCEDURE Create () : T;
   PROCEDURE Push (VAR s : T; e : INTEGER);
   PROCEDURE Pop (VAR s : T) : INTEGER;
   PROCEDURE Depth (s : T) : INTEGER;
END Stack.

INTERFACE StackRep; IMPORT Stack;
   TYPE Link = OBJECT item : INTEGER; next : Stack.T END;
   REVEAL Stack.T <: Link;
END StackRep.

MODULE StackImpl EXPORTS Stack; IMPORT StackRep;
   REVEAL T = StackRep.Link BRANDED OBJECT depth : INTEGER; END;
   (* Implementations of Create, Push, Pop, and Depth ... *)
BEGIN END StackImpl.

MODULE StackUse EXPORTS Main; IMPORT Stack;
   VAR S : Stack.T; D : INTEGER;
BEGIN
   S := Stack.Create (); Stack.Push (S, 314); D := Stack.Depth (S);
END StackUse.
```

Figure 2.12: Modula-3 stack example, adapted from Nelson [160]. Since Modula-3 supports garbage collection, no explicit deallocation routine needs to be called.

to know the size of *Stack.T*'s data record in order to compute the offset of f in *M.T*'s data record. However, even if the compiler could determine (by inspection of *StackRep*) that *Stack.T* contains the fields *next* and *item*, it could not know that it also contains the field *depth* since this fact is hidden within *StackImpl*. Hence, there are cases when a Modula-3 translating system cannot at compile-time compute the offsets of record fields. SRC Modula-3, currently the only available Modula-3 implementation, solves this by performing these computations at "logical link-time", i.e. at run-time prior to the execution of any user code. This pass (which also includes checks for illegal revelations and the construction of object type templates) can be expensive in itself, but the scheme also in general slows down run-time field accesses, since these have to be preceded by a table lookup to determine actual field offsets.

In addition to what we have seen from Figure 2.12, Modula-3 supports a simple form of unconstrained generic modules, threads (lightweight processes),

and exceptions.

2.8.10 Zuse

We will conclude this survey with a sneak preview of the language Zuse, which will be introduced in detail in the next chapter. In short, Zuse (like Modula-2 and Modula-3) has fully abstract types, but (unlike Modula-2 and Modula-3) Zuse does not pose any restrictions on the realizations of such types. Furthermore, Zuse (like Ada) supports abstract inline procedures, but (unlike Ada) does not allow compilation dependencies between implementation units. The effect of these features on a real example can be seen from the Zuse Stack example in Figure 2.13: The stack specification unit does not leak any information regarding the realization of the abstract type nor the inline-status of the exported procedures, and once the Stack specification unit has been compiled the corresponding implementation unit and the client module StackUse can be compiled in an arbitrary order. Furthermore, since Stack`T is realized as a statically allocated type, its use will not tax the dynamic allocation system at all.

Zuse also includes features for object-oriented programming similar to those found in Modula-3. However, unlike the SRC Modula-3 implementation discussed in Section 2.8.9, the Zuse translating systems (presented in Chapters 5 through 7) assure that these features do not degrade the run-time performance of object-oriented programs.

2.9 Summary

In this chapter we have examined some of the module and abstraction concepts which have been proposed for imperative and object-oriented languages. We have paid particular attention to how different languages have approached the problems inherent in the combination of encapsulation and separate compilation. We have shown that most language designs compromise orthogonality of encapsulation in favor of ease of implementation, and sometimes at the expense of reduced run-time efficiency. We summarize our findings in the table on page 46. The abbreviations used in the table are explained at the bottom of page 45.

```
SPECIFICATION Stack;
  TYPE      T                        = { * :{ :=, =: } };
  CONSTANT MaxDepth : CARDINAL =;

  PROCEDURE Create : () RETURN T =;
  PROCEDURE Push : (REF S : T; e : CARDINAL) =;
  PROCEDURE Pop : (REF S : T) RETURN CARDINAL =;
  PROCEDURE Depth : (S : T) RETURN T =;
END Stack.

IMPLEMENTATION Stack;
  CONSTANT MaxDepth : CARDINAL += 100;
  TYPE Vector == ARRAY [RANGE CARDINAL [1 .. MaxDepth]] [CARDINAL];
  TYPE T      += { { .  } } RECORD [S : Vector; D : CARDINAL ];

  PROCEDURE Create : () RETURN T +=···
  INLINE PROCEDURE Push : (REF S : T; e : CARDINAL) +=···
  INLINE PROCEDURE Pop : (REF S : T) RETURN CARDINAL +=···
  PROCEDURE Depth : (S : T) RETURN T +=···
END Stack.

PROGRAM StackUse;
  IMPORT Stack;
  VARIABLE S : Stack`T;
BEGIN S := Stack`Create(); Stack`Push (314); END StackUse.
```

Figure 2.13: ZUSE stack example. Since Stack`T is a static type, no explicit deallocation routine needs to be called.

r-res	:	realization restriction	r-rev	:	realization revelation
r-pro	:	realization protection	p-rev	:	partial revelation
i-ord	:	implementation ordering	b-sup	:	binding-time support
r-sup	:	run-time support			
o-b	:	object-based	c-b	:	class-based
o-o	:	object-oriented			
u-gen	:	unconstrained generic	c-gen	:	constrained generic
i-pol	:	inclusion polymorphism	param	:	parametric
1-1	:	one-to-one	0-1	:	zero-to-one
m-m	:	many-to-many	1-01	:	one-to-at-most-one
spec	:	specification	impl	:	implementation
(i)proc	:	(inline) procedure	mod	:	module
stat	:	static allocation of abstract types	dyn	:	untraced dynamic allocation of abstract types
GC	:	traced dynamic allocation of abstract types			

LANGUAGE	MODULE SYSTEM	ENCAPSULATION				POLY-MORPHISM	COMP. UNITS	REFERENCES
		TYPE	CONST	IPROC	ALLOC			
Ada	o-b, 1-01	r-pro, r-res	r-pro	i-ord	stat, dyn	c-gen mod & proc, param types	spec & impl, proc	[106, 179, 197, 28, 2, 3, 4]
Albericht	o-o, 1-1	r-sup			GC	i-pol	spec & impl	[163]
Alphard	c-b, 1-1	r-pro	r-pro		stat	c-gen mod	any item	[194]
Beta	o-o, m-m	r-rev		r-rev	stat, GC	i-pol	any item	[127, 132, 155, 156]
CHILL	o-b, 0-1	r-pro			stat		mod	[35, 197]
CLU	c-b, 0-1	r-rev	r-rev	r-rev	GC	c-gen mod	mod	[13, 142, 140]
C++	o-o, 1-1	r-pro	r-pro	r-rev	stat, dyn	u-gen classes & proc, i-pol	mod	[1]
Eiffel	o-o, 0-1	r-pro			GC	c-gen & u-gen classes, i-pol	classes	[151, 152, 154, 162]
Euclid	c-b, 0-1	r-pro	r-pro		stat	param types	unspecified	[135, 232, 103]
LOGLAN	o-o, 1-1	r-rev			GC	i-pol	classes	[131]
Mesa	o-b, m-m	p-rev, r-res, r-pro	r-rev	r-rev	stat, dyn		mod	[81, 235, 204]
Milano-Pascal	o-b,	b-sup	b-sup		stat		mod, proc	[45, 46, 164]
Standard ML	o-b, 1-1	r-res	r-rev		GC	param	unspecified	[91, 90, 185, 238, 210]
Modula-2	o-b, 1-1	r-res	r-rev		dyn		spec & impl	[223, 37]
Modula-2+	o-b, 1-1	r-res	r-rev		dyn, GC		spec & impl	[182, 118, 231, 233]
Modula-3	o-o, m-m	r-res, r-sup	r-rev		dyn, GC	u-gen mod, i-pol	spec & impl	[41, 42, 116, 160, 114, 40]
Oberon-1	o-o, 1-1	p-rev, r-rev	r-rev		stat, GC	i-pol	spec & impl	[226, 224, 225]
Oberon-2	o-o, 0-1	r-rev	r-rev		GC	i-pol	mod	[227, 228, 159, 158]
Object-Oberon	o-o, 1-1	r-rev	r-rev		GC	i-pol	mod	[157]
Turing	o-b, 0-1	r-rev	r-rev		stat		none	[101, 99, 102]
Turing Plus	o-b, 1-1	r-pro	r-rev		stat		spec & impl	[100]
Zuse	o-o, 1-1	r-pro, b-sup, r-res	r-pro, b-sup	r-rev, b-sup	stat, dyn	i-pol	spec & impl	

Chapter 3

A Language with Flexible Encapsulation

> **bondage-and-discipline language:** *A language* [···] *that, though ostensibly general-purpose, is designed so as to enforce an author's theory of 'right programming'* [···]
>
> *The New Hacker's Dictionary [173]*

3.1 Introduction

Modern programming languages display an amazing array of features in support of abstraction and encapsulation. The previous chapter examined some of the mechanisms which have been proposed, and discussed the penalties they introduce with respect to translation and execution time. This chapter will introduce an experimental language design – a close descendant of earlier languages such as Ada, Mesa, Modula-2, and Modula-3 – with orthogonal[1] and flexible facilities for hiding representational details of exported items. Subsequent chapters will discuss necessary and desirable properties of translators for this language, and will closely examine four concrete translating system designs.

This section will give some background information and state the rationale behind the language design. Sections 1.2 through 1.6 will present an informal account of the facilities provided by the language with respect to the flexible way in which representational details of exported items may be hidden or revealed.

[1] "Principle of *orthogonality*: language features can be composed in a free and uniform manner with predictable effects and without limitations. [83]"

Section 1.7 will examine several Zuse programs as examples of how the different kinds of encapsulation provided by the language can be put to use.

3.1.1 Language Design Rationale

The main objective of this treatise is to examine the trade-offs between language facilities for abstraction and encapsulation and the penalties in translation and execution-time they may incur. Mainly we will be interested in the way information about items exported by a module may be divided between the module's specification and implementation parts, and how this might affect the design of a translating system for the language. To this end the remainder of this chapter will examine an experimental modular language with flexible encapsulation facilities and which is tentatively named Zuse[2]. The language design has been guided by these four general principles:

1. The language should include facilities for making any exported item fully abstract, semi-abstract, or concrete. In other words, it must be possible to defer all, none, or some of the representational details of any kind of exported item to the implementation unit. Apart from being basic to sound software engineering practices, full encapsulation is also essential in order to avoid trickle-down recompilations.

2. The cost (in terms of translation-time, execution-time and storage) of using an abstract or semi-abstract item should be no greater than it would have been if the same item had been concrete.

3. The language design should not allow compilation dependencies to exist between module implementation units and should furthermore require the bulk of intra- and inter-modular static semantic checking to be performed at compile-time. These rules minimize the risk of very long compilation and execution times and are also essential in order to allow programmers to work independently in team programming situations.

4. The language should allow a high degree of freedom in the way exported items are realized, including the use of low-level facilities. It should always be possible, however, to ensure the integrity of exported items by protecting them from abuse by importers.

The first three items on this list may be paraphrased: "Full and flexible encapsulation should not incur any translation-time or execution-time overhead."

[2]for Konrad Zuse, the first modern programming language designer (see Zuse [239] and Bauer [19]).

We believe this to be an important principle, and it will permeate this entire thesis. Our reason for being very firm on this point is that full encapsulation does not make a language more *expressive* (i.e. it does not allow one to express designs that could not have been expressed in a language without encapsulation), only *safer*. Translation-time or execution-time overhead induced solely by a language's facilities for encapsulation is therefore unacceptable.

3.1.2 Providing Binding-Time Information

Subsequent chapters will examine in detail possible translation systems for Zuse, but we will offer some preliminary information here in order to provide a framework in which to discuss the kinds of constructs that the language design may allow.

The language design which will be presented in this chapter implicitly rests on the assumption that the standard tools requirement (Section 2.5) has been relaxed; i.e. that non-standard translation systems are deemed acceptable. Specifically, we will assume that the translation of a set of source code modules into an executable program goes through a stage – temporally located after the compilation of all module implementations but prior to program execution – during which inter-modular information may be exchanged. This stage, which we will call the *pasting stage*, may either coexist with existing system link-editors and linking-loaders or completely or in part take over the tasks traditionally performed by these systems programs (see Figure 1.1.2). In other words, a translating system for Zuse is assumed to fall under the **binding-time support** heading in the list of approaches to encapsulation given in Section 2.5.

The exact division of labor between compilers, *pasters* (the program active during the pasting stage), linkers, and loaders will not be defined here, but will be the subject for later discussion. For now, it suffices to restate the third language design principle above, namely that a compiler for the Zuse language is assumed to perform the bulk of syntactic and semantic analysis. In other words, the static semantic correctness of a Zuse program should be decided prior to the program being submitted for pasting.

3.2 The Zuse Module System and Concrete Export

Zuse contains the array of basic concepts found in most modular imperative and object-oriented languages and shares with these languages a common view of data and execution. Our exposition in this and the following sections will therefore focus on the area in which Zuse differs from similar languages, namely the facilities available for the protection of the integrity of exported items. We

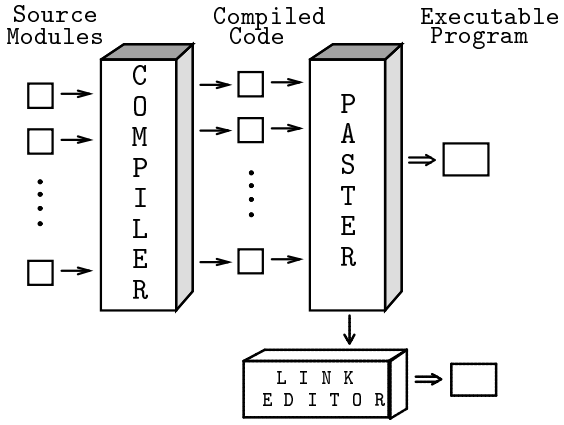


Figure 3.1: Proposed translation system.

will have little to say about other aspects of the language, such as the control structures, primitive types, and standard interfaces available, and will simply assume that the constructs necessary for systems programming are available in some form or another. We will furthermore not be concerned with the dynamic semantics of the language, unless affected by the proposed encapsulation schemes.

Nevertheless, in this section we will provide some preliminary information about the Zuse module and type systems, to serve as background material to the following sections, which will discuss the encapsulation system.

3.2.1 Modules and Separate Compilation

A Zuse module is *untyped* and made up of two units, the *specification* and the *implementation*. There is one *program* module, consisting only of one (implementation) unit, and whose module body is the last to be executed. Zuse's module system is hence of the *one-to-one* kind, and identical to that found in Modula-2.

A module may import items from other modules by listing the names of the imported modules in import-clauses at the top of a module unit. Individual items in the imported modules are referred to using the notation M`v, where M is a module and v is an item exported by M. Figure 1.2 below shows a program module P importing a separately compiled module M.

The two module units are compiled separately, with the specification compiled before the implementation. Furthermore, a unit importing a particular

module will have to be compiled *after* the specification part of that module has been compiled. These are the only rules governing the order in which module units are compiled. Consequently, once all the specification units of the modules making up a program have been compiled, the corresponding implementation units may be compiled in an arbitrary order. Further changes which do not modify any specification unit only result in the recompilation of the affected implementation units.

```
SPECIFICATION M;
  VARIABLE v : INTEGER;
END M.
```

```
IMPLEMENTATION M;
BEGIN
  v := 5;
END M.
```

```
PROGRAM P;
  IMPORT M;
BEGIN
  M`v := 1;
END P.
```

Figure 3.2: A simple Zuse program consisting of the program module P and the separately compiled module M.

3.2.2 Concrete Types

Concrete items are characterized by the fact that their definition as well as their realization are contained in the specification unit of a module. We will now introduce the Zuse type system and the syntax and informal semantics of concrete export of types, constants, and procedures.

Zuse shares most of its basic type system with Modula-2: there are equivalence types, enumeration types, subrange types, records, arrays, sets, pointers, and procedure types. The object type is inherited from Modula-3. The concrete syntax of the different types is given in Figure 1.3. The concreteness of the types is evident from the use of the special *immutable* definition symbol '==', which signifies that the realization as it is given in the specification unit is complete. Note that an object type comes in three parts: an optional supertype (preceded by the keyword WITH), a sequence of fields (the first set of brackets), and a sequence of methods (second set of brackets).

Type compatibility is *by name*, except for subrange types, which are compatible if their parent types are compatible; procedure types, which are compatible if the type, order, and mode of their formal parameters are compatible; and object types, which are compatible along their chain of inheritance.

In addition to the types inherited from Modula-2 and Modula-3, Zuse includes a special UNIQUE-type, similar to Ada's **new** type, but with a somewhat different semantics. Consider UniqueT exported by ConcreteT in Figure 1.3.

```
SPECIFICATION ConcreteT;
  TYPE
    EquivT   == CHAR;
    EnumT    == ENUM [Red, Blue, Green];
    RangeT   == RANGE EnumT [EnumT!Red .. EnumT!Blue];
    RecordT  == RECORD [x,y : BOOLEAN ];
    ArrayT   == ARRAY  [RangeT]  [CHAR];
    SetT     == SET [EnumT];
    PointerT == REF [RecordT];
    ProcT    == (a : EnumT; REF b : ARRAY OF  SetT);
    FunT     == (REF x : RangeT) RETURN CARDINAL;
    ObjectT  == OBJECT
                  [a  : INTEGER ; c  :  CHAR]
                  [P : (x : EnumT)] ;
    SubT     == WITH ObjectT OBJECT
                  [b : BOOLEAN ]
                  [Q : (z : CHAR)] ;
    UniqueT  == UNIQUE [CARDINAL];
END ConcreteT.
```

Figure 3.3: Zuse concrete types.

Inside the module ConcreteT, UniqueT behaves exactly like a CARDINAL (in other words, as if it had been an equivalence type), whereas to an importer ConcreteT`UniqueT is a type compatible only with itself. In other words, an importer may add two variables of type ConcreteT`UniqueT, but may not add a variable of type ConcreteT`UniqueT to a variable or literal of type CARDINAL. Unique types are mostly used in conjunction with *type protection*, which will be described in Section 1.6.

The syntax and semantics of basic operations on expressions and designators parallel those in comparable languages. In addition to arithmetic operations on variables of arithmetic types, field selection on records and objects (RecordName.FieldName), indexing on arrays (ArrayName[ExpressionList]), etc., basic Zuse operations include bitwise assignment, comparison (=, # (inequality), <, <=, $\cdots$), and type-coercion (TypeName<<Expression>>[3]). These operations are allowed on variables which are of an imported concrete type, as well as on variables of a locally declared type.

[3] T<<x>> is a designator when x is a designator and en expression when x is an expression. T<<x>> is legal only when the representations of T and x have the same size.

3.2.3 Concrete Constants

Zuse supports manifest (named) constants in a style similar to Modula-3 and Ada, i.e. scalar as well as structured constants are allowed. Unlike these languages, however, Zuse also supports *reference* constants, i.e. pointers to constant values. These are mainly used in conjunction with abstract types. All named constants must be given an explicit type, and concrete constants are signified by the immutable definition symbol '=='. Figure 1.4 gives a few examples.

```
SPECIFICATION ConcreteC;
  IMPORT ConcreteT;

  CONSTANT
     ScalarC  : CARDINAL          == 666;
     RecordC  : ConcreteT`RecordT == RECORD [
                                       x :== TRUE ;
                                       y :== FALSE ];
     PointerC : ConcreteT`PointerT == REF [RECORD [
                                       x :== FALSE ;
                                       y :== TRUE ]];
     ArrayC   : ConcreteT`ArrayT  == ARRAY [
                                       [ConcreteT`EnumT!Red]  :== "R";
                                       [ConcreteT`EnumT!Blue] :== "B"];
     SetC     : ConcreteT`SetT    == SET [ConcreteT`EnumT!Red];
END ConcreteC.
```

Figure 3.4: Zuse concrete constants.

3.2.4 Concrete Procedures

A Zuse procedure declaration is a simple variant of a named constant declaration, where the type (which may be named or anonymous) is the procedure's signature, the '==' marks concreteness, and the "value" of the procedure is its code and local data. The presence or absence of the keyword "INLINE" indicates the kind of linkage which should be employed for calls to the procedure (see Figure 1.5). The transfer modes available for actual parameters are pass-by-value and pass-by-reference (REF).

Zuse procedures may be nested, but a procedure may only reference its own formal parameters, variables declared locally, and global variables. Recursion is disallowed for inline procedures, both directly as well as indirectly through other inline procedures.

```
SPECIFICATION ConcreteP;
  IMPORT ConcreteT;

  PROCEDURE P1 : ConcreteT`FunT ==
  BEGIN
    x := ConcreteT`EnumT!Red;
    RETURN ORD(x);
  END P1;

  INLINE PROCEDURE P2 : (
    a      : ConcreteT`EnumT;
    REF b : BOOLEAN) ==
  BEGIN
    b := a = ConcreteT`EnumT!Red;
  END P2;
END ConcreteP.
```

Figure 3.5: Zuse concrete procedures.

3.3 Abstract Export

In accordance with the design rationale on page 2, Zuse provides full encapsulation of all items: types, constants, and (inline) procedures. Unlike other languages, Zuse does not allow the integrity of encapsulated items to be compromised in any way. In other words, none of the approaches to encapsulation (see Section 2.5) employed by previous language designs are allowed to restrict the full encapsulation capabilities of Zuse: *partial revelation* (Mesa and Oberon-1 abstract types), *realization restriction* (Modula-2 abstract types), *run-time support* (Modula-3 abstract object types), and *implementation ordering* (Ada inline procedures) are hence all ruled out.

3.3.1 Abstract Types

The statically allocated, *hidden* type is the most important concept introduced in Zuse . Figure 1.6 shows the declaration of a hidden type, with a few different possible realizations. The *mutable* definition symbol '=' is used to signify that the declaration of T in the specification is incomplete; '+=' (the *extension* symbol) indicates that a declaration in an implementation unit extends a previous one in the specification.

The operations allowed on a variable of an imported hidden type are the same as those for concrete types, with the exception of operations which require knowledge of the structure of the realization of the type, such as field selection

```
SPECIFICATION AbstractT;
  TYPE
    T = ;
  VARIABLE
    v : T;
END AbstractT.
```

```
IMPLEMENTATION AbstractT;
  TYPE
    T += INTEGER;
    — or —
    T += RECORD [a,b : INTEGER ];
    — or —
    R == RANGE CHAR ["A" .. "Z"];
    T += ARRAY [R] [CHAR];
    — or —
    T += REF [INTEGER];
END AbstractT.
```

Figure 3.6: A Zuse module exporting a hidden type with four possible realizations.

and array indexing. Consequentially, it is legal for an importer of a hidden type to test two variables of the type for equality (<, <=, etc. are also allowed), as well as to coerce a variable of the type into a variable of another type, subject to the restriction that the two variables are of equal size. The section on *Protected Types* below will discuss these matters further.

3.3.2 Abstract Constants

A hidden constant (Figure 1.7) is declared much in the same way as a hidden type: the specification unit defines the constant's name and type, whereas the realization (value) is deferred to the implementation unit. Note that the type of a hidden constant can itself be either abstract or concrete. Figure 1.7 also explains why Zuse, in contrast to most other languages, allows manifest pointer constants: without them an abstract type would be restricted to non-pointer realizations if it was accompanied by an abstract constant.

3.3.3 Abstract Procedures

There are two aspects of a Zuse procedure which may be hidden: the procedure's body and its linkage (inline-status). Figure 1.8 is a variant of the module ConcreteP where both procedures' code and inline-status have been hidden.

3.4 Semi-Abstract Export

For reasons of program logic or (translation) efficiency it may sometimes be desirable to make some parts of the realization of an exported item known to

```
SPECIFICATION AbstractC;
  CONSTANT
    Pure : REAL =;

  TYPE
    T = ;
  CONSTANT
    S : T =;
END AbstractC.
```

```
IMPLEMENTATION AbstractC;
  CONSTANT Pure : REAL += 99.44;

  TYPE
    T += RECORD [a,b : INTEGER ];
  CONSTANT
    S : T += RECORD [a :== 4; b :== 2];
    — or —
  TYPE      T    += REF [INTEGER];
  CONSTANT S : T += REF  [42];
END AbstractC.
```

Figure 3.7: Zuse hidden constants.

```
SPECIFICATION AbstractP;
  IMPORT ConcreteT;

  PROCEDURE P1 : ConcreteT`FunT =;

  PROCEDURE P2 : (
    a      : ConcreteT`EnumT;
    REF b : BOOLEAN) =;
END AbstractP.
```

```
IMPLEMENTATION AbstractP;
  IMPORT ConcreteT;

  PROCEDURE P1 : ConcreteT`FunT +=
  BEGIN — as before — END P1;

  INLINE PROCEDURE P2 : (
    a      : ConcreteT`EnumT;
    REF b : BOOLEAN) +=
  BEGIN — as before — END P2;
END AbstractP.
```

Figure 3.8: Zuse abstract procedures.

users of the item while keeping other parts private to the implementation. For this reason Zuse supports *semi-abstract* constants and procedures, and two kinds of semi-abstract types. The realization of a semi-abstract item comes in two parts: the *visible part* which is given in the specification unit, and the *hidden part* which is deferred to the implementation unit.

3.4.1 Semi-Abstract Types

Semi-abstract types come in two flavors: *semi-hidden* and *extensible*. A semi-hidden type is a hidden type whose definition consists of an indication of the type-class to which a realization of the type must belong, effectively restricting the range of possible realizations. A semi-abstract extensible type, on the other hand, is a type where the realization as it is given in the module specification

may be extended in the implementation unit.

There are four Zuse semi-hidden types: hidden pointer, subrange, object, and enumerable types. A hidden pointer type (which is equivalent to Modula-2's and Mesa's opaque type and Ada's deferred access type) may only be realized by a pointer or address type. A hidden object type is equivalent to Modula-3's opaque object type. A hidden subrange type reveals its type in the specification unit but defers its range to the implementation. A hidden enumerable type may be realized only by an ordinal type, i.e. by any of the built-in enumerable types (integer, character, boolean, etc.), by enumeration types, or by subranges of these types. Figure 1.9 gives examples of semi-hidden types.

There are four extensible types: record, enumeration, object, and subrange types. An extensible record (which is similar to Oberon-1's public projection type) may reveal some fields in the specification unit and may extend these with additional fields in the implementation unit. Similarly, an implementation unit may add identifiers to an extensible enumeration type or widen the range of an extensible subrange type. The fields, methods, and the supertype of an extensible object type may all either be revealed or hidden. Note, however, that an object may be given a supertype either in the specification or the implementation part of a module, but not both.

An importer of an extensible type is restricted to use only the components of the type which were given in the specification unit, although any variable of an extensible type will, in actuality, have all components of the complete type. Figure 1.10 gives examples of extensible types.

One should note an important difference between the two kinds of semi-abstract types: a semi-hidden type is *required* to receive a complete realization in the implementation unit, whereas an extensible type *may* be extended in the implementation.

3.4.2 Semi-Abstract Constants

The value of Zuse structured constants may be arbitrarily divided between the specification and implementation units. Note that the type of an *extensible constant* may itself be extensible or concrete. Figure 1.10 gives examples of extensible constants.

3.4.3 Semi-Abstract Procedures

In Zuse the code of a procedure cannot be divided arbitrarily between the specification and implementation unit the way, for example, the value of an extensible structured constant may. Rather, the body of an exported procedure

```
SPECIFICATION SemiHidden;
  TYPE
    EnumT  = < > ;
    RangeT  = RANGE INTEGER ;
    PointerT = REF;
    ObjectT  = OBJECT;
  VARIABLE
    v        :  EnumT;
END SemiHidden.

IMPLEMENTATION SemiHidden;
  IMPORT SYSTEM;

  TYPE
    EnumT  += INTEGER;
    RangeT  += RANGE INTEGER [-14 .. 159];
    PointerT += REF [INTEGER];
    ObjectT  += OBJECT [a : INTEGER] [ ];
    — or —
    EnumT  += RANGE CHAR ["A" .. "Z"];
    RangeT  += RANGE INTEGER [0 .. 5];
    PointerT += SYSTEM`ADDRESS;
    — or —
    EnumT  += ENUM [Red, Blue, Green];
    RangeT  += RANGE INTEGER [1 .. 1];
    PointerT += REF [RECORD [
                    a,b : EnumT;
                    x   : PointerT ]];
END SemiHidden.

PROGRAM SemiHiddenImport;
  IMPORT SemiHidden;

  VARIABLE
    S : SET [SemiHidden`EnumT];
    R : ARRAY  [SemiHidden`RangeT]  [BOOLEAN];
BEGIN
  INCL (S, SemiHidden`v);
  R [15] := TRUE;
END SemiHiddenImport.
```

Figure 3.9: Zuse semi-hidden types.

```
SPECIFICATION ExtendedT;
  TYPE
     EnumT  = ENUM [Red, Blue, Green];
     RangeT  = RANGE CARDINAL [10 .. 20];
     RecordT = RECORD [x,y : EnumT ];
     ObjectT = OBJECT
                  [v : INTEGER ]
                  [R : (x : CHAR)] ;
END ExtendedT.

IMPLEMENTATION ExtendedT;
  IMPORT ConcreteT;
  TYPE
     EnumT  += ENUM [Yellow, Orange];
     RangeT  += RANGE CARDINAL [0 .. 30];
     RecordT += RECORD [z : REAL ];
     ObjectT += WITH ConcreteT`SubObjectT OBJECT
                    [r  : RangeT ]
                    [U : (z : CHAR)];
  VARIABLE X :  SET [EnumT] ;
            Y :  ARRAY  [RangeT]  [CARDINAL];
            Z :  RecordT;
BEGIN
  INCL (X, EnumT!Orange);
  Y [25] := 5;
  Z.z := 3.14;
END ExtendedT.

PROGRAM P;
  IMPORT ExtendedT;
  VARIABLE A :  SET [ExtendedT`EnumT] ;
            B :  ARRAY  [ExtendedT`RangeT]  [CARDINAL];
            C :  ExtendedT`RecordT;
BEGIN
  (* legal: *) INCL (A, ExtendedT`EnumT!Green);
               B [12] := 5;
               C.x := ExtendedT`EnumT!Red;
  (* illegal: *) INCL (A, ExtendedT`EnumT!Orange);
                 B [25] := 5;
                 C.z := 3.14;
END P.
```

Figure 3.10: Type extension examples. A client of module ExtendedT has no knowledge of the extended parts of the types and consequently cannot reference them.

```
SPECIFICATION ExtendedC;
  TYPE
    RecordT =    RECORD [x : CHAR ];
    SetT    == SET [CHAR];
  CONSTANT
    R : RecordT = RECORD [x :== "R"];
    S : SetT    = SET ["A" .. "Z"];
END ExtendedC.
```

```
IMPLEMENTATION ExtendedC;
  TYPE
    RecordT += RECORD [y : INTEGER ];
  CONSTANT
    R : RecordT += RECORD [y :== 5];
    S : SetT    += SET ["a" .. "z"];
END ExtendedC.
```

Figure 3.11: Zuse extensible constants.

must be given in its entirety in one of the units. However, Zuse allows the linkage of an otherwise hidden procedure to be either hidden or revealed (see Figure 1.12). An exported procedure with a hidden body but revealed linkage will be known as a *semi-hidden* procedure.

```
SPECIFICATION SemiHiddenP;
  IMPORT ConcreteT;

  NOT INLINE
  PROCEDURE P1 : ConcreteT`FunT =;

  INLINE PROCEDURE P2 : (
    a    : EnumT;
    REF b : BOOLEAN) =;
END SemiHiddenP.
```

```
IMPLEMENTATION SemiHiddenP;
  IMPORT ConcreteT;

  PROCEDURE P1 : ConcreteT`FunT +=
  BEGIN — as before — END P1;

  INLINE PROCEDURE P2 : (
    a    : EnumT;
    REF b : BOOLEAN) +=
  BEGIN — as before — END P2;
END SemiHiddenP.
```

Figure 3.12: Zuse semi-abstract procedures. P1 may not be realized as an inline procedure, whereas P2 must be.

3.5 Automatic Initialization

A Zuse type may be coupled with a default value (a constant expression) which will be the initial value of any variable of that type. Automatic variable initialization of this kind is a convenient concept – supported with some variety in languages such as Ada, Mesa, and Modula-3 – which relieves programmers of the burden of manual initialization. The examples in Section 1.7 will make it evident that without automatic initialization Zuse's extensible types would

be rendered virtually useless.

ZUSE's initialization concept is built on the one found in Mesa [235, pp. 3-45 – 3-47], with the extension that initialization clauses may be hidden or (partially) revealed. An exported concrete or semi-abstract type can be given a *concrete default value* using the *immutable default* symbol ':==', or an *abstract* or *semi-abstract default value* using the *extensible default* symbol ':='. A default value is extended in the implementation unit using the *extension default* symbol ':+='.

Three additional rules determine the legality of an initialization clause:

- A type or a component of a type which has been given a default value in the specification unit may not be given a different default value in the implementation.
- In the absence of an overriding initialization clause, equivalence and subrange types inherit the default values of their base types.
- The default value of a structured type is the combination of the default values of the components.

Figure 1.13 gives some examples of legal initialization clauses.

3.6 Type Protection

Further discussions in subsequent chapters will make it clear that the indiscriminate use of ZUSE abstract items is likely to incur a translation-time penalty in the form of excessive module binding-times. The reason is that a compiler translating a module which imports abstract items will not have available the necessary information about these items in order to complete the translation. Hence some of the work will have to be deferred to module binding-time. At the expense of sacrificing encapsulation it is, as has already been mentioned, possible to reveal some or all of the realization of an item by using semi-abstract or concrete export. This can reduce the amount of work necessary at module binding-time.

However, the use of concrete or semi-abstract export will lead to less safe programs, since clients of such items are free to manipulate the items' visible parts. One way to overcome this problem is to impose restrictions on the kinds of operations a client may perform on the visible part of an imported item. This is, as we have seen in Sections 2.5 and 2.8, the chosen method in many languages. ZUSE supports a *type protection* construct based on a concept proposed by Liskov and Jones [109, 110, 111], which allows modules

```
SPECIFICATION Initialization;
  TYPE
    T1 == CHAR :== "A";
    T2 == T1 := ;
    T3 == T1 := ;
    T4 =    := ;
    T5 == RECORD [a : T1; b : T2 ];
    T6 =   RECORD [a : T1 ] := ;
    T7 == OBJECT
              [a : INTEGER :== 49;
               b : INTEGER ]
              [P : (x : CHAR)]  := ;
    Idx == RANGE CARDINAL [0 .. 30];
    T8 == ARRAY  [Idx]  [T1];
END Initialization.
```

```
IMPLEMENTATION Initialization;
  TYPE
    T3 :+= "B";
    T4 +=  RECORD [
              a : T1 :== "X";
              b : REAL ];
    T6 +=  RECORD [b : REAL ]  :==
           RECORD [a :== "R"; b :== 3.1];
    T7 :+= OBJECT [b :== 9] [P :== P1];

  PROCEDURE P1 : (
    self : T7;
    x    : CHAR) ==
  BEGIN ··· END P1;
END Initialization.
```

Figure 3.13: Examples of Zuse initialization clauses. Variables of type T1 will receive the initial value "A", as will variables of type T2 since T2 does not have its own initialization clause. Type T3 overrides T1's default value, and consequently variables of type T3 will receive the initial value "B". T8 will be initialized to an array of "A"'s.

considerable freedom in expressing how different clients may manipulate their exported items.

Zuse type protection clauses generalize and extend many of the access control mechanisms found in popular languages: Ada's *private* and *limited private* types, C++'s *friend* concept, Alphard's [194] *access control*, Gypsy's *access lists* [10], and the *provide* and *request* clauses of PIC/Ada [229]. Their main novelty is the way in which they may be combined with Zuse 's abstract and semi-abstract types to accurately control the manner in which different clients may access and manipulate different parts of an exported item.

3.6.1 The Syntax of Protection Clauses

Every exported type in Zuse may be coupled with a *protection* clause listing the operations a particular importer may perform on items of that type. Examples of operations which may be part of a protection clause include assignment, component selection, and type coercion. All kinds of Zuse types may be protected, including abstract- and semi-abstract types, allowing for very fine-grained control over the way a module's clients may manipulate the module's exported

items. Although types are the only items which may receive an explicit protection clause, other exported items (variables, constants, and procedures) may be implicitly protected by protecting their type.

A Zuse type protection clause is a list of *protection items*, where each item consists of a *friend list* and an *operator list*. The friend list describes the modules to which the protection item applies ('(M, N) :' means modules M and N, '∗ :' means "all modules", and an empty friend list means the exporting module itself), and the operator list describes the privileges granted to these modules. The general form of a Zuse type protection clause is given in Figure 1.14.

⟨*prot_clause*⟩ ::= '{' ⟨*prot_item*⟩ { ',' ⟨*prot_item*⟩ } '}' .
⟨*prot_item*⟩ ::= ⟨*friend_list*⟩ ⟨*op_list*⟩ .
⟨*friend_list*⟩ ::= ['(' ⟨*module_list*⟩ '):' | '∗ :'] .
⟨*module_list*⟩ ::= ⟨*ident*⟩ {',' ⟨*ident*⟩}.
⟨*op_list*⟩ ::= '{' ⟨*op*⟩ { ',' ⟨*op*⟩ } '}'.
⟨*op*⟩ ::= '+' | '−' | '<< >>' | ':=' | '=:' | '( )' | '. ' | 'WITH' | ···.

Figure 3.14: Extended BNF syntax of Zuse's type protection clauses.

The protection item (M, N) :{ :=, =: } applied to a type T bars modules M and N from applying any operations on variables of type T other than assignment (:=). In particular, if T is a record, M and N will not be able to access any of T's fields, although they will be able to make copies of variables of type T. Similarly, the protection item ∗ :{ +, << >> } allows all modules to perform additions and type coercions on variables of the protected type, { ( ) } allows the exporting module to perform procedure calls, and (M) :{ WITH, . } grants M permission to inherit and access the methods and instance variables of the protected object type.

3.6.2 The Applications of Type Protection

The concrete examples in Figure 1.15 and in Section 1.7.5 show that type protection in Zuse has several important applications in addition to restricting clients' access to concrete items. The most common application of protection in Zuse is to restrict the operations clients may perform on variables of an imported abstract type. This is an essential application of protection, since Zuse grants clients full access to imported items by default, regardless of whether the item is concrete, semi-abstract, or abstract. In most cases a module which

declares an abstract type will restrict all clients' access to variables of the type to assignment (∗ :{ :=, =: }). In exceptional cases, however, special clients (*friends*) may be allowed certain extra privileges, such as type coercion and bit-wise comparison ({ ∗ :{ :=, =: } }, (F) :{ << >>, = }). Such practices should, of course, be strongly discouraged, but when on occasion their need does arise, Zuse requires the friendships to be made explicit.

```
SPECIFICATION Protection;
  TYPE
    Hidden  =   { (M, N) :{ :=, =: } };
    Address == { ∗ :{ :=, =:, +, −, = } } UNIQUE [CARDINAL];
    Open    == { ∗ :{ :=, =: } } RECORD [c,d : CHAR ];
    Extend  =   { ∗ :{ . , =: } } RECORD [
                  a : { ∗ :{ :=, =: } } CHAR;
                  b : { ∗ :{ =: }, { := } } CHAR ];
    Wrong   == { (M, Q) :{ := } } RECORD [x : Hidden ];

  VARIABLE
    OK :  { ∗ :{ =: }, { := } } BOOLEAN;

  PROCEDURE P : { (M) :{ ( ) } }(a : CHAR) =;
END Protection.

IMPLEMENTATION Protection;
  TYPE
    Hidden += { { . , << >> } } RECORD [x,y : CARDINAL ];
    Open { { . , << >> } };
    Extend += { { := } }RECORD [z : REAL ];

  PROCEDURE P : (a : CHAR) +=···
END Protection.
```

Figure 3.15: Zuse type protection examples. OK behaves like a Mesa *read-only variable*. Procedure P may only be called from module M. All importers may read and update Extend.a, and read Extend.b, but only the exporting module may assign to Extend.b. Note that the exporting module may extend *its own* privileges to an exported item in the implementation unit.

3.6.3 The Semantics of Type Protection

The static rules that govern the use of type protection in Zuse are considerably more complex than those for abstract and semi-abstract types, and we will therefore examine these rules using a slighty more formal notation than

previously used in this chapter.

To ensure that a program is *access-correct* [110], i.e. that it does not illegally access protected data, one has to make sure that a type derived from a protected type does not obtain privileges not held by the base type. The following set of rules determines the legality of a declaration of a type M`T with the protection clause T_{prot}:

1. Every operator in T_{prot} must be applicable to T. For example, if (N):{ + } $\in \mathsf{T}_{prot}$, then T must be an arithmetic type.

2. If T is a composite type with a component of type C, and if (N):{ := } $\in \mathsf{T}_{prot}$, then it is required that (N):{ := } $\in \mathsf{C}_{prot}$. I.e., if a composite type is to have assignment rights then so must all of its components. The same rule applies to other operators which affect the entire type, i.e. =:, = $\cdots$ <=, and << >>. The declaration of Protection`Wrong in Figure 1.15, for example, is illegal since it gives the module Q assignment rights although Wrong.x does not have (Q) :{ := }.

3. If T is an equivalence, subrange, or unique type with base type B, then T_{prot} must be a subset of B_{prot}. Or, formally, (N):{ *op* } $\in \mathsf{T}_{prot} \Rightarrow$(N):{ *op* } $\in \mathsf{B}_{prot}$.

4. If M`T is an object type with supertype S, then it is required that (M):{ WITH } $\in \mathsf{S}_{prot}$.

It is furthermore essential to establish rules to determine the legality (with respect to protection) of every expression and statement in a program. We list the most essential such rules:

1. 'd := e' is legal (with respect to protection) in module M if

   (a) (M):{ := } $\in$ TYPE(d)$_{prot}$, and
   (b) (M):{ =: } $\in$ TYPE(e)$_{prot}$, and
   (c) TYPE(d)$_{prot} \subseteq$ TYPE(e)$_{prot}$.

2. The expression 'a **op** b' is legal in module M if

   (a) (M):{ =:, **op** } $\subseteq$ TYPE(a)$_{prot}$, and
   (b) (M):{ =:, **op** } $\subseteq$ TYPE(b)$_{prot}$.

   The protection clause of the expression *a* **op** *b* is the intersection of the protection clauses of the subexpressions, i.e. TYPE(a **op** b)$_{prot}$ is TYPE(a)$_{prot} \cap$ TYPE(b)$_{prot}$.

3. The procedure call '`N`P` $(e_1, \ldots, e_n)$' is legal in module `M` if

   (a) `N`P` is declared in module `N` as `P` $(m_1 f_1, \ldots, m_n f_n)$ where $f_i$ is the type of the $i$:th formal parameter and $m_i$ is its transfer mode (`REF` or `VAL`), and

   (b) (`M`):{ =:,( ) } $\subseteq$ TYPE(`N`P`)$_{prot}$, and

   (c) m_i = `REF` and the assignment '$e_i := f_i$' is legal in `N` with respect to protection, and

   (d) The assignment '$f_i := e_i$' is legal in `M` with respect to protection.

   The last two rules guarantee that `N`P` does not perform operations on data objects passed to it that the calling module would not have been allowed to perform.

The last rule states that a module should have write access (:=) to variables which it passes by reference. This requirement is lifted if the called procedure is declared in the same module as the variable's type. This allows one, for example, to declare types for which the predefined assignment operator is not defined, and assignment has to be done through a procedure call. Similar rules guide the use of Ada's *private* and *limited private* types.

3.7 Evaluation and Examples

Clarity, orthogonality, and expressiveness are some of the criteria which are often used to evaluate a programming language. These evaluations are often based on an analysis of the ease and elegance with which the language can express common data types and data structures such as complex numbers, stacks, and trees, and algorithms such as sorting and searching. Important though they may be, programming artifacts such as these represent a very limited sample of the wide spectrum of programming idioms found in typical complex programs. The real strengths and weaknesses of a language can be revealed only through long-term use by a large user base in a wide variety of applications.

None of the few languages with a flexible encapsulation scheme similar to Zuse have been in widespread use for any length of time: Oberon-1 – which has extensible records similar to Zuse – has been superseded by Oberon-2, which does not separate module specifications and implementations; the SRC Modula-3 version 2.0 compiler which supports extensible object types has, at

the time of writing, just been released.[4] At this point, therefore, there exists little actual empirical evidence of the utility of flexible encapsulation. Hence, to evaluate the Zuse type system we are left with the option of giving small and often contrived examples. This section will present a number of such examples and discuss the pros and cons of the Zuse solution in the light of solutions given in other languages.

3.7.1 Complex Numbers – Abstract Structured Types

Figure 1.16 shows three Zuse variants of a *Complex Number*-abstraction, a favorite data abstraction example. The first variety represents complex numbers with a Zuse abstract type, the second uses a hidden pointer type, and the third a protected concrete type. The Create, Add, and Incr operations are implemented as abstract inline procedures since they can be expected to be simple and frequently called. Procedures which do input or output, on the other hand, are unlikely to benefit from being expanded inline, and we can therefore declare the Print operation NOT INLINE.

Semantics of Assignment

The *complex number*-abstraction may appear trivial, but it does reveal some subtle semantic differences between the three approaches to data abstraction. The first complication we will consider is the difference between *reference*- and *value* semantics. In the cases when Complex`T is realized as a record (as in the first and last realizations of T in Figure 1.16), an assignment $a := b$ where $a$ and $b$ are of type Complex`T will have the effect that the value of a is overwritten with the value of b. In other words, complex variables will behave similarly to variables of the built-in numeric types REAL and INTEGER which also display value semantics. If, on the other hand, Complex`T is realized as a pointer to a record (the second realization of T in Figure 1.16), the effect of $a := b$ will be to make a and b refer to the same location. This *sharing* of values has the side effect that any future change to the variable a (using the operation Incr, for example) will affect b, and vice versa.

Distinguishing between the cases when a type exhibits reference- or value semantics is never a problem in Ada or Modula-2. A Modula-2 programmer

[4] As we have seen in Chapter 2 Oberon-1's extensible records are more restricted than Zuse's, since they require that the maximum size of the type be revealed in the specification unit. The SRC Modula-3 compiler resorts to run-time lookup of the size of parent types, in order to support extensible object types. The Zuse translation systems to be presented have no extra run-time cost associated with extensible types.

```
SPECIFICATION Complex;
  (* Zuse abstract style: *)
  TYPE       T        =   { *:{ :=, =: } };
  CONSTANT Zero : T =;

  (* Modula-2 style: *)
  TYPE       T        =    REF{ *:{ :=, =: } };
  CONSTANT Zero : T =;

  (* Ada style: *)
  TYPE       T        == { *:{ :=, =: } } RECORD [Re, Im : REAL ];
  CONSTANT Zero : T == RECORD [Re :== 0.0; Im :== 0.0];

  (* Operations: *)
  PROCEDURE Create : (Re, Im : REAL) RETURN T =;
  PROCEDURE Add : (a, b : T) RETURN T =;
  PROCEDURE Incr : (REF a : T; b : T) =;
  NOT INLINE PROCEDURE Print : (a : T) =;
END Complex.

IMPLEMENTATION Complex;
  (* Zuse abstract style: *)
  TYPE       T        += { { .  } } RECORD [Re, Im : REAL ];
  CONSTANT Zero : T += RECORD [Re :== 0.0; Im :== 0.0];

  (* Modula-2 style: *)
  TYPE       T        += { {  ↑ } } REF [RECORD [Re, Im : REAL ]];
  CONSTANT Zero : T += REF [RECORD [Re :== 0.0; Im :== 0.0]];

  (* Ada style: *)
  TYPE       T { { .  } };

  (* Operations: *)
  INLINE PROCEDURE Create : (Re, Im : REAL) RETURN T +=···
  INLINE PROCEDURE Add : (a, b : T) RETURN T +=···
  INLINE PROCEDURE Incr : (REF a : T; b : T) +=···
  PROCEDURE Print : (a, b : T) RETURN T +=···
END Complex.
```

Figure 3.16: Zuse complex arithmetic example.

knows that opaque types always have reference semantics, and an Ada programmer can determine from the private section of a package specification whether a private type is realized as a pointer or not. A Zuse programmer, on the other hand, cannot determine whether an abstract type will behave as a reference or a value merely by inspecting the exporting module's specification (see Figure 1.17). One way to handle this problem is to disallow assignment on the abstract type and to provide clients with a special assignment procedure with well-specified semantics (Figure 1.18). Another possibility is to overload the meaning of the assignment operator; however overloading is currently not a part of the Zuse language.

```
SPECIFICATION Complex;
  TYPE T = ;
END Complex.
```

```
IMPLEMENTATION Complex;
  (* Value semantics: *)
  TYPE T += RECORD [Re, Im : REAL ];

  (* Reference semantics: *)
  TYPE T += REF [RECORD [Re, Im : REAL ]];
END Complex.
```

Figure 3.17: Value or reference semantics in Zuse.

```
SPECIFICATION Complex;
  TYPE T = { *:{ =: } };
  PROCEDURE ValueAssign : (
     REF a : T;
     b     : T) =;
END Complex.
```

```
IMPLEMENTATION Complex;
  TYPE T += { { :=,.  } }RECORD [Re, Im : REAL ];
  INLINE PROCEDURE ValueAssign : (
     REF a : T; b : T) +=
  BEGIN a := b;  END ValueAssign;
     — or —
  TYPE T += { { := } } REF [RECORD [Re, Im : REAL ]];
  INLINE PROCEDURE ValueAssign : (
     REF a : T; b : T) +=
  BEGIN a ↑ := b ↑;  END ValueAssign;
END Complex.
```

Figure 3.18: A complex number package with value semantics.

Reclaiming Temporary Storage

All modern imperative languages support static as well as dynamic allocation, but the language design often displays a preference for one or the other. Ada, for

example, was designed primarily with static allocation in mind (this is evident from the way the representation of private types must be revealed in the package specification) and is undetermined on whether inaccessible storage should be reclaimed automatically or manually (implementations are allowed but not required to support garbage collection). Modula-3, on the other hand, relies primarily on dynamic allocation (object types and opaque types are always pointers) and assumes automatic storage management. Modula-2 also favors dynamic allocation (opaque types are always pointers) but relies on manual disposal of unused dynamic storage.

These different approaches to storage management lead to different run-time behavior for programs which use variables of an abstract type. Consider the following program fragment:

```
VARIABLE a, b, c, d : Complex`T;
BEGIN
.....
d := Complex`Add(a, Complex`Add (b, c));
.....
```

In Ada Complex`T would be a statically allocated private type (a *protected concrete type* in Zuse parlance. See the third variant in Figure 1.16.), and temporary storage for the result of the addition of b and c would be stored on the run-time stack and automatically reclaimed when the scope is exited. In Modula-3 Complex`T would be a traced opaque reference type, and the temporary result of Complex`Add (b, c) would be reclaimed by the garbage collector. In Modula-2 Complex`T would be an opaque type, and since Modula-2 does not have garbage collection, the memory allocated for the temporary complex variable would never be reclaimed.[5]

Abstraction vs. Allocation

We have seen from the above examples how the Zuse abstract type makes it possible to combine the abstractness of Modula-2 and Modula-3's opaque types with the static allocation of Ada's private type. In the light of the previous discussion we can now summarize the advantages of static allocation, type abstraction, and the combination of the two:

enhanced readability Abstract types do not clutter the specification unit with information irrelevant to a user of the module.

[5] See Feldman [72, pp. 127-131] for a thorough discussion of this problem in Modula-2.

efficient compilation Changes to the realization of an abstract type will not require any client of the exporting module to be recompiled.

implementation freedom Abstract types allow the implementer complete freedom in choosing the most appropriate data type for their realization.

client independence Since abstract types do not reveal their realization, clients are unlikely to depend on it for their correctness. Changes to the realization are therefore unlikely to affect clients.

value semantics An abstract type with a static realization will automatically display value semantics.

simple storage management Static allocation does not require any complex run-time storage management. Even in languages which support automatic reclamation of dynamic memory, it is prudent to use static allocation whenever possible in order not to unduly tax the garbage collector. Furthermore, in languages such as Modula-2 which only support manual reclamation of storage, there is no risk of some storage remaining un-reclaimed.

efficient allocation Static storage needed by a subprogram is created automatically on the stack when the subprogram is invoked and reclaimed automatically when the call returns. This is in contrast to the *dynamic* storage needed by the subprogram which has to be created through explicit calls to the run-time system. Initializing a $10^3 \times 10^3$ complex matrix (see Figure 1.19), for example, will require 10^6 calls to the run-time system allocation routine if Complex`T is a dynamic type.

efficient code Every reference to a variable of a dynamic abstract type will have to go through one more level of indirection than if the type had been static.

From the above list one might construe that value semantics and static allocation ought to be the preferred choice for all abstract types. This is, of course, not the case. Frequently, the desirable behavior of assignment on variables of an abstract type is to copy the *identity* of the source to the destination, rather than to make a copy of the *value* of the source. In these cases the natural realization of the abstract type is as a pointer to some structure.

3.7.2 Search Trees – Extensible Object Types

Snyder [198], in discussing encapsulation in object-oriented languages, makes two observations:

```
PROGRAM P;
  IMPORT Complex;
  TYPE Index == RANGE INTEGER [1 .. 1000];
  VARIABLE
     a   : ARRAY  [Index, Index]  [Complex`T];
     i,j  : INTEGER;
BEGIN
  FOR i IN [1..1000] DO
     FOR j IN [1..1000] DO
        a [i,j] := Complex`Create(1, 0);
     ENDFOR;
  ENDFOR;
END P.
```

Figure 3.19: Initializing a large complex matrix. If Complex`T is realized as a pointer to a record, each call to Complex`Create will imply a call to the dynamic memory allocation routine.

- A class may have two kinds of clients: those which create and operate on objects of the class (*instantiating* clients), and those which inherit from the class (*inheriting* clients). The two kinds have different visibility requirements: the former usually only manipulate the state of an object through method-calls, whereas the latter often need to access the private fields of their superclass directly.

- There are two applications of inheritance: one class may inherit from another for the sole purpose of reusing its code (*implementation* inheritance), or in order to declare that its behavior is a refinement of the behavior of its parent (*specification* inheritance or *specialization*). Since specification inheritance is a public announcement about the behavior of a class, any change to a class' specification inheritance is likely to affect its descendants. Implementation inheritance, on the other hand, is private to the class, and neither instantiating nor inheriting clients should be affected by a change.

In most object-oriented languages it is possible to make a distinction between instantiating and inheriting clients. In C++, for example, *protected* class members can only be accessed by the methods of the defining class and its descendants, and in Modula-3 different module interfaces can reveal different amounts of information to instantiating and inheriting clients. As a rule, however, languages do not distinguish between implementation and specification

inheritance, and specifically do not provide a way to hide the parent–child relationship. Modula-3 is an exception to this rule since it is possible to hide that ST is the parent of T by letting T inherit from ROOT in the module interface (see Nelson [160, pp. 143–153]):

```
INTERFACE M;
 TYPE
   Private <: ROOT ;
   T       =  Private OBJECT ··· END ;
END M.
```

```
MODULE M;
 REVEAL
   Private = ST OBJECT ··· END ;
END M.
```

```
SPECIFICATION BST;
 TYPE
   Result == ENUM [Less, Equal, Greater];
   Node   == OBJECT
               [left,right : Node  :== NIL ]
               [compare : (n : Node) RETURN Result] ;
   T      == OBJECT
               [root     : Node ]
               [insert   : (n : Node)                       :== Insert ;
                member   : (n : Node) RETURN BOOLEAN :== Member] ;

 PROCEDURE Insert : (t : T; n : Node) =;
 PROCEDURE Member : (t : T; n : Node) =;
END BST.
```

Figure 3.20: Base class for binary search trees. Derived classes will normally use the original member method but override the insert method. The implementation unit which implements Member and Insert has been omitted.

Zuse currently does not have a way to reveal different amounts of information to inheriting and instantiating clients.[6] The Zuse extensible object type, however, makes it possible to distinguish between specification and implementation inheritance, since the supertype of an exported object type can either be revealed in the specification unit or hidden in the implementation. Figure 1.21 shows two different balanced binary search tree implementations inheriting from the base class BST (Figure 1.20). The definitions of RedBlack`T and AVL`T in Figure 1.21 are examples of specification inheritance, since the object types reveal their position in the type hierarchy. Figure 1.22, on the

[6]We are, however, considering extending the current one-to-one module system with a more general one, which would allow different views of the same object to different types of clients.

other hand, is an example of implementation inheritance, since the supertype of IntSet`T is not visible to its clients. Figure 1.21 also illustrates how Zuse allows instance variables and methods to be hidden or revealed.

```
SPECIFICATION RedBlack;
  IMPORT BST;
  TYPE Node == WITH BST`Node OBJECT [Red : BOOLEAN] [ ];
       T    == WITH BST`T OBJECT [ ] [ ]  :== OBJECT [ ] [insert :== Insert];

  PROCEDURE Insert : (t : T; n : Node) =;
END RedBlack.

IMPLEMENTATION RedBlack;
  PROCEDURE Insert : (t : T; n : Node) += BEGIN ··· END Insert;
END RedBlack.


SPECIFICATION AVL;
  IMPORT BST;
  TYPE Node = WITH BST`Node OBJECT [ ] [ ];
       T    = WITH BST`T OBJECT [ ] [ ];
END AVL.

IMPLEMENTATION AVL;
  IMPORT BST;
  TYPE Balance == RANGE INTEGER [-1 .. 1];
       Node    += WITH BST`Node OBJECT [Height : Balance] [ ];
       T       += WITH BST`T OBJECT [ ] [ ]  :== OBJECT [ ] [insert :== Insert];

  PROCEDURE Insert : (t : T; n : Node) == BEGIN ··· END Insert;
END AVL.
```

Figure 3.21: Two balanced search tree implementations.

3.7.3 Graphs – Semi-hidden Types

Although Zuse makes it possible to hide all representational details of every exported item, there are sometimes compelling arguments for making some aspects of some items visible. We have already seen how an object type often needs access to the instance variables of its supertype, but there can be other reasons as well:

translation-time efficiency The Zuse translation systems proposed in this thesis (see Chapters 5 through 7) are based on the idea that inter-modular

```
SPECIFICATION IntSet;
  TYPE T = OBJECT
               [ ]
               [put : (i : INTEGER) ;
                in  : (i : INTEGER) RETURN BOOLEAN] ;
END IntSet.

IMPLEMENTATION IntSet;
  IMPORT AVL;
  TYPE Node == WITH AVL`Node OBJECT [value : INTEGER] [ ]
                   :== OBJECT [ ] [compare :== CompareInt];
       T    += WITH AVL`T OBJECT [ ] [ ]
                   :== OBJECT [ ] [ put :== PutInt; in :== InInt];
END IntSet.
```

Figure 3.22: Example of implementation inheritance.

information (i.e. information pertaining to the realization of abstract items) should be exchanged at module binding-time. A problem with this solution is that the cost of binding can be prohibitively high for programs which contain many abstract items. Revealing some information about the realization of exported items may reduce this cost.

run-time efficiency Permitting clients of a module to access the representation of an exported type may sometimes allow the application of more efficient algorithms.

The latter of these points is illustrated in the graph package of Figure 1.23, where the specification reveals either that nodes are enumerable or that they are a subrange of the type SHORTCARD. This revelation makes it possible for the depth-first-search routine in Figure 1.23 to allocate a vector of booleans (one per node) to mark the nodes which have been visited.

3.7.4 Lines in the Plane – Extensible Types and Abstract Initialization

We have already seen examples of how extensible object types can reveal some of their fields and methods, and keep some hidden. We will next give a further example of the usefulness of extensible types, this time statically allocated extensible record types. At the same time we will show how extensible types can benefit from abstract initializations.

```
SPECIFICATION Graph;
  TYPE
     T    =  REF;
     Edge = ;
     Node =  < > ;
     — or —
     Node =  RANGE SHORTCARD ;

  PROCEDURE Create : () RETURN T =;
  PROCEDURE NewNode : (G : T) RETURN Node =;
  PROCEDURE NewEdge : (G : T; f,t : Node) RETURN Edge =;
END Graph.

IMPLEMENTATION Graph;
  CONSTANT Max : SHORTCARD == ·····;
  TYPE Node += RANGE SHORTCARD [1 .. Max];

  TYPE
     Edge += RECORD [f,t : Node ];
     T    += REF [ARRAY  [Node, Node]  [BOOLEAN]];
     — or —
  TYPE
     Edge += REF [RECORD [to : Node; next : Edge ]];
     T    += REF [ARRAY  [Node]  [Edge]];
END Graph.

PROGRAM UseGraph;
  IMPORT Graph;
  PROCEDURE Dfs : (G : Graph`T; N : Graph`Node) ==
  BEGIN Visited [N] := TRUE; ····· END Dfs;

  VARIABLE
     G       : Graph`T;
     N       : Graph`Node;
     Visited : ARRAY  [Graph`Node]  [BOOLEAN];
BEGIN
  FOR N IN Graph`Node DO Visited [N] := FALSE;  ENDFOR;
  FOR N IN Graph`Node DO
     IF ¬ Visited [N] THEN Dfs (G, N); ENDIF;
  ENDFOR;
END UseGraph.
```

Figure 3.23: A graph package and a depth-first-search routine using the package.

Figure 1.24 shows the module Line, which implements operations on lines in the plane. The only operation shown is Length, which computes the distance between two points. If Length is called frequently for the same line, it may be efficient to compute the distance only once and to store the result between calls. For this purpose Line`T has an extra hidden field len which is given its value the first time Line`Length is called. T`Length is given the initial value -1.0 to distinguish between a line for which the length has been computed, and one for which it has not. This scheme, of course, only works under the assumption that the From and To fields are not changed between calls to Line`Length.

```
SPECIFICATION Line;
  IMPORT Point;
  TYPE T = RECORD [
               From : Point`T;
               To   : Point`T ]  := ;

  PROCEDURE Length : (
     REF I : T) RETURN REAL =;
END Line.
```

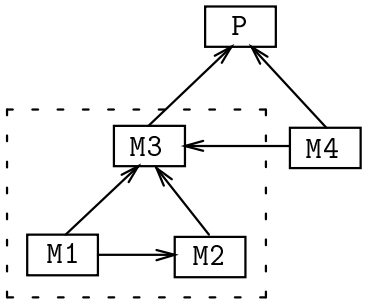
```
IMPLEMENTATION Line;
  IMPORT Point;
  TYPE T += RECORD [len : REAL := -1.0 ];

  PROCEDURE Length : (
     REF I : T) RETURN REAL +=
  BEGIN
     IF I.len < 0.0 THEN I.len := ···;  ENDIF;
     RETURN I.len;
  END Length;
END Line.
```

Figure 3.24: A module implementing lines in the plane.

3.7.5 Protection as a Structuring Tool

Most common modular languages treat programs as structure-less clusters of separately compiled modules. Although modules are often logically organized in a hierarchical fashion, there usually does not exist a way to reflect this organization in the program text. The result is that it is possible for any module to invoke the operations exported by any other module. The PIC/Ada [229] *provide* and *request* clauses represent one way of restricting the visibility among the modules in a program. We will show how the Zuse protection clause can be used in a similar fashion. Figure 1.25 (a) shows the import-export relationship in a program consisting of four separate modules and a main program. Modules M1 and M2 are logically submodules of M3, and their exported items should not be accessible from the main program module P or from M4. Figure 1.25 (b) shows the corresponding Zuse program where each exported procedure has been coupled with a protection clause to allow each module to access only the appropriate modules.



(a)

```
SPECIFICATION M1;
  PROCEDURE Q : { (M2,M3) :{ ( ) } }() =;
END M1.

SPECIFICATION M2;
  PROCEDURE Q : { (M3) :{ ( ) } }() =;
END M2.

SPECIFICATION M3;
  PROCEDURE Q : { (P) :{ ( ) } }() =;
END M3.

SPECIFICATION M4;
  PROCEDURE Q : { (M3,P) :{ ( ) } }() =;
END M4.
```

(b)

Figure 3.25: Use of protection clauses to indicate the logical structure of a set of modules.

When the visibility of all exported items in a module is affected in the same way, one might consider assigning a protection clause to the module itself, as exemplified in Figure 1.26. The clause attached to module M1, for example, would say that only modules M2 and M3 may access any of M1's exported items. This is currently not part of the Zuse language, since Zuse only allows types to carry protection clauses, and Zuse modules are type-less.

3.8 Summary

We have informally presented the flexible encapsulation primitives of the language Zuse and given some examples of their use. Zuse supports statically allocated abstract types; semi-hidden pointer, subrange, object, and enumerable types; extensible record, enumeration, object, and subrange types; abstract constants and extensible structured constants; as well as abstract and semi-abstract inline procedures. Furthermore, any concrete, abstract, or semi-abstract item may be coupled with a protection clause restricting the operations which a client module may perform on the item. Protection clauses can also be used to provide a hierarchical structure for an otherwise flat collection of modules.

We have seen how the use of abstract items instead of concrete ones has several advantages. The most important advantage is that a client of an ab-

```
SPECIFICATION M1{ (M2,M3) :{ ` } };
  PROCEDURE Q : () =;
END M1.

SPECIFICATION M2{ (M3) :{ ` } };
  PROCEDURE Q : () =;
END M2.

SPECIFICATION M3{ (P) :{ ` } };
  PROCEDURE Q : () =;
END M3.

SPECIFICATION M4{ (M3,P) :{ ` } };
  PROCEDURE Q : () =;
END M4.
```

Figure 3.26: Assigning protection clauses to modules.

stract module is rid of extraneous information which can obscure the intent of the module. In the same vein, an implementer of an abstract module is given complete freedom to use whichever data structures and algorithms he wishes, without being restricted by realization particulars already fixed by the module's specification. A more pragmatic benefit is that changes to abstract modules are less likely to incur trickle-down recompilations than changes to the corresponding concrete ones, since a greater part of the module is confined to the implementation unit.

Chapter 5

Sequential High-Level Module Binding

Es ist auf die Entwicklung von Compilern bekanntlich sehr viel Mühe vervandt worden. Vielleicht wäre es unter diesen Umständen doch richtiger gewesen, die Möglichkeiten des PK in dieser Richtung voll auszunutzen.

Konrad Zuse [239]

paste */peist/ vi* stick with paste.

pasting */peistiŋ/ n* (colloq) severe beating: *Get/Give sb a pasting.*

Oxford Advanced Learner's Dictionary of Current English

5.1 Introduction

The previous chapters have been devoted mainly to three tasks: tracing the history of abstraction and modularity, motivating the need for flexible encapsulation, and designing a modular language with a flexible encapsulation scheme. The remaining chapters will deal exclusively with the implementation aspects of this language. In the present chapter we will describe the implementation of a sequential *paster* (high-level module binder) which supports ZUSE's encapsulation primitives, while the two subsequent chapters (Chapters 6 and 7) will describe more sophisticated implementations. Chapter 8 will evaluate the different designs with regard to their efficiency.

This chapter is organized in four parts: The first part (Section 5.2) outlines the overall design of our translation system for the Zuse language, the second part (comprising Sections 5.3 through 5.6) examines low-level design decisions, the third part (Sections 5.7 through 5.9) describes the two programs which make up our translation system and possible enhancements to these programs, and the fourth part (Sections 5.10 and 5.11), finally, describes alternative approaches to the design of translating systems for Zuse-like languages.

5.2 Design Overview

The Zuse translating system (zts) which we are about to describe consists of two programs: a module compiler (zc) and a module binder (zp).[1] In this respect zts does not differ from most other modular language processors. The main differences instead lie in the way modules are bound together: rather than binding modules together exclusively at the machine code level, as is usually the case, zp is designed to bind modules together at the intermediate code as well as at the machine code level. Furthermore, Zp is able to detect and report inter-modular static error conditions such as those discussed in Section 4.2 and perform various inter-modular optimizations.

Figure 5.1 shows the actions of zts for a small program consisting of a main program module P and two modules M and N, after a change to M's specification unit. Note that zp, unlike the pre-linkers employed by many translating systems, performs the entire module binding process itself, without a call to the system link-editor.

5.2.1 The Compiler

The Zuse compiler is not very different from compilers for other similar modular languages. As can be seen from Figure 5.1, zc compiles module specification units into *symbol files* and implementation units into *object code files*. An essential point is that Zuse does not allow dependencies between implementation units, and consequently the Zuse compiler has access only to the information present in the specification units of imported modules. This is evident from Figure 5.1, where it is shown that the input to the compiler when processing a particular module unit – in addition to the source code of the unit itself – consists solely of the symbol files of imported modules.

The most significant difference between zc and other compilers may be the way zc handles translation-time expressions.[2] There are many compile-time

[1] The particular instance of zp discussed in this chapter is called the sPaster.

[2] Translation-time expressions will also be called *compile-time* or *constant* expressions.

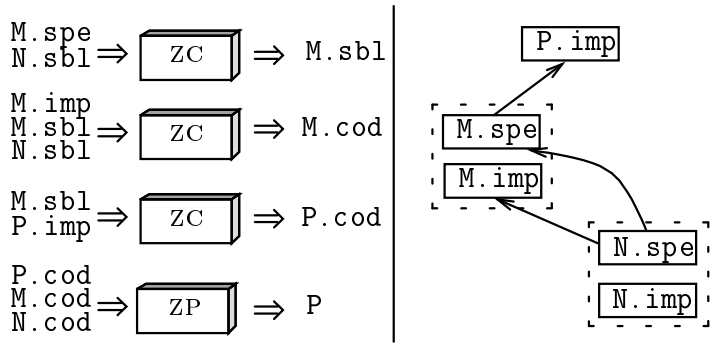


Figure 5.1: Translating actions for the program P after a change to M's specification unit. The double arrows indicate the files ZC and ZP read and produce at each stage. The suffixes `.spe` and `.imp` are used for the source code files of specification units and implementation units, respectively. Similarly, the `.sbl` suffix is used for the symbol files (the result of compiling specification units), and `.cod` is used for the resulting object code files. The executable program file has no suffix.

sources of constant expressions: sizes of types are constant expressions, as are the values of manifest constants, offsets of record fields, offsets of formal parameters in procedure activation records, addresses of variables and procedures, method templates of object types, etc. While most modular language compilers are always able to evaluate a constant expression, for a Zuse compiler this often will not be the case. Rather, since the compiler has access only to the information available in the symbol files of imported modules, the value of a translation-time expression associated with an imported abstract item will be *unknown* to the compiler. For example, when compiling a unit which declares a variable of an imported abstract type, the size of the variable (a constant expression) will be unknown and the compiler will be unable to perform the allocation of the variable.

Another consequence of the Zuse encapsulation scheme is that it is sometimes impossible for the Zuse compiler to generate (optimal) machine code for sections of source code which contain references to imported abstract items. Consider, as a simple example, a procedure which performs an assignment of two variables of an imported abstract type. The compiler – not knowing the

They are characterized by the fact that they are evaluated *prior* to the execution of the program, and that their values are retained until the program is changed.

size of the variables – will be unable to select the best instruction for the assignment: if the variables are of one of the standard sizes of the target architecture (a byte or a word, say), then the appropriate simple move-instruction will be sufficient. If, on the other hand, the variables are large, a more expensive block-move instruction will be necessary. Similarly, when processing a block of code which calls an imported abstract procedure, the compiler will not know whether the called procedure is inline or not, and if it is, the code of the procedure will not be available for expansion.

The Zuse compiler handles these and all similar cases in the same manner: if the compiler cannot perform a particular action because of inadequate inter-modular information, the action is deferred to module binding-time when all the necessary information is available. In the example above, where a procedure performs an assignment on variables of an abstract type, the compiler would leave the procedure in its intermediate code form and pass it on to the module binder for final code generation.

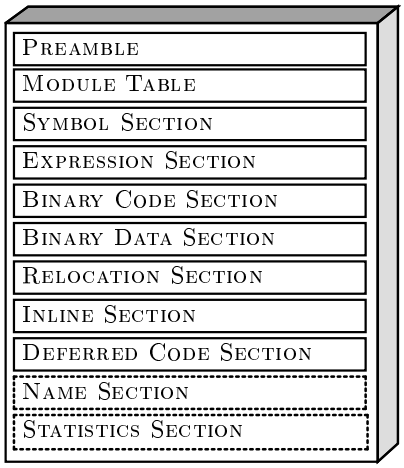
The central data structure employed by the zts is the *Constant Expression Table* (CET) which stores all inter-modular information. The CET for a module contains entries for every abstract and semi-abstract exported item, as well as symbolic constant expressions generated during the compilation process. An expression in a module's CET can refer to the values of other expressions in the same CET or in the CET of imported modules. CET expressions are often arithmetic (computing, for example, the size of a type or the offset of a record field), but can be of a variety of other types (boolean, string, list, etc.) as well. The CET furthermore stores deferred context conditions such as those discussed in Section 4.2, as well as call-graphs of procedures.

5.2.2 The Binder

The paster zp is zts's module binder. Starting with the program module, its job is to find all the modules which should be part of the final program, combine them, and then produce the resulting executable program. It should be evident from the previous discussion that in addition to the actions usually performed by module binders, zp also has to perform some tasks traditionally reserved for compilers. In particular, zp has to complete all the actions left undone by zc: deferred constant expressions have to be evaluated, code has to be generated for deferred procedures, deferred context conditions have to be checked and error messages issued if violations are detected. In addition to this, zp is designed to perform various inter-modular optimizations, such as inline expansion and dead-code elimination.

5.3 Intermediate File Formats

The ZTS symbol and object code files share the same format. An intermediate file for a module M consists of a preamble followed by seven major and two supplementary sections: the Module Table is a list of the names of the modules imported by M, the Symbol Section holds a symbolic representation of the items declared by M,[3] the Expression Section is a set of unevaluated constant expressions, the Binary Code and Data Sections contain code generated during compilation, the Relocation Section contains relocation information for the binary code, the Inline Section contains the intermediate code of inline procedures exported by M, and the Deferred Code Section contains the intermediate code of deferred procedures.



Two supplementary sections – the Name and Statistics Sections – do not take active any part in the translation process. The Name Section, which contains the source code names of the items defined in M, is only used during debugging and to produce binding-time error messages. The Statistics Section contains information used to produce source statistics (see Section 8.3.3). The object code file Preamble contains file pointers to the start of each section, as well as each section's *logical size*. Logical size is interpreted differently for different sections: for the Expression Section it is the number of expressions in the module, for the Inline and Deferred Code Sections it is the number of intermediate code instructions, for the Binary Code Section it is the number of bytes of code, etc.

[3] This section is generally empty for object code files.

5.4 The Constant Expression Table

The *Expression Section* holds the CET, which is the central ZTS data structure. The expressions organize all the inter-modular information needed during pasting: they express among other things sizes of types, static error conditions, the allocation of global variables, and the inline-status and call-graphs of procedures. The general structure of an expression is

$$e_{m,i}.\mathrm{a} \quad \triangleq \quad f(e_{m_1,i_1}.\mathrm{a}_1, \cdots, e_{m_n,i_n}.\mathrm{a}_\mathrm{n})$$

where $e_{m,i}.\mathrm{a}$ is the attribute a of the i:th expression in module m, f is a function with n parameters, and the $e_{m_k,i_k}.\mathrm{a}_\mathrm{k}$:s are references to attributes of other expressions. The function f may be an arithmetic operation, an operation to construct structured constants or method templates from their subparts, etc. In most cases, an expression will be associated with a particular exported abstract or semi-abstract item, and each attribute will express one of the item's hidden aspects. Seen this way, the CET serves as the global symbol table for abstract and semi-abstract items.

The simple example in Figure 5.2 shows how the CET generated from a module's specification unit contains a slot (an expression with an unknown right hand side) for every hidden aspect of each exported item. These slots are filled in when the corresponding implementation unit is compiled.

5.4.1 CET Operators

The CET operators and operand types reflect the needs of the translating system as well as the basic type system of the target language. Some of the operators are given in Table 5.1 below. In addition to the operand types shown (integer [$\mathbb{I}$], unsigned integer [$\mathbb{I}^+$], real [$\mathbb{R}$], boolean [$\mathbb{B}$], set [$\mathbb{S}$], error [$\mathbb{E}$], source position [$\mathbb{P}$], and address [$\mathbb{X}$]), the CET also supports the character [$\mathbb{C}$], string [$\mathbb{G}$], struct [$\mathbb{U}$] (record or array literals), and list [$\mathbb{L}$] types. Polymorphic operators take operands of type any [$\mathbb{A}$].

5.4.2 CET Attributes

Figure 5.2 on the next page shows that different items produce different kinds of entries in the CET. Every aspect of an abstract item produces a CET entry, as do the hidden aspects of semi-abstract items. Concrete items, however, do not as a rule produce any entries at all, since everything about a concrete item is known to an importer at compile-time. Procedures are an exception: all procedures, whether abstract, semi-abstract, or concrete, are represented in the

```
SPECIFICATION M;
  TYPE T = ;

  CONSTANT C : CARDINAL =;

  PROCEDURE P : () =;
  PROCEDURE Q : () =;
END M.
```

$e_{M,0}$.calls $\triangleq$? -- M
$e_{M,1}$.size $\triangleq$? -- M`T
$e_{M,2}$.value $\triangleq$? -- M`C
$e_{M,3}$.inline $\triangleq$? -- M`P
$e_{M,3}$.calls $\triangleq$?
$e_{M,3}$.addr $\triangleq$?
$e_{M,4}$.inline $\triangleq$? -- M`Q
$e_{M,4}$.calls $\triangleq$?
$e_{M,4}$.addr $\triangleq$?

```
IMPLEMENTATION M;
  IMPORT N;
  TYPE
    T += RECORD [a : CHAR; b : N`T ];
  CONSTANT
    C : CARDINAL += N`C * 4;

  INLINE PROCEDURE P : () +=
  BEGIN ··· END P;
  PROCEDURE Q : () +=
  BEGIN N`P (); END Q;
END M.
```

$e_{M,0}$.calls $\triangleq$ [ ]
$e_{M,1}$.size $\triangleq$ $e_{M,5}$.expr + $e_{N,1}$.size
$e_{M,2}$.value $\triangleq$ $e_{N,2}$.value * $e_{M,6}$.expr
$e_{M,3}$.inline $\triangleq$ $\mathbb{T}$
$e_{M,3}$.calls $\triangleq$ [ ]
$e_{M,4}$.addr $\triangleq$ *null*
$e_{M,4}$.inline $\triangleq$ $\mathbb{F}$
$e_{M,4}$.addr $\triangleq$ $0x100$
$e_{M,4}$.calls $\triangleq$ $[e_{N,4}]$
$e_{M,5}$.expr $\triangleq$ 1
$e_{M,6}$.expr $\triangleq$ 4

Figure 5.2: The CET for a module M. Any text following "--" is a comment. Inline procedure addresses are given the empty value "null." $\mathbb{T}$ and $\mathbb{F}$ signify TRUE and FALSE. Square brackets ([ ]) hold lists of expressions.

CET with entries for their address (*addr*) and their call-graphs (*calls*). This data must be present at module binding-time in order to facilitate relocation and dead-code elimination. Table 5.2 summarizes the more important attributes.

In addition to the above-mentioned attributes, each expression has a boolean attribute *global*, defined such that $e_{m,i}$.global $\triangleq \mathbb{T}$ if the value of any of $e_{m,i}$'s attributes may be used in a module other than m. Intuitively, $e_{m,i}$.global $\triangleq \mathbb{T}$ if $e_{m,i}$ represents an exported item, but the definition is complicated by the fact that the body of an exported inline procedure may refer to an item which itself is not exported. The *global*-attribute is given its value at compile-time according to the following rules:

1. $e_{m,i}$.global $\triangleq \mathbb{T}$ if $e_{m,i}$ is associated with an exported item and does not represent an intermediate quantity. For example, if the size of a type is represented by a complicated expression with several subexpressions, only

OPERATOR	SIGNATURE	COMMENTS
$a+b$, $a*b$, $\texttt{max}(a,b)$, $\cdots$	$\mathbb{R} \times \mathbb{R} \to \mathbb{R}$	Real arithmetic.
$a+b$, $a*b$, $\texttt{max}(a,b)$, $\cdots$	$\mathbb{I} \times \mathbb{I} \to \mathbb{I}$	Integer arithmetic.
$a=b$, $a \leq b$, $\cdots$	$\mathbb{A} \times \mathbb{A} \to \mathbb{A}$	Bitwise comparisons.
$a \vee b$, $a \wedge b$, $\neg a$	$\mathbb{B}(\times\mathbb{B}) \to \mathbb{B}$	Logical operations.
$a \cap b$, $a \cup b$, $\cdots$	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	Set operations.
$\|\|a,b\|\|$	$\mathbb{I} \times \mathbb{I} \to \mathbb{I}$	b-a+1.
$\texttt{if}(e,v_1,v_2)$	$\mathbb{B} \times \mathbb{A} \times \mathbb{A} \to \mathbb{A}$	**if** (e) **then** v_1 **else** v_2.
$\texttt{insert}(a,b,o,s)$	$\mathbb{A} \times \mathbb{A} \times \mathbb{I} \times \mathbb{I} \to \mathbb{A}$	Replace bytes $[o, o+s-1]$ of a with bytes $[0, s-1]$ of b.
$\texttt{extract}(a,s,o)$	$\mathbb{A} \times \mathbb{I} \times \mathbb{I} \to \mathbb{A}$	Extract bytes $[o, o+s-1]$ from a.
$\texttt{allocate}(a)$	$\mathbb{I}^+ \to \mathbb{A}$	Allocate constant value of size a.
$\texttt{variable}(a)$	$\mathbb{I}^+ \to \mathbb{X}$	Allocate global variable of size a.
$\texttt{check}(e,m,p)$	$\mathbb{B} \times \mathbb{E} \times \mathbb{P} \to \mathbb{B}$	**if** $(\neg e)$ **then** issue error message m at source position p.

Table 5.1: Selection of CET operators.

ATTRIBUTE	TYPE	KIND	ITEMS
addr	$\mathbb{X}$	syn	modules, procedures, variables.
calls	$\mathbb{L}$	syn	modules, procedures.
ref	$\mathbb{B}$	inh	modules, procedures.
size	$\mathbb{I}^+$	syn	abstract and semi-abstract types.
low_bound	$\mathbb{I}$	syn	hidden enumerable types, extensible subrange types.
high_bound	$\mathbb{I}$	syn	hidden enumerable types, extensible subrange and enumeration types.
width	$\mathbb{I}^+$	syn	hidden enumerable types, extensible subrange and enumeration types.
template_size	$\mathbb{I}^+$	syn	extensible object types.
template	$\mathbb{A}$	syn	extensible object types.
default	$\mathbb{A}$	syn	types with an abstract or semi-abstract default value.
has_default	$\mathbb{B}$	syn	types with an abstract or semi-abstract default value.
inline	$\mathbb{B}$	syn	abstract procedures.
value	$\mathbb{A}$	syn	abstract and semi-abstract constants.
expr	$\mathbb{A}$	syn	literal values and intermediate expressions.

Table 5.2: A selection of CET attributes, their type, kind (synthesized or inherited), and the items to which they apply.

the top-level expression will be *global*.

2. $e_{m,i}$.global $\triangleq \mathbb{T}$ if there exists an inline procedure P exported by m such that the body of P (or the body of any inline procedure declared in m and called by P) refers to $e_{m,i}$.

3. Otherwise, $e_{m,i}$.global $\triangleq \mathbb{F}$.

We will leave any further discussion of the *global*-attribute to Section 6.5 of the next chapter, where its use will become evident.

Borrowing from the terminology of the theory of *attribute grammars* (to which ZTS expressions bear an obvious resemblance), we notice that all the attributes discussed so far have been of the *synthesized* kind. An expression attribute is said to be synthesized if its value is defined solely in terms of the values of its subexpressions. For example, since the size of a structured type is a function of the sizes of its component types, the attribute *size* must be synthesized. One attribute is an exception: the boolean *ref*-attribute, which is defined to be $\mathbb{T}$ if the procedure with which it is associated might be called at run-time. Since a procedure can be called only if any of the procedures which call it may be called, *ref* is an *inherited*-attribute. At compile-time, only the expressions $e_{m,i}$ associated with module bodies have $e_{m,i}$.ref $\triangleq \mathbb{T}$.

5.4.3 CET Examples

It is sometimes convenient to think of the CET as a graph where each $e_{m,i}$.a is a node labeled with the operation f with n outgoing edges to the nodes $e_{m_1,i_1}.\mathrm{a}_1, \cdots, e_{m_n,i_n}.\mathrm{a}_\mathrm{n}$. In most cases the graph will be a forest of expression DAGs, but the following example shows that the graph may sometimes contain cycles:

$$
\begin{array}{lcll}
e_{M,0}.\text{calls} & \triangleq & [\,] & -- \ \mathsf{M} \\
e_{M,1}.\text{size} & \triangleq & e_{N,1}.\text{size} & -- \ \mathsf{M`T1} \\
e_{M,2}.\text{value} & \triangleq & e_{N,2}.\text{value} + e_{M,4}.\text{expr} & -- \ \mathsf{M`C1} \\
e_{M,3}.\text{inline} & \triangleq & \mathbb{T} & -- \ \mathsf{M`P} \\
e_{M,3}.\text{calls} & \triangleq & [e_{N,3}] & \\
e_{M,4}.\text{expr} & \triangleq & 4 & \\
 & & & \\
e_{N,0}.\text{calls} & \triangleq & [\,] & -- \ \mathsf{N} \\
e_{N,1}.\text{size} & \triangleq & e_{M,1}.\text{size} + e_{M,1}.\text{size} & -- \ \mathsf{N`T1} \\
e_{N,2}.\text{value} & \triangleq & e_{M,2}.\text{value} * e_{N,4}.\text{expr} & -- \ \mathsf{N`C1} \\
e_{N,3}.\text{inline} & \triangleq & \mathbb{T} & -- \ \mathsf{N`P} \\
e_{N,3}.\text{calls} & \triangleq & [e_{M,3}] & \\
e_{M,4}.\text{expr} & \triangleq & 5 &
\end{array}
$$

The example corresponds to the CETs resulting from modules M and N in Figure 4.1 (page 84), in which the declarations of T2, T3, and C2 have been removed. The illegal declarations of M and N are evident from the concatenation of the two CETs: the recursive definitions of M`T1 and N`T1 correspond to a cycle involving the types' *size*-attributes, and the recursive inline calls can be discerned from the call-graph of the procedures M`P and N`P.

Other types of deferred static error checks can also be expressed in the CET, which may be seen from the following CET for the program in Section 4.2.4 (page 86):

$$
\begin{array}{lcll}
e_{M,0}.\text{calls} & \triangleq & [\,] & -- \ \mathsf{M} \\
e_{M,1}.\text{size} & \triangleq & 8 & -- \ \mathsf{M`T} \\
 & & & \\
e_{P,0}.\text{calls} & \triangleq & [\,] & -- \ \mathsf{P} \\
e_{P,1}.\text{addr} & \triangleq & \texttt{variable}(e_{M,1}.\text{size}) & -- \ \mathsf{V2} \\
e_{P,2}.\text{expr} & \triangleq & e_{M,1}.\text{size} = e_{P,6}.\text{expr} & \\
e_{P,3}.\text{expr} & \triangleq & \texttt{"Illegal type coercion"} & \\
e_{P,4}.\text{expr} & \triangleq & \texttt{"Module P, line 7"} & \\
e_{P,5}.\text{expr} & \triangleq & \texttt{check}(e_{P,2}.\text{expr}, e_{P,3}.\text{expr}, e_{P,4}.\text{expr}) & \\
e_{P,6}.\text{expr} & \triangleq & 2 &
\end{array}
$$

Since the two types involved in the coercion do not have the same size, the test in expression $e_{P,2}$ will fail, $e_{P,2}.\text{expr} \triangleq \mathbb{F}$, and a side effect of evaluating $e_{P,5}$ will be to issue the error-message.

Our final example will be the construction of method templates for object types. Zts implements an object variable as a pointer to a record containing the type's instance variables together with a pointer to the type's method template (see also Sections 2.7.4 and 2.8.9). This template is a constant record containing, among other things, the addresses of the type's methods. The method template for an object type T with a supertype S is constructed by concatenating S's template with the addresses of any additional methods introduced by T, and replacing those of S's methods which were overridden by T. In the case when S is imported and not concrete, T's template cannot be built at compile-time. An example of this situation is given in Figure 5.3.

5.5 The Intermediate Form

There are several potentially time-consuming operations that have to be performed during module binding, all involving the program in its intermediate form. Therefore, great care has to be taken in the design of the intermediate

```
SPECIFICATION M;                 e_{M,1}.size          ≜  ?   --  M`T
  TYPE T = OBJECT;               e_{M,1}.template_size ≜  ?
END M.                           e_{M,1}.template      ≜  ?
IMPLEMENTATION M;
  TYPE T += OBJECT [f : CHAR] [P : (x : CHAR) :== P1] ;
  PROCEDURE P1 : (x : CHAR) == BEGIN ··· END P1;
END M.
e_{M,1}.size          ≜  1          --  M`T
e_{M,1}.template_size ≜  4
e_{M,1}.template      ≜  {0x100}
e_{M,2}.addr          ≜  0x100      --  M`P1
```

```
PROGRAM P;
  IMPORT M;
  TYPE T == WITH M`T OBJECT [x : CHAR] [Q : (x : CHAR) :== Q1] ;
  PROCEDURE Q1 : (x : CHAR) == BEGIN ··· END Q1;
END P.
e_{P,1}.size          ≜  e_{M,1}.size + e_{P,3}.expr                  --  P`T
e_{P,1}.template_size ≜  e_{M,1}.template_size + e_{P,4}.expr
e_{P,1}.template      ≜  insert(e_{M,1}.template, e_{P,2}.addr,
                                 e_{P,4}.expr, e_{P,4}.expr)
e_{P,2}.addr          ≜  0x500                                        --  P`Q1
e_{P,3}.expr          ≜  1
e_{P,4}.expr          ≜  4
```

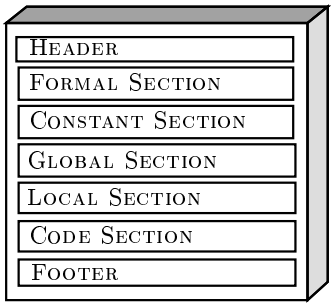
Figure 5.3: Construction of object type templates. Irrelevant expressions have been omitted. The size of P`T's instance variable record ($e_{P,1}$.size) will evaluate to 2, and its template ($e_{P,1}$.template) will be $\{0x100, 0x500\}$.

representation. The code must be *compact* so that it may be quickly loaded from the object code file, *low-level* in order for the code generation algorithm to be simple and efficient, and yet *high-level* in the sense of enough program structure being retained for the effective application of inline expansion of subprograms and other code optimization techniques. It is obvious that the resulting representation must be a compromise between these conflicting criteria.

Zts uses an *attributed linear code*[4] which resembles somewhat the intermediate codes described by Ganapathi [79] and Kornerup [129]. A Z-code *block* contains the intermediate code for a procedure, function, or module body, and

[4] henceforth, Z-code

consists of a header, a footer, and five major sections:



The Header contains a reference to the CET entry for the block; The Formal Section lists the block's formal parameters; The Constant and Global Sections are lists of the constant values and global variables referenced in the block; The Local Section declares local variables; The Code Section contains the actual intermediate code instructions which represent the body of the corresponding procedure, and the Footer ends the block.

The Formal, Constant, Global, and Local Sections will be called *declarative*, since they declare each object used by the block. These declarations are essential to the application of many code generation and optimization techniques which require the unique identification of all referenced objects, such as formal parameters, global and local variables, constants, called procedures, and compiler generated temporaries. Including these declarative sections within Z-code blocks furthermore makes each block a self-contained unit whose only non-local references are to CET expressions. This will be convenient, as we will see from the next chapter, since Z-code blocks will sometimes be transferred between the sites of a distributed system.

Each section within a block contains zero or more numbered Z-code instructions: 4-tuples made up of an attributed operator followed by three attributed arguments. Arguments are typically labels (references to other instructions), literal values, addresses, or references to CET expressions. Attributes typically convey type and size information or information which aids in the code generation and optimization phases.

5.5.1 Z-code Attributes

The table at the top of the next page lists the attributes applicable to Z-code instructions. The *Mode*-attribute indicates the transfer mode of formal and actual parameters. The *Modified*-attribute is $\mathbb{T}$ when a block modifies a particular formal parameter.

ATTRIBUTE	DEFINITION
Block	One of *Proc/Func/Module/Prog*.
Type	One of the standard types ($\mathbb{I}/\mathbb{I}^+/\mathbb{B}/\mathbb{R}/\mathbb{C}/\mathbb{G}/\mathbb{X}/\mathbb{S}/\mathbb{U}$) of Section 5.4.1.
Size	A literal integer value, a standard size (*Byte/Short/Long*), or a CET reference of type $\mathbb{I}^+$.
Mode	One of *VAL/REF*.
Modified	A literal boolean value ($\mathbb{T}$ or $\mathbb{F}$).

The current implementation only defines three argument attributes:

ATTRIBUTE	DEFINITION
Live	A literal boolean value ($\mathbb{T}$ or $\mathbb{F}$).
Use	A literal boolean value ($\mathbb{T}$ or $\mathbb{F}$).
Ind	A literal boolean value ($\mathbb{T}$ or $\mathbb{F}$).

The *Use*- and *Live*-attributes indicate whether the argument has a next use (will be referenced later on in the same basic block) or is alive (the current value will be used later on). The *Ind*-attribute (normally shown as a '$*$' if $\mathbb{T}$) indicates that the argument is an indirect reference to the actual value.

5.5.2 Z-code Instructions

Instruction arguments can be of four types:

ARGUMENT	DEFINITION
Label	A Z-code tuple number.
Value	A literal constant such as 1 or `"Hello!"`, or a CET reference.
Address	A relocatable address or a CET reference of type $\mathbb{X}$.
CET	A CET expression.

Some of the Z-code instructions can be found in Tables 5.4 and 5.5. Table 5.4 contains the *declarative* instructions of Z-code, i.e. the instructions which introduce new data items. Table 5.5 shows the *imperative* instructions found in the Code Section.

5.5.3 Z-code Examples and Discussion

We give one short Z-code example in Figure 5.6, which mainly shows how unknown quantities are referenced through the CET.

It is natural at this point to question the extent to which the chosen representation fulfills the criteria set up at the beginning of the section, or whether some other intermediate form would have served the purpose better. It should be clear that Z-code sacrifices compactness for ease of code generation and optimizations such as inline expansion. Languages such as postfix notation,

INSTRUCTION	ATTRIBUTES	ARGUMENTS	SECTION
BEGIN	*Block*	a:CET (c, g, l):*Label*	Header
Beginning of block a. The arguments c, g, and l point to the first instructions in the Constant, Global, and Local Sections, respectively.			
FORMAL	*Mode*/*Size*/*Type*/*Modified*	$u : \mathbb{I}^+$	Formal
A formal parameter. u is the formal's *usage-count*.			
OPEN	*Mode*/*Size*/*Type*/*Modified*	h:*Label*; $u : \mathbb{I}^+$	Formal
An open array formal parameter. *Size* and *Type* refer to the array element. h references the formal parameter holding the high bound of the array.			
CONST	*Size*/*Type*	v:*Value*; $u : \mathbb{I}^+$	Constant
A constant value.			
GLOBAL	*Size*/*Type*	a:*Address*; $u : \mathbb{I}^+$	Global
Reference to a global variable.			
LOCAL	*Size*/*Type*	$u : \mathbb{I}^+$	Local
A local variable.			
TEMP	*Size*/*Type*	$u : \mathbb{I}^+$	Code
Compiler generated temporary.			
END	*Block*		Footer
End of a block.			

Figure 5.4: Z-code declarative instructions and the sections in which they occur. Most declarative instructions have a *usage-count* argument u, computed in line with the Dragon book [6, pp. 542-543]

in which the relationships between intermediate code instructions are implicit rather than explicit, are evidently more compact than 4-tuple notations. Furthermore, Z-code wastes space by including type and size information in declarative as well as imperative instructions, and by including explicit declarations for each compiler generated temporary variable. This redundant information, while slowing down the reading and writing of intermediate code blocks, simplifies code generation.

In regard to the level of representation, the next section will show that Z-code includes features which make it well suited for inline expansion. While Z-code's small instruction set and severely restricted addressing modes (only the ASSIGN- and INIT-operators support indirect referencing) make it suitable for RISC-like target architectures, code generators which want to take advantage of complex addressing modes will have to reconstruct lost context. The next-use and usage-count information aid code generation, as does the *Modified*-attribute of formal parameters.[5] The reason that this type of information is included

[5]The current implementation passes large (larger than a machine word) formal value parameters by reference, and it is the callee's responsibility to make a local copy upon entry. If

INSTRUCTION	ATTRIBUTES	ARGUMENTS
ASSIGN	*Size/Type*	*a/Ind/Live/Use*; *b/Ind/Live/Use*
$a \leftarrow b$. $a/* \leftarrow b$ means assignment through a (an address).		
INIT	*Size/Type*	*a/Ind/Live/Use*; *b/Live/Use*; *h:Value*
If $h = \mathbb{T}$ then $a \leftarrow b$. b will always be a constant value.		
LOAD		*a/Live/Use*; *b/Live/Use*;
a is assigned the address of the variable referenced by b.		
ACTUAL	*Mode/Size/Type*	*e/Live/Use*; *l*
CALL		*p*:CET; *f*
CALLIND		*r/Live/Use*; *f*
A call to procedure p or an indirect call to the procedure referenced by variable r. f points to the first **ACTUAL** of the call, l to the previous one. e is the (value or address of) the actual argument.		
ADD	*Size/Type*	*a/Live/Use*; *b/Live/Use*; *c/Live/Use*;
$a \leftarrow b + c$. *Type* is $\mathbb{I}$, $\mathbb{I}^+$, or $\mathbb{R}$.		
JUMPLT	*Size/Type*	*a/Live/Use*; *b/Live/Use*; *L*
JUMP		*L*
Conditional jump (**JUMPLT**) to L if $a < b$, and unconditional jump (**JUMP**) to L.		
RANGE		(*v1 v2*):*Value*; *L*
CASE	*Size/Type*	*e/Live/Use*; *f*; *d*
The **RANGE**-instruction is a jump-table entry for the values $[v1 \ldots v2]$ causing a jump to L. The **CASE**-instruction jumps to the label associated with the value of the expression e. f points to the first **RANGE** instruction, and d to the default (**ELSE**) instruction.		

Figure 5.5: A subset of Z-code's imperative instructions. Arguments are of type *Label* unless otherwise specified.

directly in the intermediate code is to make each Z-code block self-contained.

There are certainly a host of other intermediate code forms which could be employed instead of Z-code, and which might facilitate either optimization or code generation, or which would improve portability. Holt's [98, 59] *Data Descriptors*, for example, are likely to be good building blocks for simple and portable code generators, but are less likely to be suitable for extensive optimization. The *Program Dependence Graph* [74] and the *Static Single Assignment Form* [61, 219] are two interesting recent developments designed to support aggressive optimization techniques. It is not clear at this point, however, whether code generation and optimization from these representations can be made efficient enough for our purposes. On the other hand, while representations which are closer to the target architecture, such as *Register Transfer*

the callee can determine that it does not modify the formal (if *Modified* $= \mathbb{F}$), no copy need be made.

```
SPECIFICATION M;
  TYPE T = := ;
  CONSTANT C : CARDINAL =;
  VARIABLE  V : T;
  VARIABLE  X : CARDINAL;
  PROCEDURE P : (f : T) =;
END M.
```

$e_{M,1}.size \triangleq ?$
$e_{M,1}.default \triangleq ?$
$e_{M,1}.has\_default \triangleq ?$
$e_{M,2}.value \triangleq ?$
$e_{M,3}.addr \triangleq$ `variable`$(e_{M,1}$.size$)$
$e_{M,4}.inline \triangleq ?$
$e_{M,4}.calls \triangleq ?$
$e_{M,4}.addr \triangleq ?$

```
PROGRAM N;
  IMPORT M;

  PROCEDURE P : (
    f : M`T) ==
  VARIABLE
    r : CARDINAL;
    x : M`T;
  BEGIN
    f := M`V;
    r := M`C + M`X;
    M`P (x);
  END P;
END N.
```

#	INSTRUCTION	ARGUMENTS		
1	**BEGIN**/*Proc*	$e_{N,1}$ 3 5 7		– P
2	**FORMAL**/*VAL*/$e_{M,1}$.size/$\mathbb{A}$/$\mathbb{T}$		#1	– f
3	**CONST**/*Long*/$\mathbb{I}^+$	$e_{M,2}$.value	#1	– M`C
4	**CONST**/$e_{M,1}$.size/$\mathbb{A}$	$e_{M,1}$.default	#1	– M`T
5	**GLOBAL**/$e_{M,1}$.size/$\mathbb{A}$	$e_{M,3}$.addr	#1	– M`V
6	**GLOBAL**/*Long*/$\mathbb{I}^+$	[M, 0]	#1	– M`X
7	**LOCAL**/*Long*/$\mathbb{I}^+$		#1	– r
8	**LOCAL**/$e_{M,1}$.size/$\mathbb{A}$		#2	– x
9	**INIT**/$e_{M,1}$.size/$\mathbb{A}$	8/$\mathbb{T}$/$\mathbb{T}$	4/$\mathbb{F}$/$\mathbb{F}$	$e_{M,1}$.has_default
10	**ASSIGN**/$e_{M,1}$.size/$\mathbb{A}$	2/$\mathbb{F}$/$\mathbb{F}$	5/$\mathbb{F}$/$\mathbb{F}$	
11	**TEMP**/*Long*/$\mathbb{I}^+$	#2		
12	**ADD**/*Long*/$\mathbb{I}^+$	11/$\mathbb{T}$/$\mathbb{T}$	3/$\mathbb{F}$/$\mathbb{F}$	6/$\mathbb{F}$/$\mathbb{F}$
13	**ASSIGN**/*Long*/$\mathbb{I}^+$	7/$\mathbb{F}$/$\mathbb{F}$	11/$\mathbb{F}$/$\mathbb{F}$	
14	**ACTUAL**/$e_{M,1}$.size/$\mathbb{A}$	8/$\mathbb{F}$/$\mathbb{F}$	0	
15	**CALL**	$e_{M,4}$ 14		
16	**END**/*Proc*			

Figure 5.6: A Z-code translation of the procedure *P*. "[M, 0]" is the relocatable address of M`X, computed by adding 0 to the base address of M's data area. Usage-counts are prefixed by a #-sign.

Lists (RTL) [21, 63, 202], might improve the speed of code generation, the lack of type and size information[6] for instructions which manipulate abstract items will make their application difficult. It would be difficult, for example, to give an RTL translation of procedure P's first assignment in Figure 5.6, since it is not known at compile-time whether the arguments are small enough to fit in registers or whether they are larger structures which need to be manipulated through address registers. Furthermore, low-level representations such as RTL are less suited to high-level optimizations such as inline expansion. RTL inlining in the GNU C compiler [202], for example, requires 1900 lines of code, whereas the ZTS inline expansion module is only 400 lines long.

[6] At compile-time, when the intermediate code for deferred procedures is generated.

5.6 Inline Expansion

The merits of procedure inlining[7] have for a long time been the subject of much debate. Several issues have been at stake:

- Should inlining be user defined or should it be the responsibility of the translating system to determine the routine calls which should be inlined?
- How can a translating system determine when inlining will be profitable?
- Which optimizations are likely to benefit from the larger scope opened up by procedure inlining?
- Should optimizations be performed before or after inlining?
- How does the target architecture affect the benefits of inlining?
- How can inlining be performed efficiently?

Mesa, Ada, and C++, as well as dialects of many other languages (e.g. Maclaren's [143] Pascal dialect), require the programmer to specify the routines which should be subjected to integration. This is sometimes referred to as *user-directed inlining*. Cuse goes one step further by promoting inlining to an abstraction primitive (see Section 3.3.3).

Translating systems for languages without explicit inlining directives have sometimes chosen the other route, which is to heuristically determine the routines which will profit from integration.[8] The advantage of this method is that it treats inlining as just another optimization technique whose application is transparent to the programmer. Determining the routines and call-sites which will profit from integration has turned out to be a difficult problem and several heuristic methods have been devised: Ball [18] used data flow analysis to estimate how inlining calls with constant actual arguments affects optimization; Scheifler [188] used a heuristic which inlined small routines, routines with just one call-site, and routines which (according to collected profiling information) were called often; McFarling [146] considered the effect of inlining on instruction cache performance, also in the light of profiling data.

[7] Inline procedures are also known as *open* procedures, and inline expansion is sometimes referred to as procedure *integration*, *substitution*, or *merging*.

[8] The MIPS compiler suite [94] supports both inlining directives and an internal heuristics.

Several studies have investigated the benefits of inlining, and it has sometimes been questioned whether the improvements in execution-time are outweighed by increased translation time. Scheifler [188] reports execution-time savings from 5 to 28% (unoptimized vs. unoptimized integrated code), and Richardson [177] 15 to 23% (optimized vs. optimized integrated code), whereas Himelstein [94] notes that the degradation in cache performance outweighs the benefits of inlining to the point that the MIPS compiler has turned off inlining altogether. Furthermore, Cooper's [55] study of FORTRAN compilers' ability to capitalize on inlining showed no consistent improvements. Holler [97] reports that the Sun-3 C compiler ran on average a factor 3.78 slower on programs subjected to her source-to-source C inliner, while Richardson [177] reports that their inliner integrated in an optimizing compiler slowed down compilation by a factor of 4.8.

Additional questions are where in the translation process inlining should be performed and how inlining across module boundaries should be handled. Holler [97] inlines at the source code level and considers all source modules referenced by a program in order to extract the necessary function definitions. Himelstein [94] concatenates the intermediate code files of each module prior to integration. Sale [183] suggests link-time intermediate code integration, much as is done in this thesis. Languages such as C++ and Mesa require the body of inline routines to appear in module interfaces in order to be considered for cross-module inlining.

5.6.1 Inlining in ZTS

Both the ZTS compiler and module binder are equipped to perform inline expansions. ZC performs all intra-modular expansions and all inter-modular expansions of concrete inline routines but defers inter-modular expansions of abstract and semi-abstract inline routines to ZP. We distinguish between two kinds of expansions: *inline-in-inline* expansions (integration of calls to inline routines in inline routines) and expansions of inline in non-inline routines. Both ZC and ZP perform the two kinds in separate phases. Our reasons for separating the two kinds of inlining will be made evident in the next chapter. In order to minimize the number of expansions, the inline-in-inline integrations are performed in reverse topological order (i.e. bottom-up in the inline call-graph),[9] and the remaining expansions are performed in a subsequent phase. The ZTS inliner rejects directly as well as indirectly recursive inline calls; i.e. the inline

[9]This technique seems to have been independently discovered by several authors. See, for example, Hall [88, pp. 35, 103], Holler [97, p. 28], and Hwu [149, section 3.3]. Allen [7, p. 711] shows, however, that the quality of the resulting code can suffer from the use of this technique.

call-graph is assumed to be acyclic.

We will give a brief description of the ZTS inliner to illustrate how the Z-code design makes for very simple and efficient inline expansion. In Algorithm 5.1 below *caller* and *callee* are the Z-code tuple numbers of the **BEGIN**-instructions of the caller and callee, respectively, and *call-site* is the tuple-number of the **CALL**-instruction which is to be substituted.

Algorithm 5.1 (Expand Inline Call)

```
PROCEDURE Expand (caller, callee, call-site);
    M ← MergeSection (caller.c, callee.c);
    M ← M ∪ MergeSection (caller.g, callee.g);
    M ← M ∪ MapFormals (call-site.f, callee + 1);
    M ← M ∪ CopyCode (call-site, callee.l);
    Update (call-site.f, M);
END Expand;
```

Expand starts by merging the Constant and Global Sections of the callee into the corresponding section in the caller. If the sections are sorted this can be done in time linear in the total number of tuples in the sections. As part of the merging process a map M from the callee's old Z-code tuple numbers to its new numbers in the caller is constructed. *MapFormals* maps the callee's formal parameters to the actuals used in the call. In some cases this will involve the copying of an actual into a fresh local variable, and this is also performed by *MapFormals*. The copying is necessary for pass-by-value actuals whose corresponding formal parameter is modified by the callee. This is discernible from the formal's *Modified*-attribute. *CopyCode* copies the callee's Local and Code Sections into the call-site and removes the **ACTUAL** and **CALL**-instructions. Finally, the inserted instructions are updated from the tuple-map M. The entire process runs in time proportional to the number of tuples in the callee plus the number of tuples in the caller's Constant and Global Sections.

5.7 The Compiler

Zuse-type encapsulation puts a heavy burden on the module binder, since the lack of compile-time inter-modular information forces the compiler to defer many of its tasks to binding-time. The design of ZC has therefore been guided by the general principle that – even when encapsulation restricts the compiler's access to inter-modular information – as much translation work as possible should be performed at compile-time.

With some exceptions, ZC resembles compilers for traditional modular languages, and traditional techniques can be employed for most of ZC's translation tasks. The present implementation runs in the six phases shown below:

Algorithm 5.2 (Compilation)

1 : Syntactic Phase
2 : Evaluation Phase
3 : Semantic Phase
4 : Intermediate Code Phase
5 : Machine Code Phase
6 : Output Phase

The phases are much the same whether a specification or implementation unit is compiled, except that the compilation of an implementation includes an initial phase which loads all sections of the corresponding symbol file, and that the Output Phase generates object code files from implementation units and symbol files for specification units. We next give a brief description of each of the major phases.

5.7.1 Syntactic Phase

Since Zuse, like Modula-2, allows procedure and variable identifiers to be used prior to their declaration, syntactic and semantic analysis are performed in distinct phases. The Syntactic Phase builds the symbol table and the abstract syntax tree and loads the symbol files of imported modules:

Algorithm 5.3 (Syntactic Phase)

1. Load the Symbol and Inline Sections from the symbol files of imported modules.[10] In the current implementation we use the **B**β approach to symbol file processing (see Section 2.7.1).

2. Build symbol table entries for the declared items.

3. If we are compiling a specification unit then create CET entries for every hidden aspect of every declared abstract or semi-abstract item.

4. Build the abstract syntax tree for declared procedures.

[10] The present implementation does not support concrete procedures, and hence the Inline Section will always be empty.

The processing of declarations and the building of the symbol table will call for the evaluation of many constant expressions. These stem from such things as manifest constants, the calculation of sizes of types, offsets of record fields, etc. Because of Zuse's flexible encapsulation scheme virtually every constant expression which arises during the compilation of a Zuse module can involve terms whose values are unknown at compile-time. When an expression cannot be evaluated due to insufficient inter-modular information, a corresponding CET entry is built instead. Similarly, context conditions (such as those in Section 4.2.2) which cannot be checked at compile-time generate the appropriate CET `check`-expressions.

To reduce the amount of work that has to be done at binding-time we employ three techniques to minimize the number of CET expressions:

- common subexpression elimination
- algebraic simplifications
- declaration reordering

The elimination of common subexpressions turns the CET into a graph (usually a DAG) rather than a tree. There are mainly two cases when the reordering of declarations pays off: record fields and global variables. Consider, for example, the following declaration ($e_{F,4}$.size represents the size of F`T, $e_{F,2}$.expr the offset of F`T.c, and $e_{F,3}$.expr the offset of F`T.d):[11]

```
PROGRAM F;
  IMPORT M,N;
  TYPE T == UNSAFE RECORD [
              a : M`T;
              b : INTEGER;
              c : N`T;
              d : INTEGER ];
END F.
```

$$
\begin{aligned}
e_{F,1}.\text{expr} &\triangleq 4 \\
e_{F,2}.\text{expr} &\triangleq e_{M,1}.\text{size} + e_{F,1}.\text{expr} \\
e_{F,3}.\text{expr} &\triangleq e_{F,2}.\text{expr} + e_{N,1}.\text{size} \\
e_{F,4}.\text{size} &\triangleq e_{F,3}.\text{expr} + e_{F,1}.\text{expr}
\end{aligned}
$$

If instead the record is reorganized so that fields of known size precede those of unknown size, we are able to save one expression:

```
PROGRAM F;
  IMPORT M,N;
  TYPE T == RECORD [
              b : INTEGER;
              d : INTEGER;
              a : M`T;
              c : N`T ];
END F.
```

$$
\begin{aligned}
e_{F,1}.\text{expr} &\triangleq 8 \\
e_{F,2}.\text{expr} &\triangleq e_{F,1}.\text{expr} + e_{M,1}.\text{size} \\
e_{F,3}.\text{size} &\triangleq e_{F,2}.\text{expr} + e_{N,1}.\text{size}
\end{aligned}
$$

[11] The Zuse **UNSAFE** keyword is used to prevent reorderings of the kind presented here. This facility is useful mainly when interfacing with other languages.

Furthermore, in contrast to the first case when only the offset of a is known at compile-time, the reordering allows us to calculate the offsets of b and d. This may save work at module binding-time since procedures which only reference the a, b, and d fields of T need not be deferred.

5.7.2 The Evaluation and Semantic Phases

The Evaluation Phase (which need only be performed during the compilation of implementation units) is necessary since some of the constant expressions generated during the compilation of the specification may become evaluable in the light of the information in the implementation unit:

Algorithm 5.4 (Evaluation Phase)

1. Evaluate as many CET expressions as possible.
2. Update the symbol table with the newly computed values.

In addition to traversing and decorating the abstract syntax tree produced during the Syntactic Phase, the Semantic Phase performs the following actions:

Algorithm 5.5 (Semantic Phase)

1. Check for static semantic correctness.
2. Build the CET call-graph.
3. Create the appropriate CET `check`-expressions for the static checks which cannot be performed due to insufficient inter-modular information (such as those in Sections 4.2.3 and 4.2.4).
4. Set the *global*-attribute of all CET expressions according to the rules in Section 5.4.2.

We should note here that, in spite of what has been said, not all deferred constant expressions and deferred static tests need to generate explicit CET entries. For example, we do not generate CET expressions that calculate the activation record offsets of local variables and formal parameters, and we do not generate CET checks for the uniqueness of case statement labels (see Section 4.2.2). The reason is that the machine code generator handles both these cases more efficiently on its own without the explicit help of the CET.

5.7.3 Intermediate and Machine Code Phases

The Intermediate Code Phase produces and optimizes Z-code blocks:

Algorithm 5.6 (Intermediate Code Phase)

1. Generate a Z-code block for each procedure, function, and module body in the abstract syntax tree.
2. Expand inline calls in available inline procedures according to a bottom-up traversal of the inline call-graph.
3. Expand inline calls in non-inline procedures.
4. Optimize each Z-code block.

Before the Machine Code Phase can commence code generation, it must determine which procedures need to be deferred. Intuitively, it is necessary to defer any Z-code block containing references to the CET, but we will show that in some special cases this rule can be relaxed. We say that a Z-code block B needs to be deferred iff it contains an instruction I such that:

1. I is a **CALL**-instruction and the callee is an imported abstract procedure or an imported inline semi-abstract procedure. Note that this excludes calls to local procedures, imported concrete procedures, and imported semi-abstract non-inline procedures.
2. I is a **GLOBAL**-instruction with an unknown *size*-attribute, and B performs some operation (such as assignment or bitwise comparison) on the global variable which requires knowledge of its size. This excludes operations which involve only the global's address, such as passing it as a reference parameter, accessing one of its components (if it is a record, array, or set), and taking its address.
3. I is a **FORMAL**/*REF* or **OPEN**/*REF*-instruction with an unknown *size*-attribute, and the conditions in the previous point hold.
4. I is any other kind of instruction (other than **BEGIN** and **END**), and one of I's attributes or arguments contains a CET references.

In other words, Z-code blocks which only manipulate the *address* of abstract or semi-abstract items or the addresses of objects which depend on such items need not be deferred. The reason is, of course, that the size of an address is always known, and that it is possible to generate code for a procedure even if it

contains references to addresses which are not yet known. Filling in the correct addresses once they are known is handled by the relocation system described in the next section. We can now conclude this section by listing the tasks performed by the Machine Code Phase:

Algorithm 5.7 (Machine Code Phase)

1. Compute basic block and next-use information.
2. Determine which Z-code blocks need to be deferred.
3. Generate machine code for the non-deferred Z-code blocks.

5.8 The Sequential Paster

Having examined the design of the Zuse compiler and the data structures shared by the compiler and the sPaster (the *sequential paster*), we can now turn to the design of the sPaster itself. In some ways the sPaster resembles ordinary systems link editors; in other ways it is completely different. Both the sPaster and ordinary link editors combine the code produced by a compiler for separately compiled modules into an executable file. The sPaster (as well as most link editors) runs in several phases, the first phase loading definitions and the last phase performing relocations. However, unlike link editors the sPaster is equipped to perform code generation, inter-modular optimization, and some static semantic checking.

Our present sPaster design runs in four major phases:

- Phase 1 determines which modules will make up the resulting program, loads the expressions and inline procedures of all modules, and copies the binary code and data sections to the resulting executable file.
- Phase 2 evaluates the expressions, allocates global variables, performs deferred semantic checks, determines unreferenced procedures, and expands inline calls in inline procedures.
- Phase 3 reads the intermediate code of every referenced procedure, updates the code with the expression values calculated during Phase 2, expands inline calls, optimizes the intermediate code, generates machine code, and writes the generated code to the executable file.
- Phase 4 performs final relocation.

In addition to providing a nice conceptual framework for discussing the necessary tasks of high-level binders, we have several good reasons for dividing the sPaster's processing along these lines. First of all, the sPaster can be expected to have to deal with very large amounts of data, and it is essential to try to minimize the amount that has to be kept in primary memory at any one time. It would – for any but the smallest programs – be quite unacceptable to try to store all the information in all the object code files in primary memory.[12] Instead we open each object code file twice, each time reading only the necessary information: the first time during Phase 1 when the necessary global information is loaded (essentially the CET and the inline procedures), and the second time during Phase 3 when the deferred procedures are read. Once Phase 3 has completed code generation and optimization for the deferred procedures of a particular module, all the data which has been gathered for that module can be safely discarded. Only the information read during Phase 1 and computed during Phase 2 need remain in primary memory during the entire execution of the sPaster.

Separating expression evaluation and inline-in-inline procedure integration from the rest of the sPaster's processing has several advantages. First of all, since Phase 1 has completed the reading of the Expression Sections when Phase 2 commences processing, all CET expressions are available to it for evaluation. If instead evaluation ran concurrently with Phase 1, then the evaluation algorithm would have to be able to handle the case when a certain expression was not yet available. Another reason for separating the evaluation of expressions from the rest of the sPaster processing is that it enables static errors to be found early. If instead Phase 2 were incorporated into Phase 3 (with expression evaluation and procedure integration proceeding on an as need basis), a static error might go undetected until the very end of Phase 3. Furthermore, since our current design evaluates all expressions prior to the start of Phase 3, the code generator and optimizer do not need to know about unevaluated expressions. In fact, the sPaster's code generator is identical to the one used in the Zuse compiler. Finally, as we shall see in the next chapter, a separate evaluation phase enables efficient distributed evaluation and replication of CET expressions.

While the division of labor among Phase 1, Phase 2, and Phase 3 may seem quite natural, it is not unreasonable to question the need for a separate relocation phase (Phase 4). One might instead consider performing relocation concurrently with Phase 1 and Phase 3, updating the executable file as new addresses of procedures and other data items become available. We will examine

[12] The total size of the object code files for the largest program in the test-suite used in Chapter 8 is over 11 MB.

this approach further in Section 6.9 of the next chapter. Section 8.4.1 will discuss whether a less strict separation between the four phases would have a computational advantage.

We will now consider each phase in more detail.

5.8.1 Phase 1

Finding the modules which are going to take part in the final program means finding the closure of the modules' Module Table. This is accomplished by Algorithm 5.8 below:

Algorithm 5.8 (Phase 1)

1. Let M be the set of modules found so far, and let M initially contain the main module. Let U be a queue (with the operations *Insert*, *Delete*, and *Empty*) of hitherto unprocessed modules. U is also initialized to contain the main module. Let PC be the program counter.

2. Execute the following code:

```
PC ← 0;
M ← {(main module)};
U ← {(main module)};
Create the executable file;
REPEAT
    n ← Delete (U);
    Locate and open n's object file and load the preamble;
    Read n's Module Table L;
    FOR ALL V SUCH THAT (V ∈ L) ∧ (V ∉ M) DO
        M ← M ∪ {V};
        Put (U, V);
    END;
    Load n's Expression and Inline Sections;
    FOR EACH e_{n,i}.addr≠? DO
        Let e_{n,i}.addr ≙ e_{n,i}.addr + PC;
    END;
    Copy n's Data Section to the executable file;
    Copy n's Binary Code Section to the executable file;
    INC (PC, The size of the Data and Binary Code Sections);
    Load n's Relocation Section;
    Close n's object file;
UNTIL Empty (U);
```

The only part of this algorithm which may require some explanation is the updating of the *addr*-attributes. As explained in Sections 5.3 and 5.7, the Binary Code Section holds the machine code generated at compile-time for procedures which do not reference any imported abstract items. The compiler assigns procedures addresses relative to the start of the module and stores them in the CET's *addr*-attributes. When the Binary Code Section is copied to the executable file the procedure's addresses of course change, and this is reflected in the updating of the *addr*-attributes.

5.8.2 Phase 2

Phase 2 performs two independent tasks: expression evaluation and inline-in-inline expansion. Algorithm 5.9 describes these actions:

Algorithm 5.9 (Phase 2)

1. Evaluate all CET expressions in a depth-first traversal. Write any generated structured constant data objects (such as object type templates and literal records, strings, sets, or arrays) or allocated variables to the executable file, and update the *PC* accordingly. If any of the deferred static semantic checks have failed, or if the inline call-graph is not acyclic, issue the appropriate error messages and terminate processing.

2. Store generated relocation information.

3. Fill in calculated expression values from the CET in the Z-code of all inline procedures.

4. Expand inline calls in referenced ($e_{m,i}$.ref $\triangleq \mathbb{T}$) inline procedures according to a bottom-up traversal of the inline call-graph.

The relocation information generated during Phase 2 stems from the construction of method templates. A method template contains method addresses, but if a method is deferred its address will not be known until Phase 3. Hence, the addresses in the method templates will in some cases have to be filled in during relocation (Phase 4).

5.8.3 Phase 3

Once Phase 2 has been completed, all inter-modular information is available and Phase 3 can proceed to generate code for deferred procedures. Since Phase 2 has determined the procedures which may be referenced at run-time, we can save some translation-time and code space by ignoring all other procedures.

Algorithm 5.10 (Phase 3)

```
PROCEDURE Phase3 ();
   FOR EACH module n DO
      Reopen n's object file;
      Read n's Deferred Code Section;
      Process each referenced deferred procedure e_{n,i}:;
      FOR ALL i SUCH THAT (e_{n,i}.addr ≙ ?) ∧ (e_{n,i}.ref ≙ 𝕋) DO
         Let e_{n,i}.addr ≙ PC;
         Fill in calculated expression values from the CET;
         Expand calls to inline procedures;
         IF some calls were expanded THEN
            Recompute basic blocks and next-use information;
         END;
         Optimize;
         Generate machine code and increment PC accordingly;
         Store the generated relocation information;
         Write the code to the executable file;
      END;
      Close n's object file;
   END;
END Phase3;
```

In addition to ignoring unreferenced procedures there are a few other opportunities for improving the performance of the sPaster's Phase 3. One possible optimization is to use efficient machine and operating system dependent idioms for reading the object code files. In the current `Unix` implementation, for example, the `mmap` system call is used to map the object code files directly to the sPaster's address space. Furthermore, inline and deferred procedures are stored on the object code files in the form of the compiler's memory image of the relevant Z-code. Thus, loading the deferred code segment becomes a simple matter of setting a pointer to the beginning of the segment and letting the virtual memory system do the actual reading.[13]

In Section 5.7 we noted that the CET does not contain explicit checks for the uniqueness of case statement labels. These checks are instead performed during the sPaster's code generation phase. While this may be more efficient (detecting multiple identical labels is a trivial extension of the building of the case statement jump table), it does postpone static semantic error detection from Phase 2 to Phase3.

[13] Similar techniques are commonly used by interpreted systems such as ML and Smalltalk to allow quick loading of saved execution environments.

5.8.4 Phase 4

The last binding-time action is to update the generated code in the executable file with the procedure and data addresses which were not available when the code was written. This corresponds to the *relocation* task commonly performed by link editors. The relocation information loaded during Phase 1 and generated during Phase 2 and Phase 3 is in the form of an ordered set of *relocation items* $(PC, e_{m,i})$, where PC is a position on the executable file where $e_{m,i}$.addr should eventually be written. Phase 4 becomes the simple task of considering each $(PC, e_{m,i})$ item and updating the address at position PC on the executable file with the newly computed address $e_{m,i}$.addr.

5.8.5 Discussion

It should be evident (and this conjecture will be corroborated by the tests in Section 8.4) that Phase 3 is likely to be the most expensive of the four sPaster phases. One can imagine a worst-case scenario in which every item in a program (directly or indirectly) depends on imported abstract items, where each exported procedure is declared inline, and where each deferred procedure is referenced. Not only would the sPaster have to perform a large number of procedure integrations, but the post-integration deferred procedures would most likely be very large, resulting in very expensive optimization and code generation. While we will argue that the distributed and incremental versions of the sPaster (Chapters 6 and 7) will be able to handle this kind of situation efficiently, it should be clear that the sPaster itself would achieve very poor response-time.

It is of course also imperative that the optimization and code generation algorithms used in Phase 3 are both effective and efficient. The current implementation does not perform any of the traditional intra-procedural optimizations (except for some trivial local optimizations such as constant folding and reduction in strength), and the code generation algorithm used (the Dragon book [6, section 9.6]) is far from optimal. Future versions of ZTS will explore the possibility of using other fast, but still potent, code generators (such as Wendt [221] or Fraser [78]), and will implement traditional intra as well as inter-procedural optimizations (Wall [214, 216, 217], Callahan [38], Chow [49]). It may also be possible to speed up optimization by incrementally updating next-use and flow analysis information after the integration of a procedure (see Pollock [168]), rather than recomputing it from scratch as is done in Algorithm 5.10.

In contrast to most `Unix` compilers, the sPaster generates machine code directly (without going through the system assembler) and also produces the executable file directly (without – as in Figure 3.1.2 – going through the system

link editor). While this improves translation efficiency, it hampers portability and – since the link editor is often the instrument which combines modules written in different languages – makes interfacing to other languages difficult. Future versions of ZTS will either include a conversion program from the system object code format (`.o`-format in the case of `Unix`) to the object code format used by ZTS, or instrument ZP to directly handle object code files in different formats.

5.9 Future Work

In this section we will consider a few other translation tasks which would benefit from precise inter-modular information but which are hampered by separate compilation. The algorithms sketched in this section are not part of the current ZTS implementation, but are being considered for future versions.

5.9.1 Managing Module Time-Stamps

Most modular languages binders perform some sort of time-stamp checks to determine if the modules taking part in a bind have been compiled in the correct order. These checks can be easily incorporated into the CET evaluation phase. We give each CET module entry a string-valued ($\mathbb{G}$) attribute *time-stamp*, which holds the time of compilation of the module's specification unit. We furthermore create an entry containing the time-stamp of each imported module's specification unit as found during compilation, and a deferred check that these time-stamps correspond to the ones found during module binding. Consider, for example, a module M importing a module N, and whose CET is given below. $e_{M,0}$.timestamp holds the time of compilation of M's specification unit, and $e_{M,1}$.expr holds the time when the version of N's specification unit imported by M was compiled. When M and N are bound together this time is compared to the time-stamp found in N, and if they are not the same an error message is issued.[14]

$e_{M,0}$.timestamp	$\triangleq$	`3/23/92 11:18:01`	`--` M
$e_{M,1}$.expr	$\triangleq$	`3/23/92 10:07:11`	`--` N's ts
$e_{M,2}$.expr	$\triangleq$	$e_{M,1}$.expr $=$ $e_{N,0}$.timestamp	
$e_{M,3}$.expr	$\triangleq$	`"Wrong compilation order"`	
$e_{M,4}$.expr	$\triangleq$	`"Module M, line 2"`	
$e_{M,5}$.expr	$\triangleq$	`check`($e_{M,2}$.expr, $e_{M,3}$.expr, $e_{M,4}$.expr)	

[14] This would occur, for instance, if the module units were compiled in the order: `N.spe`, `M.spe`, `M.imp`, `N.spe`, `N.imp`.

5.9.2 Modified Formal Parameters

In Sections 5.5 and 5.6 we saw that the boolean Z-code attribute *Modified* is used during inline expansion and code generation to avoid unnecessary copying of pass-by-value formals. The procedure used today for calculating the attribute is conservative: a routine is said to modify a formal if it either assigns directly to it, takes its address, or passes it by reference to another routine.[15] A more accurate method would give each formal parameter of each exported routine a CET *modified* entry:

$$
\begin{array}{lcll}
e_{M,1}.\text{addr} & \triangleq & ? & -- \text{ Procedure P} \\
e_{M,2}.\text{modified} & \triangleq & e_{Q,5}.\text{modified} \vee e_{N,7}.\text{modified} & -- \text{ x} \\
e_{M,3}.\text{modified} & \triangleq & \mathbb{T} & -- \text{ y}
\end{array}
$$

Here procedure P has two formal parameters x and y which are modified if $e_{M,2}.\text{modified} \triangleq \mathbb{T}$ and $e_{M,3}.\text{modified} \triangleq \mathbb{T}$, respectively. The example indicates that y is modified directly by P (through an assignment, for example), and that x is passed as a reference parameter to two routines in modules Q and N, and is hence modified if these routines in turn modify their corresponding formals.

5.9.3 Efficient Subtype Tests

In object-oriented programs it is frequently necessary to determine where in the subtype-hierarchy a certain object belongs. Such run-time type tests are sometimes inserted implicitly by the compiler when it cannot determine an object's type at compile-time. Zuse and Modula-3 furthermore support a standard function ISTYPE (a, T) which returns TRUE if the object a's type is a subtype of the object type T (see also Section 4.4). In addition to the obvious algorithm for performing such tests (which runs in time proportional to the height of the inheritance tree), there exists two $\mathcal{O}(1)$ algorithms, Dietz [65] and Cohen [50]. Dietz' algorithm requires storing, for each object type, its pre and post-order number in the inheritance tree. The SRC Modula-3 compiler assigns these type-codes at logical link-time (at run-time, but before the execution of user code), which can be expensive for programs with a large number of object types.[16] Future versions of zts will employ the CET to do type-code assignment at module binding-time.

The expressions to the left in Figure 5.7 show the CET (prior to evaluation) for three modules M, N, and P. Each module exports an object type T. N`T and

[15] Holler [97, p. 23] uses the same method.

[16] The SRC Modula-3 run-time system (v1.6) requires 24 Sun-3/80 CPU seconds to process a linear inheritance tree of 100 object types.

$e_{M,1}$.pre	$\triangleq$	?	$e_{M,1}$.pre	$\triangleq$	1
$e_{M,1}$.post	$\triangleq$	?	$e_{M,1}$.post	$\triangleq$	3
$e_{M,1}$.supertype	$\triangleq$	*null*	$e_{M,1}$.supertype	$\triangleq$	*null*
$e_{M,1}$.subtypes	$\triangleq$	?	$e_{M,1}$.subtypes	$\triangleq$	$[e_{N,1}, e_{P,1}]$
$e_{N,1}$.pre	$\triangleq$	?	$e_{N,1}$.pre	$\triangleq$	2
$e_{N,1}$.post	$\triangleq$	?	$e_{N,1}$.post	$\triangleq$	1
$e_{N,1}$.supertype	$\triangleq$	$e_{M,1}$	$e_{N,1}$.supertype	$\triangleq$	$e_{M,1}$
$e_{N,1}$.subtypes	$\triangleq$	?	$e_{N,1}$.subtypes	$\triangleq$	[ ]
$e_{P,1}$.pre	$\triangleq$	?	$e_{P,1}$.pre	$\triangleq$	3
$e_{P,1}$.post	$\triangleq$	?	$e_{P,1}$.post	$\triangleq$	2
$e_{P,1}$.supertype	$\triangleq$	$e_{M,1}$	$e_{P,1}$.supertype	$\triangleq$	$e_{M,1}$
$e_{P,1}$.subtypes	$\triangleq$	?	$e_{P,1}$.subtypes	$\triangleq$	[ ]

Figure 5.7: Calculating type-codes.

P`T both inherit from M`T, which itself does not have a supertype. The CET evaluation runs in two steps: first the *supertype*-attributes are used to build the inheritance tree (the *subtypes*-attributes), and then the type-codes are assigned in a depth-first traversal of the tree. The resulting CET is shown to the right in Figure 5.7. Once the type-codes have been assigned they can be stored in the method templates. To test if the type of object S is a subtype of type T, we simply look up the type-numbers and check that $(pre(\mathsf{T}) \leq pre(\mathsf{S})) \wedge (post(\mathsf{T}) \geq post(\mathsf{S}))$.

5.9.4 Inline Iterators

To date only a few languages have incorporated explicit support for *control abstraction*; i.e. iteration over the elements of data abstractions representing collections such as *sets* or *lists*. CLU's [13, 142, 140] *iterators* and Alphard's [195, 194] *generator forms* are two early exceptions. We find the absence of such constructs in most modern programming languages regrettable – iteration is, after all, one of the most common programming activities – and intend to include CLU-style iterators in future versions of Zuse. It will then be natural – and almost to trivial – to extend zts to handle cross-module inlining of non-recursive iterators. Such a facility is not available in any translating system we are aware of, but would be a great contribution to the efficiency of many programs (particularly numerical) that rely heavily on non-recursive iteration.

5.9.5 Sharing Code Among Generic Modules

Future versions of Zuse will sport a generic module facility similar to the one found in Ada. It will then be natural to extend zts to support code sharing among instances of generic modules, as described, for example, by Bray [36] and Rosenberg [180]. We will not describe in detail all the considerations which have to be taken into account in order for code sharing to be effective, but rather give a simple example and examine one possible implementation strategy.

Consider the generic module Stack below, which has a formal type parameter E (the element type) and a formal constant parameter Depth (the maximum number of stack elements):

```
SPECIFICATION Stack[
  TYPE       E                    = ;
  CONSTANT Depth : CARDINAL =  ] ;

  TYPE T =  := ;
  PROCEDURE Push : (REF s : T; x : E) =;
END Stack.
```

In addition to expressions for its exported items, Stack also has CET expressions for its formal generic parameters:

$$
\begin{array}{lcll}
e_{Stack,1}.\text{instance\_size} & \triangleq & ? & -- \ \text{E} \\
e_{Stack,2}.\text{instance\_value} & \triangleq & ? & -- \ \text{Depth} \\
e_{Stack,3}.\text{size} & \triangleq & ? & -- \ \text{T} \\
e_{Stack,4}.\text{addr} & \triangleq & ? & -- \ \text{Push}
\end{array}
$$

Stack's implementation unit may use the formal generic parameters as if they had been imported abstract items:

```
IMPLEMENTATION Stack;
  TYPE Idx == RANGE CARDINAL [0 .. Depth];
  TYPE T  += RECORD [s : ARRAY  [Idx]  [E]; d : Idx :== 0 ];
  PROCEDURE Push : (REF s : T; x : E) +=···
END Stack.
```

The CET resulting from Stack's implementation unit does not differ significantly from similar ones seen in Section 5.4, except for the fact that there are still undefined expressions:

$$
\begin{array}{lcll}
e_{Stack,1}.\text{instance\_size} & \triangleq & ? & -- \ \text{E} \\
e_{Stack,2}.\text{instance\_value} & \triangleq & ? & -- \ \text{Depth} \\
e_{Stack,3}.\text{size} & \triangleq & e_{M,5}.\text{expr} + e_{Stack,6}.\text{expr} & -- \ \text{T} \\
e_{Stack,4}.\text{addr} & \triangleq & ? & -- \ \text{Push} \\
e_{Stack,5}.\text{expr} & \triangleq & e_{Stack,1}.\text{instance\_size} * e_{Stack,1}.\text{instance\_value} & \\
e_{Stack,6}.\text{expr} & \triangleq & 4 &
\end{array}
$$

One difference becomes evident when we examine a client of Stack and the corresponding CET:

```
PROGRAM P;
  IMPORT Stack;
  INSTANCE  CharStack == Stack [CHAR, 100];
  VARIABLE  S         : CharStack;
BEGIN CharStack`Push (S, "A"); END P.
```

As seen from P's CET below, Stack defines – in addition to its own CET – a function `instance` which takes two parameters (corresponding to its generic formals) and returns an offset into Stack's own CET.

$$
\begin{array}{lcl}
e_{P,1}.\text{instance\_size} & \triangleq & 1 \\
e_{P,2}.\text{instance\_value} & \triangleq & 100 \\
e_{P,3}.\text{size} & \triangleq & e_{Stack,(2+e_{P,5}.\text{expr})}.\text{size} \\
e_{P,4}.\text{addr} & \triangleq & e_{Stack,(3+e_{P,5}.\text{expr})}.\text{addr} \\
e_{P,5}.\text{expr} & \triangleq & Stack.\texttt{instance}(e_{P,1}.\text{instance\_size}, \\
 & & \quad e_{P,2}.\text{instance\_value})
\end{array}
$$

The intention is that each time *Stack*.`instance` is called during CET evaluation (which is once for every source-level instantiation of Stack), it is decided whether the instance should share code with one of the previous instances or if a new one should be created. In the latter case `instance` extends Stack's CET by making a copy of all its expressions, substituting the generic actuals for the formals, and returning the number of the first of the new expressions. In our example *Stack*.`instance` would first create the expressions below, and then return 7. During Phase 3 one Push routine would be generated for each Push entry in Stack's CET.

$$
\begin{array}{lcl}
e_{Stack,7}.\text{instance\_size} & \triangleq & 1 \\
e_{Stack,8}.\text{instance\_value} & \triangleq & 100 \\
e_{Stack,9}.\text{size} & \triangleq & e_{M,12}.\text{expr} + e_{Stack,13}.\text{expr} \\
e_{Stack,10}.\text{addr} & \triangleq & ? \\
e_{Stack,12}.\text{expr} & \triangleq & e_{Stack,7}.\text{instance\_size} * e_{Stack,8}.\text{instance\_value} \\
e_{Stack,13}.\text{expr} & \triangleq & 4
\end{array}
$$

The simplest definition of `instance` creates a unique instance for each distinct set of generic actuals, and allow code sharing only between instances whose actuals match exactly. A more elaborate definition might, for example, attempt to treat one or more of the generic formals as run-time parameters, which would allow more instantiations to fall in the same instance class.

5.10 Other Methods of Implementation

It should be obvious that the implementation strategy advocated in this chapter may lead to substantial link-time overhead, and one may ask whether there exist any viable alternatives. The possibilities fall naturally into four categories:

Sophisticated pasters Construct faster pasters. These would be similar to the one described in this chapter, but would use more sophisticated translation techniques.

Restricted separate compilation Since the translation problems associated with the Zuse language stem from the use of separate compilation, a translating system which does away with or restricts the separate compilation facilities may achieve faster turn-around times.

Reference semantics Use traditional compilation techniques, but have the compiler allocate variables of unknown size on the heap, rather than on the stack. Also, whenever necessary, manipulate variables by address rather than by value.

General code Use traditional compilation techniques and allocate variables on the stack, but generate size independent code for variable references. In other words, for an assignment `a := b` where the size of the arguments is unknown at compile-time, generate a block-move instruction.

The first category will be thoroughly explored in Section 5.11 and in the next two chapters. The second category can be divided into systems which retain a restricted separate compilation facility (allowing, for example, compilation dependencies between implementation units), and those which abandon separate compilation altogether (these include some of the integrated programming environments examined in Section 7.2). The third category (which is similar to the strategies used by the CLU [13], Alberich [163], and SRC Modula-3 implementations) has several shortcomings: Memory allocation has to be deferred till run-time; the sizes of types, the values of constants, and the templates of object types will have to be calculated at run-time; variable references have to go through an extra level of indirection; and the scheme is likely to heavily tax the (required) garbage collection system. Implementations which fall in the fourth category will have less need for garbage collection, but will still have to employ a more complicated run-time memory allocation scheme since activation record offsets cannot be computed at compile-time.

Allowing compilation dependencies between implementation units as suggested in the second category might at first seem to be a viable solution. Unfortunately, however, this method renders some perfectly legal Zuse programs untranslatable. Consider the following two modules:

```
SPECIFICATION M;
  TYPE
     T = ;
     U = ;
END M.

IMPLEMENTATION M;
  IMPORT N;
  TYPE
     T += INTEGER;
     U += N`T;
END M.

SPECIFICATION N;
  TYPE
     T = ;
     U = ;
END N.

IMPLEMENTATION N;
  IMPORT M;
  TYPE
     T += INTEGER;
     U += M`T;
END N.
```

In this example it is not possible to find an order of compilation between the implementation units of M and N such that the compiler always has all the relevant information available in order to determine the size of M`U and N`U. In other words, to calculate M`U's size N's implementation unit has to be compiled prior to M's implementation unit, and vice versa. Note that unlike the program in Figure 4.1, this example does not involve any illegal recursion. One way around the problem of mutually dependent implementation units is to allow several units to take part in the same compilation. Still remaining, however, is the question of how to determine when this is necessary. Situations similar to the one in the example above also occur for abstract constants and abstract inline procedures.

The last two categories will generally produce inferior code, but there may be ways to overcome this problem. The run-time code generation and optimization techniques examined by Keppel [122] and Chambers [47] might, for example, prove valuable. Such techniques would allow the compiler to generate inferior code, and then let the run-time code generator regenerate (optimal) code for those sections which turn out to be heavily executed. Unfortunately, run-time code generation is at this time still a largely uncharted area.

5.11 Hierarchical High-Level Module Binding

We will end this chapter with a discussion of *hierarchical pasting*, a technique which might prove useful for programs which contain parts which are rarely changed.

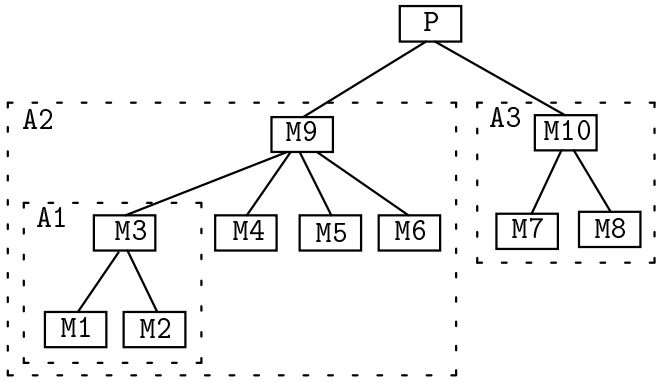


Figure 5.8: Hierarchical binding of a tree of modules. In the first step modules {M1,M2,M3} are joined to form the library A1, and {M7,M8,M10} are joined to form the library A3. The second step joins modules {M4,M5,M6,M9} and library A1, and the last step joins the main module P with libraries A2 and A3 to form the executable program.

5.11.1 Libraries of Modules

Many link-editors allow a set of object files to be bound together – not to form an executable program but rather to make a "super-object file" – which may then take part in later links. Such collections of object files are often termed *libraries*. The profits of such a scheme are twofold: collections of modules are easily distributed if joined into one library-file, and it is cheaper for the linker to process a set of modules joined into a library than to process the individual object files by themselves. The gains in linking speed stem partly because fewer files have to be read, and partly because dependencies between modules within a library are resolved once and for all by the linker when the library is created.

Figure 5.8 shows how a tree of modules may be joined in successive steps, first to form libraries, and finally to make the executable program. We will term this process *hierarchical binding*. It should be clear that hierarchical binding is most profitable when a program contains collections of closely knit modules which have reached a stable point in their development cycle. Joining such a collection of modules into a library not only increases linker performance but also serves as a means of abstraction; the details of the functionality of the individual modules may be disregarded and replaced by the functionality of the library.

5.11.2 The hPaster

Because of the complex dependencies that exist between Zuse modules the potential profit of hierarchical binding applied to Zuse is greater than for other similar languages. All the inter-modular exchange of information performed by the sPaster when creating an executable program can be performed by a *hierarchical Paster* (*hPaster*) to form a library out of a collection of modules: calls to inline procedures exported by any of the modules in the collection can be expanded, expressions can be partially or completely evaluated, and code can be generated for procedures which make use of no more inter-modular information than that which may be gleaned from the modules in the collection. With a few exceptions the algorithms of Section 5.8 will remain intact:

Phase 1 Instead of traversing the import graph to determine the modules to take part in a program, Phase 1 is given an explicit list of modules to be bound into a library. The hPaster Phase 1 creates a new object file, rather than an executable file, and gives it the library name selected by the user. The data and code read from the modules of the library is written to the Data and Binary Code Sections of the object file.

Phase 2 The hPaster can not assume, as in Algorithm 5.9, that all necessary expressions and inline procedures are available. Instead, only those expressions whose arguments are available should be evaluated, and only calls to those inline procedures whose code is available should be expanded. All (evaluated or unevaluated) expressions are written to the new object file's Expression Section, and the inline procedures are written to the Inline Section.

Phase 3 The hPaster's Phase 3 fills in the newly calculated expression values, expands calls to available inline procedures, and thereafter generates code for those deferred procedures which contain no further references to unknown entities. The generated binary code is written to the Binary Code Section of the object file, while the remaining deferred procedures are written to the Deferred Code Section.

Phase 4 Phase 4 will in general not be able to perform any relocations, since the final position of the generated code is not known. Instead the relocation information read during Phase 1 and generated during Phase 2 and 3 is written to the Relocation Section of the new object file.

We should note that the hPaster's Phase 3 will have to process *all* deferred procedures, since (unlike in Algorithm 5.10) the hPaster can not know which procedures may be referenced at run-time.

5.12 Summary

The purpose of this chapter has been to investigate the requirements imposed on translating systems by languages such as the one presented in Chapter 3 and to present one concrete translating system design. Since our main targets are system implementation languages, our primary design goal has been the production of highly optimized code, even if that must be at the expense of prolonged translation times. This goal has lead us to favor our present design over the alternatives in Section 5.10.

It would, however, be wrong to assume that translating systems such as ZTS will always lead to excessively long translation times. On the contrary, if we consider the total time spent compiling and binding a program over the *program's entire lifetime*, it may well be that ZTS will compare quite favorably to other more traditional systems. The reason is, of course, that ZTS allows modules with completely abstract specification units. If the information stored in a specification unit can be kept at a bare minimum, the risk of having to change that information is minimized, and so is the risk of trickle-down recompilations.

While the ZUSE language was from its conception designed to take advantage of high-level binding, other languages that were designed with traditional translation techniques in mind may also benefit from ZTS-style translation. This is particularly true of Modula-3, which, in contrast to most other languages, has features that *require* post-compilation exchange of inter-modular information. In addition to being able to handle Modula-3's opaque and partially revealed object type, ZTS would also allow Modula-3 to be extended with *statically allocated* object types, today missing from the language.

Others have presented solutions somewhat similar to ours. Particularly relevant are the works of Sale [183], Chow [48, 94], Horowitz [104], and Celentano [45, 46]. However, the works of Sale and Chow appear in print around the same time as the first published account of the work reported in this chapter (Collberg [52]) and should be considered independent work. It is furthermore not exactly clear how the Milano-Pascal linker was actually implemented, since both the report describing the work (Paganini [164], in Italian) and the implementation have apparently been lost (Ghezzi [82]). The translating system proposed by Sale was never implemented [184].

Our design is unique in many respects: It is the first to use link-time processing to handle problems (such as the construction of object type templates and the implementation of efficient run-time type tests) that arise when object-oriented programming meets separate compilation and type encapsulation. Previous solutions, such as the ones used in the SRC Modula-3 implementation,

use expensive run-time processing instead. Also, while others have proposed the use of link-time processing either to improve encapsulation (Celentano and Sale) or cross-module optimization (Chow and Wall [214, 216, 217]), ZTS is the first functional system which is designed to handle both. We will see from the following chapters, that ZTS is also the first system to use distributed and incremental translation techniques for improved control over encapsulation, and the first to use distributed techniques to facilitate inter-modular optimization.

Chapter 6

Distributed High-Level Module Binding

Law of Probable Dispersal:

Whatever it is that hits the fan will not be evenly distributed.

Anonymous

6.1 Introduction

It should come as no surprise that pasting is a considerably more expensive operation than conventional link-editing, especially for programs which rely heavily on abstract types and abstract inline procedures. Indeed, if the code generated during pasting is going to be subjected to aggressive optimization, pasting may prove to be prohibitively slow. Fortunately, however, pasting is a process which is relatively amenable to parallelization over a distributed processing system. This chapter will present a distributed version of the sequential Paster, called a *dPaster*, which runs on a network of workstations. Chapter 8 will present empirical results indicating that distributed pasting of large programs can be made as fast as, or faster than, conventional link-editing.

The chapter is organized as follows: Section 6.2 gives an overview of parallel and distributed architectures, in particular the workstation/server model. We next discuss various interprocess communication techniques (Section 6.3), especially the remote procedure call protocol. Earlier attempts at parallel translation are covered in Section 6.4, after which we describe the design of the

distributed Paster (Sections 6.5 through 6.9) followed by a section on possible future developments (Section 6.10). Finally, in Section 6.11 we discuss implementatation particulars and evaluate our design.

6.2 Distributed Processing Systems

A distributed system consists of a number of *sites* connected by a communication network. Each site has a processing unit, a local memory, and a unique name, and can communicate with other sites on the network by means of message passing. In a *loosely-coupled* distributed system message passing is the only means of communication; in a *tightly-coupled* system processors may also communicate through shared global storage.

There are several variations on the tightly-coupled model. The *multiprocessor* is, at the user level, analogous to a conventional single-processor multiprocess system, the difference being the existence of a large pool of processors with a shared memory. The operating system dynamically allocates processors and memory segments to users' tasks. *Array processors* consist of a large number of simple identical processing elements connected in a regular topology such as a vector or matrix. Array processors fall into the so-called *SIMD* model (single-instruction-multiple-data) where, at each stage of the computation, each processing element executes the same instruction on its local data.

The *workstation/server model* is a popular example of a loosely-coupled distributed system. In this model, a site is either a single-user computer providing considerable computing resources and powerful interaction facilities, or a server providing access to shared resources such as file stores and printers. Workstations and servers are linked by a high-speed *Local Area Network* (LAN) such as the 10Mb/s Ethernet. Files are organized in a *network file system* [138, 187] allowing workstations access to files stored on remote file-servers. 121

Examples of such distributed filing systems are the **Xerox Distributed File System** (XDFS) and the **Sun Network File System** (NFS). Figure 6.1 shows the logical buildup of the workstation network on which the test results in Chapter 8 were obtained.

One important characteristic of workstation networks is the fact that at any one point in time there is likely to exist unused computing power. This has lead to the design of distributed application programs intended to solve computationally intensive problems by taking advantage of unused computational resources. Examples include distributed compilers, distributed animation programs, and distributed programs intended to solve large numerical or number-theoretical problems [174].

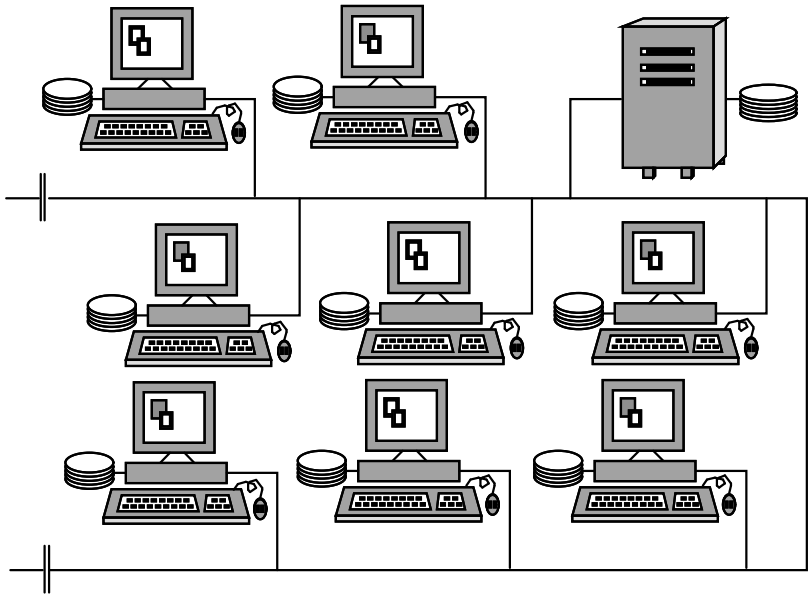


Figure 6.1: The workstation network on which the test results in Chapter 8 were obtained. The 8 workstations (Sun 3/80) have 8 MB of internal memory and an internal 60 MB disk. They are connected by a 10Mb/s Ethernet. The file-server (a Sun 3/50) has one 600 MB disk.

6.3 Models of Network Communication

Many distributed applications are built using a simple message passing model of communication. That is, the application assumes the existence of two primitive operations **send** (to send a packet of data from one site to another) and **receive** (to wait for an incoming packet). This model has the advantage of being simple to implement – the two operations are often provided as operating system primitives – but is often cumbersome to use and prone to programmer error. Other higher level models have therefore been invented, often designed as extensions to a simple message passing protocol. Examples include *Communicating Sequential Processes* (CSP), Linda, and *Remote Procedure Call* (RPC) protocols. A Linda [80] program, which can either be a concurrent single-processor program or an application distributed over a network, is centered around a global *tuple-space*. Processes may put tuples (named data items) in the tuple-space or fetch tuples from the tuple-space, effectively realizing process synchronization and data interchange. CSP is a programming notation rather than a complete language, but CSP style process communication, which

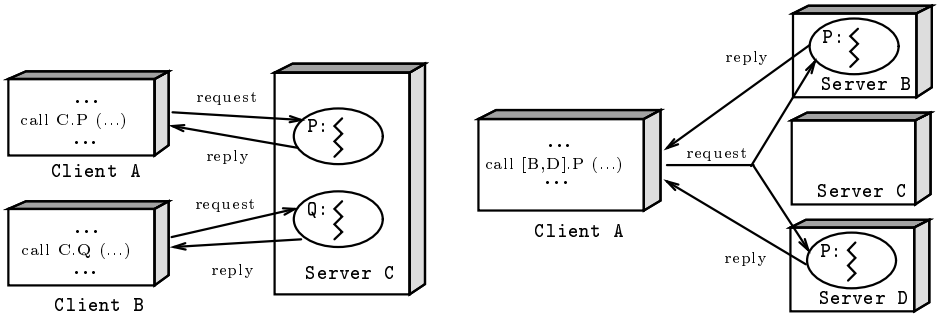


Figure 6.2: A server simultaneously serving two client calls (left) and, a client making a multicast call to two servers (right). Each call executes within its own lightweight process.

is based on synchronous message passing and selective communication, has been embodied in several languages, most notably Occam.

6.3.1 Remote Procedure Call

The semantics of a Remote Procedure Call [17, 25] is similar to that of an ordinary procedure call: A *client* program on one site calls a procedure executing on another site (a *server*) by sending it a *request* message (including a set of *in* parameters) and then waiting for a reply. When the remote procedure has finished executing, a *reply* message (including any *out* parameters) is sent back to the client, which resumes execution at the point after the call. There are, however, some semantic differences. Firstly, clients, servers, or message transfer may fail. Secondly, the caller and the callee execute in different address spaces – indeed on different sites – and therefore do not have access to each other's global variables. Lastly, the cost of an RPC is substantially higher than for a local procedure call; a remote call may take perhaps 100 to 1000 times longer to complete than a local call.

The distributed Paster which will be described later in this chapter has been implemented using an RPC protocol similar to the one described in a seminal paper by Birrell and Nelson [25]. It has the following characteristics:

- A server may service any number of calls concurrently.
- Each call on the server executes in its own lightweight process. This means that all calls share access to the same set of global variables.

- A call may take an arbitrarily long time to complete.
- Clients use timeouts to detect a failed server. Processes are assumed to be *fail-stop*.
- *In* and *out* parameters may be any arbitrarily complex data type.
- The protocol includes broadcast and multicast facilities.

Figure 6.2 (left) shows two clients simultaneously making a remote procedure call to the same server. The two procedures P and Q execute concurrently on the server, each within its own lightweight process. Figure 6.2 (right) is an example of a multicast remote procedure call. It is realized as one outgoing request message to the servers involved and one reply message from each server back to the client. All servers receive the same *in* parameters, but the client gets one set of *out* parameters from each server. A broadcast is similar to a multicast except that all servers receive the call and may elect to reply or not. In other words, a multicast call to n servers receives n results, whereas a broadcast call may yield zero-to-one result from each reachable server.

We extend our algorithmic notation from the previous chapter with remote procedure call primitives. In the first example below, a client C makes a call to the function P on server S with in-parameters x, y, z. When a call to P arrives at S, the actual parameters are copied to the formal parameters and the body of the function is executed. The **REPLY** statement returns results from the server back to the client. The second example shows C making a multicast remote procedure call to procedure P on sites $S_1, \ldots, S_s$, receiving s replies in return. Lastly we see an example of a broadcast remote procedure call receiving an indeterminate number of replies followed by a call discarding all except the first reply received.

1.
S: **ENTRY PROCEDURE** P (a,b,c:*int*) $\rightarrow$ *int*;
  REPLY $a + b * c$;
END P;

C: $v \leftarrow$ **CALL** $S.P\ (x, y, z)$;

2.
C: $v_1, \ldots, v_s \leftarrow$ **CALL** $[S_1, \ldots, S_s].P\ (1, 5, 7)$;

3.
C: $v_j, \ldots, v_k \leftarrow$ **CALL** $*.P\ (1, 5, 7)$;
C: $v_j \leftarrow$ **CALL** $1.P\ (1, 5, 7)$;

An RPC protocol like the one just described is a very attractive tool in the design of distributed applications. The similarity to ordinary local calls makes it easy for an application programmer to understand and use, and the high-level operations hide some of the details of network programming such as retransmissions, timeouts, and the marshaling of parameters. The overhead of RPC protocols in general and the current implementation in particular will be discussed later in this chapter.

6.4 Distributed Translation

Much work has gone into adapting language processors for use with parallel and distributed systems.[1] Several kinds of problems have been studied: *Parallelizing compilers* analyze sequential programs in order to detect implicitly parallel code which may be transformed to an explicit parallel form. Most of the work in this area has been restricted to vectorizing compilers for numerical FORTRAN programs. *Parallel compilers* try to speed up the compilation process by exploiting the inherent parallelism in a compiler and making it run on parallel hardware. Researchers working on *distributed compilers* focus on compilation speedup as well as the creation of distributed multi-user programming environments, in some cases under the assumption that the source code is distributed over the sites in the system. *Distributed building* [137, 75, 14, 15, 16] studies the related problem of configuration management in a distributed environment.

6.4.1 Models of Distributed Translation

Three conceptually different schemes for parallel and distributed compiling have been proposed (see Khanna [123]): Compilers using *functional decomposition* allocate each compilation subtask (lexical, syntactic, and semantic analysis and code generation) on different communicating processors. The lexical analysis processor reads the source program and transforms it into a stream of tokens which is passed on to the syntactic analysis processor. It, in turn, will analyze the syntactic structure and send it on along the pipeline for subsequent semantic analysis and code generation. In the *data decomposition* method an initial pass partitions the source code into roughly equal sized chunks, either arbitrarily or at the token or block level. The chunks are then processed concurrently by subcompilers residing on each processor. A final pass merges the results produced by the subcompilers. Finally, *grammatical decomposition* lets each

[1] Henceforth, we will use the term *parallel* for tightly-coupled systems such as array processors. The term *distributed* will be reserved for loosely-coupled systems such as the workstation/server model.

processor handle one particular linguistic construct. The source code is partitioned at the statement level and each chunk of code is passed to the processor handling the particular type of statement which the chunk contains.

Much of the work that has been done on parallel and distributed compilation has assumed that the program to be compiled is delivered to the compiler as a monolithic text. This is not always the case. Large programs will often be modularized, and in a distributed system the modules may be scattered among the sites of the system. Kaplan and Kaiser [120, 119] describe a *distributed language-based environment* designed to run on a network of workstations where a module resides on the workstation of the programmer who is responsible for the development of that module. Each module is stored as an attributed abstract syntax tree which is always kept up-to-date with respect to editing changes. A collection of tools such as structure-oriented editors, incremental parsers, semantic checkers, and code generators work on this common representation of the program and exchange information about changes with tools operating on other sites. In contrast to the other systems described here, distributed language-based environments are usually designed not with efficiency of translation but rather efficiency of team programming in mind.

Modularization has also spawned an interest in simple variants of the data decomposition method where ordinary sequential compilers are made to run concurrently on different sites in a distributed system, each compiling one or several source modules. Such systems, usually termed *distributed MAKE programs*, *distributed configuration managers*, or *distributed builders*, have to consider several problems:

- It must be decided which sites should compile which modules. This may depend on the current load on each site, the projected compilation time of each module, and from which sites the source code of the various modules is accessible.

- Where there exist compilation dependencies between modules, compilers running on different sites must be synchronized.

- Many configuration managers rely on the modification time of source and object modules in order to determine which modules have to be compiled. This may be a problem if the system time varies between sites.

6.4.2 Gains of Distributed and Parallel Translation

The advantages of a distributed translator lie, as we have seen, either in the distribution of the source (which may allow the cooperation of programmers

working on different machines) or in improved translator performance. The computational gain incurred by a parallel algorithm is measured in terms of its *speedup*[2] – its performance relative to a sequential algorithm solving the same problem. More formally:

Definition 6.1 (Speedup and Efficiency) If a parallel algorithm using p processors terminates in $T_p(n)$ time, and if the best possible serial algorithm terminates in $T^*(n)$ time, then the speedup of the algorithm is given by the ratio

$$S_p(n) = \frac{T^*(n)}{T_p(n)}.$$

The *efficiency* of the algorithm measures the fraction of time that a typical processor is usefully employed and is defined as

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)}.$$

When the optimal serial time $T^*(n)$ of a certain problem is unknown, $T^*(n)$ has to be given a different definition. One possibility is to let $S_p(n)$ be the *self-relative speedup* of the algorithm, in which case $T^*(n)$ is defined to be the time required by the parallel algorithm running on one processor.

If we have a certain parallel algorithm with a speedup of $S_p(n)$ it is natural to try to decrease the total solution time of a problem of a given size by increasing the number of processors. However, we will often experience that, due to *communication penalty*, there is an upper bound on the number of processors which can be gainfully employed in solving the problem. That is, as we increase the number of processors, the amount of communication necessary to coordinate the actions of the processors also increases. This is evident from some of the experimental parallel and distributed translation systems which have been reported in the literature. The speedups reported on are often fairly modest (between 2 and 4) and are achieved with a small number of processors where the gain from adding more processors is often negligible or negative. We give some examples of parallel and distributed translation systems from the literature:

- Baalbergen [15] reports that **Amake**, a distributed MAKE program for the Amoeba distributed operating system, achieves a self-relative speedup of 3.60 with a system of 5 sites when compiling a program made up of "several tens of C source files." Doubling the number of processors only increases the speedup to 3.81.

[2]The terminology and definitions used in this section are taken from Bertsekas and Tsitsiklis [24].

- Boehm and Zwaenenepoel [30] report on a distributed Pascal compiler running on Sun-2 workstations connected by an Ethernet network. A sequential parser builds a syntax tree which is split up into subtrees and sent out to attribute evaluators executing on remote sites. A network of 6 workstations is reported to achieve a speedup of up to 3.

- Wortman [231, 118, 233], whose concurrent Modula-2+ compiler runs on a shared memory multiprocessor, reports that the speedup achieved is influenced by the size of the modules compiled. For large modules 3 processors achieved a self-relative speedup of 2.6, whereas 6 processors reached a speedup of 4.1. For small modules the speedups were 1.7 and 1.9 respectively. The compiler creates a separate compilation process for each imported interface and for each function or procedure. Each compilation process is made up of several pipelined tasks, starting with a lexical analyzer and ending with a task which merges the produced code with the code produced by other processes.

- Katseff [121] presents an assembler running on a message-passing multiprocessor whose design is based on the data decomposition-technique. The inherent simplicity of assemblers compared to compilers is evident from the speedups reported by Katseff: 4 processors reached a self-relative speedup of 3.77, whereas 8 processors achieved 6.8. Adding more processors did not increase the speedup.

- That the simplicity of the compiled language has positive effects on the speedup achieved by a parallel compiler is confirmed in Gross [85]. Their cross-compiler, which runs on an Ethernet-based network of Sun workstations and produces code for a systolic array computer, achieves a "a speedup ranging from 3 to 6 using not more than 9 processors." This speedup can also be explained by the aggressive optimization techniques used,[3] which creates ample opportunity for parallelization.

6.4.3 Distributed Binding

Most of the parallel and distributed compilers reported on here suffer from the same problem: The distributed compilation phase is followed by a sequential link-edit phase. As link-editors become more sophisticated [48, 214, 218, 215, 216, 217, 94, 168, 186] and as programs grow larger and more modularized, the linking phase becomes something of a bottle-neck in the edit-compile-link

[3] The authors report that sequential compilation of a 300 line function takes $\approx$ 20 minutes, and that "compilation times measured in hours are not unusual."

cycle of program development. Considering that all other aspects of language translation have been subject to attempts at parallelization, it is remarkable that (apart from a remark by Katseff [121]) there has yet to be any work on parallelizing the linking process. Although the rest of this chapter will focus on distributed *high-level* binding, it is conceivable that some of the results will carry over to conventional link-editing as well.

6.5 The Distributed Paster

In the previous chapter we discussed in detail the design of the sPaster, a high-level module binder. It will be evident from the empirical tests in Chapter 8 that pasting is a considerably more expensive operation than conventional link-editing. This should come as no surprise since the Paster takes over some of the tasks usually performed during compilation, such as storage allocation, inline expansion, optimization, and code generation. The rest of this chapter will be devoted to the exploration of one possible avenue of Paster speedup, namely the design of a *distributed* Paster, the *dPaster*.

The distributed Paster can be said to be based on the *data decomposition* model of parallelization, in that each site taking part in the build process is responsible for a subset of the modules which make up the program. The distributed Paster is made up of three kinds of processes:

Master The Master is invoked by a user in the same way he would a sequential Paster. It is the main conductor of processing, deciding which site should process which modules, initiating the various processing phases, and relaying possible errors back to the user.

Slave There is (a maximum of) one Slave running on each site. All Slaves are dormant until contacted by the Master. They then wake up, take part in a bind orchestrated by the Master, after which they again return to their dormant state.

Servant There is one Servant process running on the local network, its job being to maintain the result of the bind, the *executable file*. Like the Slaves, the Servant process is dormant until awakened at the beginning of a bind. Static data, machine code, and relocation information produced by the Slaves is sent to the Servant as it becomes available. The Servant is responsible for writing the code and data to the executable file and performing the updates according to the relocation information received.

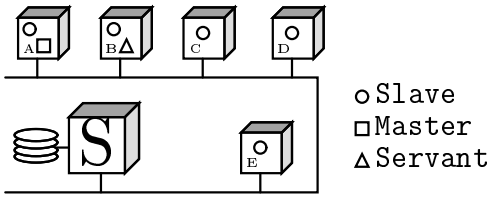


Figure 6.3: A network consisting of one file-server and five workstations. There are seven active dPaster processes: one Master, one Servant, and five Slaves. There is no process active on the file-server (S).

Figure 6.3 shows a network consisting of 5 workstations A–E, and a file server S. There are five Slaves, one per site, one Servant process (on site B), and one Master (on site A).

In the following we will be making certain assumptions about the underlying distributed system:

1. The system is an instance of the workstation/server model.
2. In addition to simple message-passing the network also supports *broadcasting* and *multicasting*.
3. All sites have access to the object code files of all modules.

The algorithms presented later in this chapter will make heavy use of multicasting, and it is therefore imperative that the underlying communications system implements this efficiently.

6.5.1 An Overview of the Distributed Paster

The design goal of any distributed application is to keep processes as independent of each other as possible. The more data that has to be shared among processes, and the more processing tasks which require processes to synchronize their activities, the more communication overhead will be generated. On the other hand, programming distributed applications is a notoriously difficult task with many possible pitfalls, and a clean division of labor between processes – even at the price of extra communication – may pay off in a more stable program. Hence, we have kept the first three phases of the sequential Paster in our design of the distributed Paster. The beginning of each phase serves as the primary *point of synchronization*; when a Slave has finished the processing of

one phase it must wait for the other Slaves to finish before proceeding. Organizing the processing in this regular fashion has the advantage that every Slave at the onset of a phase knows exactly which information is available to it and which is not. The disadvantage is a certain loss of Slave independence, which may lead to reduced efficiency of the Paster.

In addition to the three main phases of the distributed Paster, whose missions are similar to those of the corresponding phases in the sPaster, we have included an introductory phase, phase 0, responsible for initiating communication between the processes. Phase 1 finds the modules which are to take part in the build, and builds the different tables used during subsequent phases. It also decides on the distribution of the modules among the Slave processes, attempting to balance the load on the different sites. Phase 2 evaluates expressions, expands inline-in-inline procedures, and distributes the results to the Slaves who will need them during phase 3. Phase 3, which is virtually identical to its sPaster namesake, processes deferred procedures. The remainder of this section will examine the design of each of the three phases in turn.

The following definitions will be in effect for the rest of this chapter:

s	: The number of Slaves.
$S_1, \ldots, S_s$	: The Slaves.
m	: The number of modules being bound.
n	: The number of expressions and the number of nodes in the expression graph.
d	: The average out-degree of the nodes in the expression graph.
α_e	: The maximum number of expressions per packet.
α_v	: The maximum number of values (*evaluated expressions*) per packet.

6.6 Phases 0 and 1

The design of Phase 1 is interesting mainly because of its significance to the efficiency of subsequent phases. It is during Phase 1 that the distribution of the modules over the Slaves is decided on, and this essentially determines the amount of work each Slave has to perform. A skewed work distribution will make Slaves arrive at the synchronization points at widely differing times, which in turn will result in suboptimal performance. In addition to trying to anticipate the Slaves' workload, the distributed Phase 1 performs many of the same tasks as its sequential counterpart: the modules which are to take part in the bind have to be found, as must their corresponding object files; the constant expressions and the inline procedures have to be read; the Binary

Code, Data, and Relocation Sections have to be processed. Furthermore, before the pasting can commence, the Master must determine which Slaves are free to help out with the processing and inform them of any special conditions which apply to the particular session. This may include search paths for object files and flags to indicate whether code for debugging or profiling purposes should be included, load maps should be generated, etc.

6.6.1 Organizing the Work

As we have already mentioned, the distributed Paster is based on the *data decomposition* model of parallelization: each Slave is responsible for a subset of the modules which make up the program. This means that once a Slave has accepted the task of processing a given module, no other process – neither Master, Slave, nor Servant – has direct access to any data pertaining to that module. The owner Slave is responsible for finding the module's object file, reading and processing pertinent data from the file, and distributing the information to any needy process. The intention is to try to make Slaves work as independently of each other and of the Master as possible. A high degree of independence means a high degree of parallelization, which hopefully will lead to high factors of speedup. During Phase 1 the Slaves do not communicate with each other at all. They receive requests from the Master to process certain modules and send information about the modules back to the Master. Slaves also communicate with the Servant during Phase 1, sending it the Binary Code and Data Sections read from the modules' object files.

Finding the modules which are going to be part of the final program entails finding the closure of the module import graph with respect to the main module. To this end, the Slaves send the Module Table of each module to the Master, which keeps a *module map* M mapping the names of modules to unique integers. When the Master discovers that a module in a received Module Table is not yet in M, it is inserted and sent to a Slave for processing. The Slave reads the Module Table of the new module and passes it back to the Master. This process continues until all modules (directly or indirectly) referenced by the main module have been found and have been assigned to and processed by a Slave. The Master furthermore distributes relevant parts of the module map among the Slaves. When communicating among each other and with the Master the Slaves refer to individual modules only by their system-wide unique module numbers, which ensures that all Slaves share the same view of the program being built.

6.6.2 Working in Step

Load balancing, attempting to make Slaves work approximately in step by assigning to each of them the appropriate amount of work, is currently achieved by estimating the time needed to process each module. A Slave makes this estimate for a module assigned to it on the basis of the information gleaned from the module's object file preamble (see Section 5.3), the current load on the site on which the Slave runs, and possibly other relevant information. The calculated load for a particular module is passed back to the Master, which keeps a record of the total amount of work currently assigned to each Slave. When the Master finds a new module which has yet to be processed, it is assigned to the Slave which currently has the lightest workload. More sophisticated load balancing algorithms are discussed in Section 6.10.1.

To determine the binding-time cost of processing a module we use an estimating function *cost* of the form

$$cost(m, s) = s.l(\lambda_1 + \lambda_2 m.d + \lambda_3 m.e + \lambda_4 m.i + \lambda_5 m.c)$$

where m is a module, $m.d$, $m.e$, $m.i$, and $m.c$ represent the logical sizes of the module's Binary Code+Data, Expression, Inline, and Deferred Code Sections respectively, and the λ_i's are empirically determined constants. The parameter s is the Slave who owns m and $s.l$ the current load of the site on which the Slave executes.[4] Setting $\lambda_1 = 1$ and $\lambda_{2,\ldots,5} = 0$ will assign to all Slaves (approximately) the same number of modules, regardless of the complexity of each module, whereas $\lambda_5 = 1$ and $\lambda_{1,\ldots,4} = 0$ will only balance Phase 3 processing.

6.6.3 Phase 0 and 1 Algorithms

We are now ready to present the algorithms run by the Master and the Slaves during the two initial phases. Before any actual binding operations can take place, the Master must make contact with the Servant and the Slaves which are to take part in the processing. Phase 0 is responsible for finding available processes and distributing to them information relevant during the entire pasting session.

Algorithm 6.1 (Phase 0 – The Master)

1. Locate (using a network-wide broadcast) one free Slave and retrieve from it the list of all available free Slaves and the Servant.

[4] $s.l$ might, for example, be 1 for an unloaded site and 100 for a site currently utilizing all of its resources. The current implementation sets $s.l = 1$ for all s.

$(S_1, \ldots, S_s, Servant) \leftarrow$ **CALL** 1.*FindSlaves* ();

2. Send (using a multicast call) to all Slaves global data pertinent to the session. This includes the network address of the Servant, command line switches, environment variables, object file search paths, etc. Also inform the Servant of the name of the program to be bound, so that it may create the resulting executable file.

   CALL $[S_1, \ldots, S_s]$.*GlobalData* (*Environment*,...);
   CALL Servant.*Create* (*Name of program*);

During Phase 1 the Master repeatedly selects a Slave with a minimum workload and sends it an unprocessed module in a *ProcessModule* call. It simultaneously listens for incoming *ModuleData* calls which contain the Module Table and other data pertaining to a particular module. The modules in the Module Table which are not already in the module map are inserted and are made available for subsequent *ProcessModule* calls. In reply to the *ModuleData* call the Master sends, among other things, the module map identifiers of the modules in the Module Table. This means that the module map is partially replicated at each Slave, with each Slave holding only the part of the map corresponding to the modules which it itself owns or which are directly imported by these modules. The processing terminates when the Master has no more unprocessed modules and has received a *ModuleData* call for each of the modules in the module map.

Algorithm 6.2 (Phase 1 – The Master)

1. Let M be the module map, mapping module names to the integers $1, \ldots, m$ (M^{-1} is the reverse mapping). M initially contains only the main module.

2. Let U be a queue (with the operations *Insert*, *Delete*, and *Empty*) of hitherto unprocessed modules. U, too, is initialized to contain the main module.

3. Let Q be a priority queue initially containing the Slaves $S_1, \ldots, S_s$, each with the priority 0. Q supports the operations *Promote (Q, S, P)* to increase Slave S's priority by P and *Min (Q)* which returns (without deleting) a Slave with minimum priority.

4. Let PC be the program counter of the resulting program. The Master runs the following Phase 1 main program (the termination condition of the loop has been omitted for brevity):

$PC \leftarrow 0$;
$m \leftarrow 1$;
$M \leftarrow \{(mainmodule) \mapsto m\}$;
$U \leftarrow \{m\}$;
LOOP
  $s \leftarrow Min\ (Q)$;
  $n \leftarrow Delete\ (U)$;
  CALL $s.ProcessModule\ (n,\ M^{-1}(n))$;
  WAIT $\neg Empty$ (U);
END;

5. On receipt of a call *ModuleData (S, N, F, Z, I)* from a Slave S, the Master executes the code below. N is the module processed by S, F is S's estimate of the cost of processing N, I is N's Module Table, and Z is the size of N's Binary Code and Data Sections.

ENTRY PROCEDURE *ModuleData* (S, N, F, Z, I) $\rightarrow$ *int, map*;
  FOR ALL V **SUCH THAT** $(V \in I) \wedge (V \notin M)$ **DO**
    INC (m);
    $M \leftarrow M \cup \{V \mapsto m\}$;
    Insert (U, m);
  END;
  $L \leftarrow$ *The part of* M *corresponding to the modules in* I;
  REPLY PC, L;
  INC $(PC,\ Z)$;
  Promote (Q, S, F);
END *ModuleData*;

The program segments above can be seen as two processes in a *producer-consumer* relationship. The consumer process takes modules from the queue U and sends them off to a Slave for processing, while the producer extracts new modules from the Module Tables of incoming calls and inserts them into U.

We see from the above algorithm that the Master is in control of two global data items: the module map M which is (partially) replicated on each Slave, and the program counter PC which resides solely with the Master. Centralized control of the program counter is necessary since the Slaves will produce code and data independently of each other, and each code or data segment produced must receive a unique location. Therefore, for each new code segment generated, the Slaves have to inform the Master of the segment's size and in return receive its start address, which is the current value of the Master's PC.

During Phase 1, the Slaves respond to the *ProcessModule* call from the Master and themselves make *ModuleData* calls to inform the Master of possible new modules. Binary Code and Data Sections read from the object file during Phase 1 are sent to the Servant in *BinaryCode* calls. Apart from this, the Slaves perform the same tasks as does the sequential Paster during Phase 1.

Algorithm 6.3 (Phase 1 – The Slave)

1. Let M_s be the Slave S's initially empty module map.
2. On receipt of a *ProcessModule (n, N)* call from the Master (n is the number and N the name of the module assigned to S), the Slave S executes the code below. First the data from the declaration part of the object file is read and the Master is sent any potentially new modules from the Module Table, together with the size of the Binary Code and Data Sections, and the calculated cost of processing the module. In reply the Slave receives the part of the module map which pertains to the new modules in the Module Table, and the program location where the binary code and data should reside. Last, the binary code and data is sent to the Servant, together with the pertinent relocation information.

ENTRY PROCEDURE *ProcessModule* (n, N);
  REPLY;
  $M_s \leftarrow M_s \cup \{N \mapsto n\}$;
  Open n's object file and read the preamble;
  Read n's Module Table, Expression *and* Inline Sections*;*
  Read n's data and code C of size Z with relocation R;
  Close n's object file;
  $F \leftarrow cost(n, S)$;
  $I \leftarrow$ *The modules in n's* Module Table *but not in* M_s;
  $(PC, L) \leftarrow$ **CALL** Master.*ModuleData* (S, M, F, Z, I);
  $M_s \leftarrow M_s \cup L$;
  $A \leftarrow$ *Addresses of code and data objects read*;
  CALL Servant.*BinaryCode* (PC, C, R, A);
END *ProcessModule*;

6.6.4 Summary – Phase 1

Two important decisions are made during Phase 1: which modules are to take part in the build and which Slaves should process which modules. These decisions uniquely determine the state of the distributed Paster at the completion of the phase and the amount of work each Slave will have to perform during subsequent phases. Each Slave holds the constant expressions and the inline

procedures of the modules assigned to it and will have to share these objects with the other Slaves during Phase 2. Furthermore, each Slave will have to generate code for all the modules it owns. This is a consequence of the load balancing principle currently employed, which is based on an initial estimate of the amount of work incurred by each module.

6.7 Phase 2

Phase 2 is in many ways the part of the Paster which is the most challenging to parallelize. It is during Phase 2 that the Slaves exchange information gathered during Phase 1, thereby paving the way for Phase 3. In fact, all Slave-to-Slave communication is carried out during Phase 2, ridding the computationally expensive Phase 3 from any disrupting communication.

As we saw from the previous section, Phase 1 partitions the node set of the CET graph into s parts, one part for each Slave. The primary objective of Phase 2 is to evaluate each of the CET expressions, and to replicate (at least) those values which each slave may need during Phase 3. The partitioning of the nodes of the graph implies that – from the viewpoint of a particular slave – some nodes will be *internal* while others will be *external*.[5] Internal nodes will, in general, not present any problems, while some method will have to be devised for slaves to access the values of external nodes.

The next few sections will examine several alternative evaluation methods and in Section 6.11 we will compare them with respect to their message complexity.

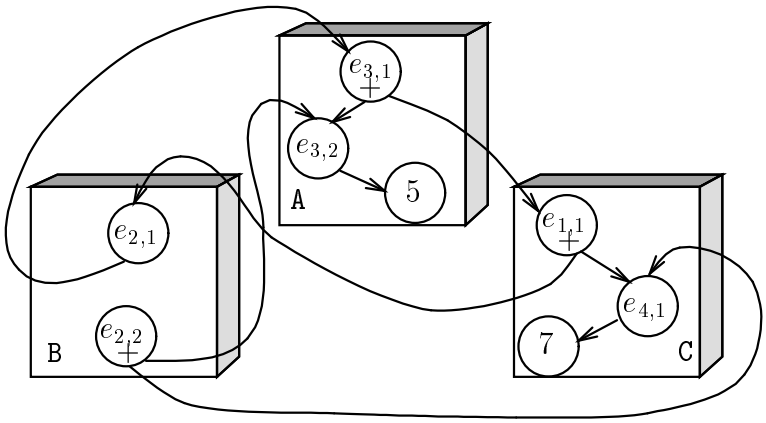
6.7.1 Distributed Depth-First Evaluation

We will start by examining three unsophisticated algorithms for expression evaluation, and then, in the next section, proceed with the algorithm actually employed in the current implementation. First, it should be obvious that simplicity can be achieved at the expense of a reduction in parallelism:

Algorithm 6.4 (sequential) Each Slave sends its part of the graph to an agreed upon special Slave, say S_1, which evaluates the entire graph and multicasts the results to the rest of the Slaves.

Algorithm 6.5 (redundant) Each Slave multicasts its part of the graph to all other Slaves. Once all Slaves have received every subgraph, they evaluate the complete graph in parallel.

[5] We will also talk about edges being *internal* or *external*, depending on whether they connect nodes residing on the same or different sites.



Site B		
$e_{2,1}$	$\triangleq$	$e_{3,1}$
$e_{2,2}$	$\triangleq$	$e_{3,2} + e_{4,1}$

Site A		
$e_{3,1}$	$\triangleq$	$e_{3,2} + e_{1,1}$
$e_{3,2}$	$\triangleq$	5

Site C		
$e_{1,1}$	$\triangleq$	$e_{4,1} + e_{2,1}$
$e_{4,1}$	$\triangleq$	7

Figure 6.4: The distributed expression graph of four modules distributed over three Slaves. The graph contains a cycle comprising expressions $e_{1,1}$, $e_{2,1}$, and $e_{3,1}$.

Another straightforward solution is to mimic the sequential evaluation method:

Algorithm 6.6 (distributed depth-first) Let each Slave simultaneously evaluate its local subgraph depth-first. When a Slave comes upon an external edge it sends a message to the Slave which owns the corresponding node with a request for that node's value.

This last method requires every Slave to know the owner of every node in the graph. Furthermore, extra care has to be taken in those cases when the graph contains a cycle. Consider, e.g., Figure 6.4 where four modules $(M_1, \ldots, M_4)$ reside on the three Slaves A, B, and C. Circular definitions of manifest constants in M_1, M_2, and M_3 have resulted in a cycle comprising the nodes $e_{1,1}$, $e_{2,1}$, and $e_{3,1}$.[6] A distributed depth-first evaluation of this graph will result in a deadlock situation where Slaves A, B, and C will be waiting for values from each other.

A number of techniques have been devised in order to handle distributed deadlocks (see, for example, Bracha [34], Helary [93], and Singhal [196]): *Dead-*

[6] In this and the following examples the actual attributes involved in a given expression will be left unspecified.

lock avoidance algorithms make sure that deadlocks can never occur by maintaining a global state (a so-called *wait-for graph*) which is consulted and updated before a possible deadlock-causing operation is attempted. *Deadlock detection algorithms* are run at regular intervals, or when processing seems to have come to a standstill, to check for the presence of a deadlock situation. If it is determined that a deadlock situation is at hand, the offending dependencies are located and broken. Since it would seem that cyclic expression graphs would be quite unusual – they are, after all, the result of rather obscure programmer errors – the appropriate way to handle deadlocks would seem to be by employing a detection algorithm. Under normal circumstances, it would not interfere with the evaluation process. We can, however, get away without a separate deadlock-breaking algorithm by extending the depth-first evaluation algorithm with a *site-migration* facility.

Algorithm 6.7 (site migration) We allow several (lightweight) evaluation processes to execute concurrently at each site, possibly evaluating the same value simultaneously. When a process needs the value of an external node, instead of asking the owner for its value, the process itself *migrates* to the appropriate site and continues the depth-first evaluation. In order to avoid getting itself into a deadlock situation it does so *regardless* of whether another process is currently evaluating the same node. A process evaluating a node on a cycle will detect the cycle after it has been completely traversed and the process has returned to the node where the evaluation commenced.

Consider again the cycle $e_{1,1}$, $e_{2,1}$, and $e_{3,1}$ in Figure 6.4. Slave A needs the value of node $e_{1,1}$ in order to evaluate node $e_{3,1}$. It therefore migrates to site C and from there to site B to evaluate $e_{2,1}$. From site B it returns to node $e_{1,1}$ on site A and can therefore conclude that $e_{1,1}$ is on a cycle.

6.7.2 Multicast Evaluation

The last method we will study, and the one employed in the actual implementation, makes heavy use of the multicast facility. It is a conceptually simple iterative algorithm where each Slave evaluates as much as it can, distributes the new values to the other Slaves, and waits for the arrival of new values. The exact method of evaluation depends on the nature of the attributes being computed, but we give the general structure of the evaluation algorithm run by Slave S_k on the next page:

ENTRY PROCEDURE *Evaluate* ();
  REPLY;
  $\mathcal{I}_x$: *Initialize;*
  LOOP
    $\mathcal{E}_x$: *Compute new data,* X_{new}*;*
    $\mathcal{T}_x$: *Determine the data* X_{out} *which should be transmitted;*
    IF $X_{out} \neq \{\ \}$ **THEN CALL** $[\ldots, S_j, \ldots]_{j \neq k}$.*Values* (k, X_{out}) **END**;
    WAIT *Incoming Values-calls*;
  END;
END *Evaluate*;

ENTRY PROCEDURE *Values* (k, X_{in});
  REPLY;
  $\mathcal{U}_x$: *Store the new data* X_{in}*;*
END *Values*;

Based on this sketch, Algorithm 6.8 below contains the plug-in statements for the evaluation of synthesized attributes. The set E_{left} holds the expressions which have yet to be evaluated, E_{new} the ones which were evaluated during the present round, and E_{out} the global expression values which are to be distributed to the other Slaves.

ALGORITHM 6.8 (EVALUATE SYNTHESIZED ATTRIBUTES)

$\mathcal{I}_e$: $E_{left} \leftarrow \{$*All* $e_{m,i}.\mathrm{a}$ *such that (*S_k *owns* m*)* $\wedge$ (a *is synthesized)*$\}$;

$\mathcal{E}_e$: $E_{new} \leftarrow \{$*All* $e_{m,i}.\mathrm{a} \in E_{left}$ *whose arguments have known values*$\}$;
  $E_{left} \leftarrow E_{left} \setminus E_{new}$;

$\mathcal{T}_e$: $E_{out} \leftarrow \{$*All tuples* (m, i, a, V) *such that*
    $(e_{m,i}.\mathrm{a} \in E_{new}) \wedge (e_{m,i}.\mathrm{a} \triangleq \mathrm{V}) \wedge e_{m,i}.\mathrm{global}\}$;

$\mathcal{U}_e$: **FOR EACH** $(m, i, a, V) \in E_{in}$ **DO** *Let* $e_{m,i}.\mathrm{a} \triangleq V$ **END**;

Computing the inherited attribute *ref* is slightly more complicated, as can be seen from Algorithm 6.9 below. On Slave S_k R_{ref} starts out containing the set of $e_{m,i}$:s corresponding to the bodies of the modules owned by the slave, and during each subsequent round it will hold the new $e_{m,i}$:s received from the other slaves. R_{new} will hold the $e_{m,i}$:s reachable through the *calls*-attributes of the expressions in R_{ref}. R_{out}, finally, holds the set of procedure identifiers which are to be distributed to the other Slaves, while R_{sent} holds those $e_{m,i}$:s which will not need to be transmitted again.

Algorithm 6.9 (evaluate inherited attributes)

$\mathcal{I}_r$: $R_{ref} \leftarrow \{$*All* $e_{m,i}$ *such that* $(S_k$ *owns* $m) \wedge e_{m,i}$.ref $\}$;
$R_{sent} \leftarrow \{\}$;

$\mathcal{E}_r$: $R_{new} \leftarrow \{$*All* $e_{m,i}$ *reachable from some* $e_{n,j} \in R_{ref}\}$;

$\mathcal{T}_r$: $R_{out} \leftarrow \{$*All tuples* (m,i) *such that*
$(e_{m,i} \in R_{new}) \wedge (e_{m,i} \notin R_{sent}) \wedge (S_k$ *does not own* $m)\}$;
$R_{sent} \leftarrow R_{sent} \cup R_{new}$;

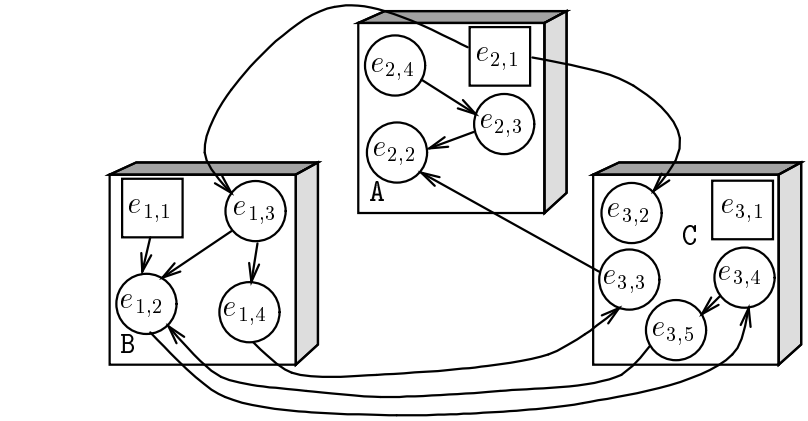
$\mathcal{U}_r$: $R_{ref} \leftarrow \{$*All* $e_{m,i}$ *such that* $((m,i) \in R_{in}) \wedge (S_k$ *owns* $m) \wedge (\neg e_{m,i}$.ref$)\}$;
FOR EACH $(m,i) \in R_{in}$ **DO**
Let $e_{m,i}$.ref $\triangleq \mathbb{T}$;
$R_{sent} \leftarrow R_{sent} \cup \{e_{m,i}\}$;
END;

Although superficially uncomplicated, careful study of the algorithms above reveals several subtleties. How, for example, are cycles among the synthesized attributes discovered? And when does the algorithm terminate? It turns out that these two questions are interrelated. Applying Algorithm 6.8 to the simple example of Figure 6.4 spawns the following communication between the Slaves:

Slave	Pass 1	Pass 2
A	$\{e_{3,2}\}$	$\{\ \}$
B	$\{\ \}$	$\{e_{2,2}\}$
C	$\{e_{4,1}\ \}$	$\{\ \}$

During the first round of Algorithm 6.8 Slave A will multicast the value of $e_{3,2}$ and Slave C will send $e_{4,1}$. Once these values have arrived at Slave B, $e_{2,2}$ may be evaluated and its value distributed to the other Slaves. This, however, will not further any new computations, and the processing will come to a standstill, even though there are still three unevaluated expressions, $e_{1,1}$, $e_{2,1}$, and $e_{3,1}$. The fact that there remain expressions which have yet to receive a value, and still no more evaluations are possible, indicate the presence of one or more cycles. Indeed, $e_{1,1}$, $e_{2,1}$, and $e_{3,1}$ form such a cycle.

Algorithm 6.9 displays a similar behavior. Consider the example in Figure 6.5 which shows a configuration of three Slaves, three modules, and the call-graph of thirteen procedures. The roots of the call-graph (corresponding to module bodies) are boxed; ordinary procedures are circled. The calculation of referenced procedures – which corresponds to the calculation of the nodes of the closure of the graph with respect to the boxed root nodes – in this case runs in three passes:



Site B

$\boxed{e_{1,1}}$	$\triangleq$	$[e_{1,2}]$
$e_{1,2}$	$\triangleq$	$[e_{3,4}]$
$e_{1,3}$	$\triangleq$	$[e_{1,2}, e_{1,4}]$
$e_{1,4}$	$\triangleq$	$[e_{3,3}]$

Site A

$\boxed{e_{2,1}}$	$\triangleq$	$[e_{1,3}, e_{3,2}]$
$e_{2,2}$	$\triangleq$	$[\,]$
$e_{2,3}$	$\triangleq$	$[e_{2,2}]$
$e_{2,4}$	$\triangleq$	$[e_{2,3}]$

Site C

$\boxed{e_{3,1}}$	$\triangleq$	$[\,]$
$e_{3,2}$	$\triangleq$	$[\,]$
$e_{3,3}$	$\triangleq$	$[e_{2,2}]$
$e_{3,4}$	$\triangleq$	$[e_{3,5}]$
$e_{3,5}$	$\triangleq$	$[e_{1,2}]$

Figure 6.5: The distributed call-graph of three modules distributed over three Slaves.

Slave	Pass 1	Pass 2	Pass 3
A	$\{e_{1,3}, e_{3,2}\}$	$\{\,\}$	$\{\,\}$
B	$\{e_{3,4}\}$	$\{e_{3,3}\}$	$\{\,\}$
C	$\{\,\}$	$\{e_{1,2}\}$	$\{e_{2,2}\}$

In the first pass each Slave multicasts the nodes reachable from the root nodes and residing on other Slaves. Subsequent passes multicast the nodes deemed reachable from the nodes which were received in the previous pass. After the third pass, the closure has been computed and processing comes to a standstill.

6.7.3 Detecting Termination

We see that both in the case of evaluating arithmetic expressions and in computing the closure of the call-graph we arrive at a point where the distributed computation ceases because no Slave has anything more to compute and no Slave receives more input from other Slaves to further its computation. Yet, it is impossible for any individual Slave in a passive state to determine whether

computation as a whole has ceased or whether some other Slave is active and may eventually distribute data which may make new computations possible. This problem is usually termed the *Distributed Termination Detection Problem* (DTDP) and has been studied in vast detail [66, 77, 93, 105, 144, 205, 211]. The following definitions are adapted from Huang [105]:

Definition 6.2 (Termination) A process taking part in a distributed computation may be either *active* or *passive*. Only active processes may send messages. An active process may become passive at any time, and a passive process may be reactivated by receiving a message. A distributed computation is said to have *terminated* iff all processes are in a passive state and there are no messages in transfer.

A *Distributed Termination Detection Algorithm* (DTDA) is a protocol which runs concurrently with a distributed application and reports when said application has terminated. There is a multitude of such algorithms intended for use with various models of distributed systems. Algorithms may assume a certain network topology, synchronous or asynchronous communication, the existence of an external leader site, etc. The evaluation algorithm presented earlier in this chapter has some very special properties which makes it possible to employ a new and simple DTDA. First of all, the evaluation algorithm only makes use of multicast messages. In other words, every message sent by one of the Slaves is received by all other Slaves. Secondly, all messages are in the form of remote procedure calls. This means that message-passing is *synchronous*; a Slave knows after having sent a message and having received the replies that the message has arrived at its destinations. Lastly, our RPC protocol allows calls to take an arbitrarily long time to complete. Our termination algorithm makes use of all these facts.

Algorithm 6.10 (termination-detection)

1. Let there be s sites $S_1, \ldots, S_s$ executing a distributed application. Sites communicate exclusively by multicast remote procedure calls.

2. A site is either in an *active* state (performing some computation on local data, sending messages to other sites, or itself responding to such messages) or in a *passive* state which is characterized by the absence of local computation and message-passing. A site may enter a passive state only after having determined that all remote calls it has initiated have returned, all incoming calls have been returned, and that no further local processing is possible.

3. Let each site S_i maintain a count C_i of all multicast messages sent or received during the execution of the underlying algorithm, excluding any control messages introduced by the termination detection protocol.

4. Choose one site, say S_1, to be the detector of termination.

5. On entering a passive state, S_1 sends a multicast remote procedure call **DETECT** to $S_2, \ldots, S_s$. On receipt of a **DETECT** call site S_i continues processing until a passive state is reached. At this point a reply is sent to S_1 including the current value of C_i.

6. Having received replies from all other sites, S_1 compares the values of $C_1, \ldots, C_s$. If all values are equal, S_1 concludes that the underlying algorithm has terminated. Otherwise, the process is repeated with a new **DETECT** call once S_1 again becomes passive.

We will prove the correctness of the algorithm by initially considering a simpler variant, identical to Algorithm 6.10, except for the following points:

ALGORITHM 6.11 (TERMINATION-DETECTION-A)

1. Associate with each message a unique identifier and let each site maintain a set M_i of the identifiers of the messages it has either sent or received.

2. On reaching a passive state let S_i return from the **DETECT** call with the value of M_i.

3. When S_1 has received replies from all other sites, the M_i's are compared for equality. If $M_1 = M_2 = \ldots = M_n$, S_1 may conclude that the underlying algorithm has terminated.

THEOREM 6.1 (TERMINATION-DETECTION-A) Algorithm 6.11 detects termination iff the underlying computation has terminated.

PROOF

($\Leftarrow$) When the underlying computation terminates it has done so because all sites have determined that there is no more basic computation to be carried out. Hence all sites have entered a passive state and returned their M_i. M_i contains all messages S_i has seen; i.e. those S_i has sent itself and those it has received from $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_s\}$. If S_i has sent a message m, m must be in M_i (since it was sent by S_i) and it must be in the M_j's of all other sites

since it was received by them.[7] Therefore $M_1 = M_2 = \ldots = M_n$ and the algorithm will report termination.

($\Rightarrow$) Assume that Algorithm 6.11 has reported termination, that the S_i's have entered their passive state in the order $\{S_1, S_2, \ldots, S_{s-1}, S_s\}$,[8] that they have returned the value $M = M_1 = M_2 = \ldots = M_n$, but that there are still some active processes. Then some site, say S_i, after having returned M_i and entered a passive state, will have been reactivated by a message $m \notin M_i$ originating from site S_j. Then $j > i$ since $\{S_1, S_2, \ldots, S_{j-1}\}$ were passive at this time. But then $m \in M_j$ which is a contradiction since $M_j = M = M_i$. Therefore, we can conclude that the detection algorithm will not report false termination. □

Theorem 6.2 (termination-detection) Algorithm 6.10 detects termination iff the underlying computation has terminated.

Proof

($\Leftarrow$) By the first part of the proof of the previous theorem, if all basic computation has ended then in Algorithm 6.11 $M_1 = M_2 = \ldots = M_n$. But since $C_i = \|M_i\|$, $C_1 = C_2 = \ldots = C_n$ and Algorithm 6.10 will report termination.

($\Rightarrow$) We use an argument similar to the one in the second part of the proof of the previous theorem. If some passive site S_i was reactivated by a message m originating from site S_j, then $j > i$. But since S_j has seen at least the same messages as S_i at the time it goes passive, and since m arrived at S_i after it went passive and hence is not included in its count C_i, $C_j > C_i$, which is a contradiction. Therefore, there will be no detection of false termination. □

It seems reasonable at this point to question whether it is actually necessary for the sites to maintain even a count of the messages they have seen. Let us, for a moment, consider an algorithm where one site (exactly as in the two algorithms just presented) repeatedly asks the other sites if they have terminated. When a site becomes passive it replies – not with the set of messages it has seen or even with the message count – but with a simple **yes**. A simple three site example is enough to show that this method is not sufficient: Sites S_2 and S_3 are both active; site S_1 is passive and sends a **DETECT** message to the other two sites. Site S_2 goes passive and returns **yes** to S_1. Site S_3 sends a message which reactivates S_2, goes passive, and returns **yes** to S_1. S_1

[7] Remember that all messages are in the form of multicast remote procedure calls and that a site does not enter a passive state until all of its outgoing calls have been completed.

[8] After possible renumbering. If two sites enter their passive state at the same time we assign an arbitrary ordering between them.

has now received an affirmative reply from the two other sites and therefore, erroneously, concludes that the processing has terminated.

It should be noted that Algorithm 6.10 is not immune to failures. As has already been mentioned, we assume that the underlying RPC protocol detects failed sites and makes sure that all messages eventually reach their destination. If, for example, we allow sites to crash, only to reenter the system at a later time, a more fault-tolerant protocol than Algorithm 6.10 must be employed.

6.7.4 Distributed Inline Expansion

At the completion of Phase 1 the set of inline procedures has been partitioned over the s Slaves, and each inline procedure resides on the Slave which owns the module exporting it. The task of Phase 2 is to ensure that when Phase 3 commences each Slave will have direct access to all inline procedures referenced by any module owned by the Slave. Once this has been accomplished each Slave knows that it has local access to all the global entities it will need during the processing of Phase 3 – all expression values as well as all inline procedures. Consequentially, Phase 3 will require no inter-Slave communication.

The distributed Phase 2 must accomplish three things in connection with the inline procedures: Cycles in the inline graph have to be discovered, inline-in-inline calls have to be expanded, and the resulting procedures have to be distributed to the Slaves where they will be needed during Phase 3. The detection of inline cycles can easily be incorporated into the expression evaluation algorithms presented earlier. In Algorithm 6.12 below, the Slaves start by exchanging information about which of their procedures are inline. This is accomplished by the remote procedures *SendInline* and *IsInline*. Thereafter the auxiliary CET attribute *icalls* given to each inline procedure is initialized to the number of calls $e_{m,i}$ makes to external inline procedures.[9]

Algorithm 6.12 (inline cycle detection)

$\mathcal{I}_i$: *SendInline* ();
  $I_{left} \leftarrow \{$*All* $e_{m,i}$ *such that* (S_k *owns* m) $\wedge\, e_{m,i}$.inline$\}$;
  FOR EACH $(m, i) \in I_{left}$ **DO**
    FOR ALL $e_{n,j} \in e_{m,i}$.calls **SUCH THAT**
      (S_k *does not own* n) $\wedge\, e_{n,j}$.inline **DO**
      $e_{m,i}$.icalls $\leftarrow e_{m,i}$.icalls + 1;
    END;
  END;

[9] Algorithm 6.12 as presented does not account for cycles contained exclusively on one Slave.

$\mathcal{E}_i$: $I_{new} \leftarrow \{$*All* $e_{m,i} \in I_{left}$ *such that* $(e_{m,i}.\text{icalls} \triangleq 0)\}$;
$I_{left} \leftarrow I_{left} \setminus I_{new}$;

$\mathcal{T}_i$: $I_{out} \leftarrow \{$*All tuples* (m,i) *such that* $(e_{m,i}.\text{a} \in I_{new}) \wedge e_{m,i}.\text{global}\}$;

$\mathcal{U}_i$: **FOR EACH** $(n,j) \in I_{in}$ **DO**
FOR ALL $e_{m,i} \in I_{left}$ **SUCH THAT** $e_{n,j} \in e_{m,i}.\text{calls}$ **DO**
$e_{m,i}.\text{icalls} \leftarrow e_{m,i}.\text{icalls} - 1$;
END;
END;

PROCEDURE *SendInline* ();
$I_{out} \leftarrow \{$*All tuples* (m,i) *such that* $(S_k$ *owns* $m) \wedge e_{m,i}.\text{inline}\}$;
CALL $[\ldots, S_j, \ldots]_{j \neq k}$.*IsInline* (I_{out});
FOR $j \in 1 \ldots s-1$ **DO WAIT** *Incoming IsInline-call*; **END**;
END *SendInline*;

ENTRY PROCEDURE *IsInline* (I_{in});
FOR EACH $(n,j) \in I_{in}$ **DO** *Let* $e_{n,j}.\text{inline} \triangleq \mathbb{T}$ **END**;
REPLY;
END *IsInline*;

It should be fairly obvious that Algorithm 6.12 is really a distributed variant of topological sorting, as presented for example by Mehlhorn [148, pp. 4 ff.]. The algorithm starts by determining the out-degree of each node in the inline call-graph (the *icalls*-attribute), and then proceeds to distribute the set of nodes whose out-degree is 0. On receipt of a set of such nodes (I_{in}), a Slave decrements the *icalls*-attribute of each of its remaining inline procedures (I_{left}) that calls any of the procedures in I_{in}. The process continues until either I_{left} is empty on all Slaves, or all the processing comes to a stand-still due to a cycle in the graph. We illustrate the actions of Algorithm 6.12 using the call-graph of Figure 6.5 on page 6.5 (the example assumes that all procedures are inline):

Slave	*SendInline*	Pass 1	Pass 2	Pass 3
A	$\{e_{1,2},e_{1,3},e_{1,4}\}$	$\{e_{2,2},e_{2,3},e_{2,4}\}$	{ }	{ }
B	$\{e_{2,2},e_{2,3},e_{2,4}\}$	{ }	{ }	$\{e_{1,4}\}$
C	$\{e_{3,2},e_{3,3},e_{3,4},e_{3,5}\}$	$\{e_{3,2}\}$	$\{e_{3,3}\}$	{ }

Pass	$e_{1,2}$	$e_{1,3}$	$e_{1,4}$	$e_{2,2}$	$e_{2,3}$	$e_{2,4}$	$e_{3,2}$	$e_{3,3}$	$e_{3,4}$	$e_{3,5}$
0	1	2	1	0	0	0	0	1	1	1
1	1	2	1	0	0	0	0	0	1	1
2	1	2	0	0	0	0	0	0	1	1

In the table of out-degrees (*icalls*-values) above we note that only calls to external procedures are considered, and that prior to the first pass (pass 0 in the table) there are four procedures of out-degree 0 ($e_{2,2}$,$e_{2,3}$,$e_{2,4}$). These are transmitted during pass 1, after which only one new procedure ($e_{3,3}$) gets out-degree 0. After the third pass there is no more communication even though there are still four procedures with out-degree > 0. These are either part of the inline call cycle or call procedures on the cycle.

Once we have determined that the inline call-graph is acyclic we can safely proceed with the actual expansions. It should be clear that it is quite possible to employ techniques similar to any of the ones proposed for expression evaluation. The current implementation uses a technique similar to Algorithm 6.12, with one exception. Since inline procedures can be fairly large objects (especially after having been subject to a few expansions), and the implementation of our remote procedure call protocol is slightly inefficient in multicasting large messages, an inline procedure is only distributed to those Slaves who actually need it rather than to all Slaves. This scheme is likely to pay off in a system with a large module to Slave ($\frac{m}{s}$) ratio, where the probability of a particular Slave needing a particular inline procedure is fairly small.

We can now add one more item to the list of tasks which have to be accomplished during the distributed Phase 2: each Slave must be informed of which other Slaves need which of its inline procedures. Fortunately, this is a simple task which can be easily integrated with the calculation of the *ref*-attribute. We give each $e_{m,i}$ representing an inline procedure an initially empty set attribute *needed* where $S_j \in e_{m,i}$.needed on Slave S_k if S_k owns $e_{m,i}$ and Slave S_j needs $e_{m,i}$. Algorithm 6.9 is altered as follows:

$\mathcal{U}_r$: $R_{ref} \leftarrow \{$*All* $e_{m,i}$ *such that* $((m,i) \in R_{in}) \wedge$ (*S_k owns m*) $\wedge (\neg e_{m,i}.\text{ref})\}$;
FOR EACH $(m,i) \in R_{in}$ **DO**
Let $e_{m,i}.\text{ref} \triangleq \mathbb{T}$;
IF $\neg e_{m,i}$.inline **THEN** $R_{sent} \leftarrow R_{sent} \cup \{e_{m,i}\}$; **END**;
IF $e_{m,i}.\text{inline} \wedge$ (*S_k owns m*) **THEN**
$e_{m,i}.\text{needed} \leftarrow e_{m,i}.\text{needed} \cup \{S_j\}$;
END;
END;

The first **IF**-statement in the code fragment above ensures that each slave will always inform the other slaves of all the inline procedures it needs.

To complete Phase 2 we merely have to perform the actual inline expansions and distribute the results. We know from the first part of Phase 2 that there are no inline cycles and where the different inline procedures need to be replicated. The distributed inline expansion algorithm is sketched on the next page:

Algorithm 6.13 (inline expansion)

ENTRY PROCEDURE *Expand* ();
Replace all CET references in all inline procs with the newly computed values;
$I_{left} \leftarrow \{$*All* $e_{m,i}$ *such that (*S_k *owns* $m) \wedge e_{m,i}$.inline$\}$;

WHILE $I_{left} \neq \{\}$ **DO**
Expand all calls to procedures whose code is available;
$I_{new} \leftarrow \{$*All* $e_{m,i} \in I_{left}$ *without any remaining inline calls*$\}$;
$I_{left} \leftarrow I_{left} \setminus I_{new}$;

FOR EACH $e_{m,i} \in I_{new}$ **DO**
CALL [$e_{m,i}$.needed].*InlineProc* ((m,i),*code of* $e_{m,i}$);
END;
WAIT *Incoming InlineProc-calls*;
END;
REPLY;
END *Expand*;

ENTRY PROCEDURE *InlineProc* ((m,i), *code of* $e_{m,i}$);
REPLY;
Store the code of $e_{m,i}$*;*
END *InlineProc*;

6.7.5 Summary – Phase 2

We are now ready to put together the bits and pieces of the Phase 2 algorithms. Note that in all algorithms presented in this chapter, all intra-process synchronization has been omitted. Since all incoming remote procedure calls execute in their own lightweight processes, global data (such as the CET, the inline procedure store, and the global variables *calls* and *active* below) must of course be protected using monitors and condition variables or similar concepts.

Algorithm 6.14 (evaluation)

ENTRY PROCEDURE *Evaluate* ();
IF k$\neq$1 **THEN REPLY END**;
calls $\leftarrow$ 0;
active $\leftarrow$ `TRUE`;
$\mathcal{I}_i$; $\mathcal{I}_e$; $\mathcal{I}_r$;
LOOP
$\mathcal{E}_i$; $\mathcal{E}_e$; $\mathcal{E}_r$;
$\mathcal{T}_i$; $\mathcal{T}_e$; $\mathcal{T}_r$;

```
        IF (E_out, R_out, I_out)≠({}, {}, {}) THEN
            CALL [..., S_j, ...]_{j≠k}.Values (k, E_out, R_out, I_out);
            calls ← calls + 1;
        END;

        active ← FALSE;
        IF k=1 THEN
            (C_2, C_3, ···, C_s) ← CALL [S_2, ..., S_s].Detect ();
            IF calls = C_2 = ··· = C_s THEN REPLY END;
        END;
        WAIT Incoming Values-calls;
    END;
END Evaluate;

ENTRY PROCEDURE Values (k, E_in, R_in, I_in);
    active ← TRUE;
    calls ← calls + 1;
    REPLY;
    U_i; U_e; U_r;
END Values;

ENTRY PROCEDURE Detect ();
    WAIT ¬active;
    REPLY calls;
END Detect;
```

Algorithm 6.15 (Phase 2 – The Master)

```
ENTRY PROCEDURE Master-Phase2 ();
    CALL [S_1, ..., S_s].Evaluate ();
    CALL [S_1, ..., S_s].Expand ();
END Master-Phase2;
```

It is interesting to note that the problem of distributed evaluation of expressions arises in two other areas related to language implementation, namely parallel attribute evaluation [30, 120, 125, 126, 133, 134, 117] and parallel evaluation of functional programs [115]. There are, however, important distinctions between expression evaluation as it has been described in this section and the way it is treated in these two areas. The most important difference is the way expressions are assigned to the sites of the evaluating network. In our case precious little is known about the structure of the expressions prior to their evaluation, and hence the expression-to-site mapping has to be done more or less at random. In the case of parallel evaluation of attribute grammars and

functional programs, the expressions to be evaluated are often known *a priori*, and the goal of the research in these areas is to develop algorithms which compute optimal expression-to-site mappings.

Expression evaluation and graph traversals have also been studied for parallel random access machines; see for example Gibbons [84] and Er [70]. The distributed CET is also reminiscent of the *replicated blackboards* used in AI systems (Engelmore [69]).

6.8 Phase 3

The design of Phase 2 ensures that all global data, expression values as well as inline procedures, needed by a Slave during Phase 3 has been replicated and is locally accessible. Therefore, the design of Phase 3 becomes uncomplicated, in fact almost identical to its sequential counterpart. A further consequence is that during Phase 3 no communication between Slaves is necessary; the only communication occurs from the Slaves to the Master (to gain access to the global program location counter) and from the Slaves to the Servant (to send the generated code). In other words, a Slave will not be interrupted by any communication during Phase 3 other than that which it has initiated itself.

Algorithm 6.16 (Phase 3 – The Master)

1. On completion of Phase 2 the Master sends a multicast *DoPhase3 ()* call to all Slaves. The Slaves return from the call when they have completed Phase 3 processing.

2. On receipt of a *MachineCode (D)* call from a Slave S the Master returns the current value of the program counter and increments by D.

   ENTRY PROCEDURE MachineCode (D) → *int*;
   REPLY PC;
   INC (PC, D);
   END *MachineCode*;

Algorithm 6.17 (Phase 3 – The Slave) On receipt of a *DoPhase3 ()* call from the Master the Slave S starts processing the modules it owns, one at a time. Unknown values in the linear code are updated, calls to inline procedures are expanded, and machine code is generated. The Master is informed of the amount of code generated and the Slave receives the code's new program location. The code, its location, its relocation information, and the new addresses of generated procedures are passed on to the Servant, who writes the code to the executable file.

```
ENTRY PROCEDURE DoPhase3 ();
    FOR EACH module N owned by S DO
        Open N's object file;
        Read N's Deferred Code Section;
        FOR EACH referenced deferred procedure in N DO
            Fill in calculated expression values;
            Expand calls to inline procedures;
            IF some calls were expanded THEN
                Recompute basic blocks and next-use information;
            END;
            Optimize;
        END;
        C, Z, R ← Generate machine code C of size Z
            with relocation R for the deferred procedures;
        Close N's object file;
        PC ← CALL Master.MachineCode (Z);
        Update procedure addresses with the new PC;
        A ← Addresses of generated procedures;
        CALL Servant.BinaryCode (PC, C, R, A);
    END;
    REPLY;
END DoPhase3;
```

Once all Slaves have returned from the *DoPhase3 ()* call the Master will just have to wait for the Servant (using a *Close ()* call) to complete the writing and updating of the executable file.

6.9 Distributed Relocation

So far we have avoided describing the exact nature of the actions performed by the Servant. In fact, we have yet to produce a clear motivation for the existence of a Servant process. The reason for this lapse is that the issue of how the code produced by the Slaves is merged to form the resulting program file and how this file is updated during relocation is a very complex one, and one whose solution will have a profound impact on the efficiency of the distributed Paster. We will devote this section to examining three different solutions, in particular the one currently employed. We label the three solutions as follows:

temporary Each Slave stores the code it generates on a temporary file. At the end of the processing the files are concatenated to form the executable program file. Relocation can either be performed on the complete program file by one dedicated Slave (in which case that Slave has to be sent all

the relocation information and routine addresses generated by the other Slaves) or by each Slave on their own temporary file prior to concatenation. In the latter case the Slaves must share the addresses of the routines they have generated with *all* other Slaves.

shared The Slaves share ownership of the executable file, allowing them to write to it and update it concurrently. Relocation can either be performed by one dedicated Slave or by all Slaves concurrently.

servant The Slaves send the generated code to a special process (a Servant) which maintains the executable file. Along with the code, the Servant must also be sent the relocation information and routine addresses generated by the Slaves.

The *temporary* solution is attractive because of its simplicity. One drawback, however, is that the concatenation of the temporary files has to be performed sequentially.

The *shared* solution runs into quite different problems, since it gives Slaves shared access to a global file in a network file system. *Client file caching* is an important element of most distributed file systems which allows clients to store local copies of recently accessed file blocks. Changes to the file affect primarily the local copy, and the original file held by the server is updated according to some agreed-upon convention. When several clients concurrently modify the same file it becomes necessary to provide mechanisms that assure that each client holds a consistent set of file blocks. The NFS distributed file system uses file caching on both clients and servers and relies primarily on clients periodically checking the file modification date of remote files to protect against inconsistencies. For additional protection NFS provides *write-through* files, *un-cached* files, and a distributed file locking service. Write-through files force a client issuing a write-operation to wait until the server has written the data to the disk. The locking service consists of a network *lock daemon* with which clients can communicate in order to reserve parts of files for exclusive use. Programs such as the distributed Paster where several clients make frequent concurrent modifications to the same remote file cannot rely on modification date checks alone to protect against inconsistencies, since the checks are only done periodically (in NFS every three seconds) and there are several sources of time lags. A combination of un-cached files and file locking is sufficient, but only if the locking is done on the block level. In other words, blocks must only be modified using an atomic read-update-write sequence lest simultaneous modification of the same block should result in inconsistencies. This is a serious drawback if file blocks are large (in NFS they are 8k bytes) and the average

write operation only affects a small part of the block. In this case clients will frequently be waiting to obtain the lock of a particular file block.

The *servant* approach avoids any problem arising from multiple concurrent access to the executable file since only one process – the Servant itself – actually writes and updates the file. Neither is there any copying overhead, as is the case with the *temporary* approach. Furthermore, it is not difficult for the Servant to perform relocations concurrently with the production of code. Figure 6.6 gives a sketch of the Servant as it is currently implemented in the dPaster. The Servant consists of two concurrent lightweight processes, the *Writer* and the *Updater*. The *Writer* processes incoming code and data and writes it to the executable file. The *Updater* updates the executable file according to the procedure addresses and relocation information it receives. Naturally, the two processes must serialize access to the file-blocks in order not to write to the same part of the file at the same time. A simple and effective way to achieve serialization is for the *Writer* to lock all file-blocks until they have been completely written. When a file-block is complete the lock is turned over to the *Updater*, which can perform relocations in the block as soon as the relevant information is received.

6.10 Future Work

With the exception of the design of Phase 2, we have so far only examined one possible approach to distributed module binding. It is our intention to investigate possible improvements of the current implementation, as well as fundamentally different overall designs. In this section we will examine alternatives to the load balancing algorithm of Section 6.6.2 and present a distributed binder based on functional rather than data decomposition.

6.10.1 Load Balancing Revisited

In Section 6.6.2 we described the load balancing scheme used by the current dPaster implementation. This scheme, which assigns modules to slaves based on estimates of the amount of binding-time processing required per module, is attractive primarily because of its simplicity. The main problem with the method is that it is in general impossible during Phase 1 to accurately estimate the amount of work that a given module will impose on Phase 3. The reason is that it is not known until after Phase 2 which deferred procedures contain calls to inline procedures.[10] Not surprisingly, the empirical test results in Sec-

[10] Obviously, a procedure which makes many inline calls will be more expensive to process than one which does not; not only do we have to perform the actual expansions, but the calling

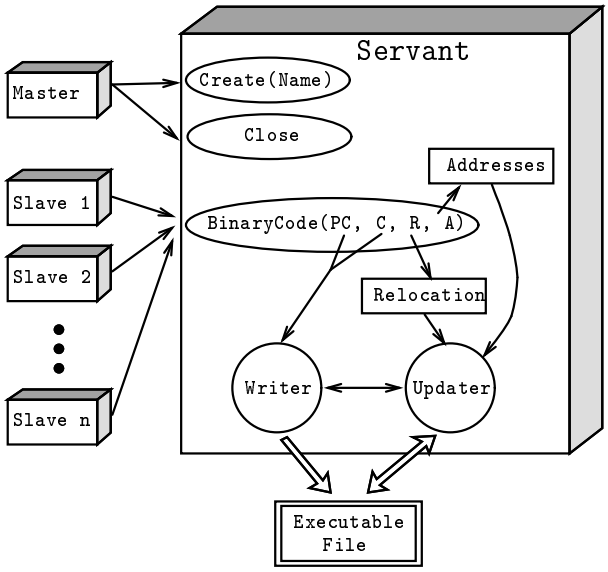


Figure 6.6: Schematic structure of the Servant. The *BinaryCode* calls received from the Slaves contain generated code and data (**C**), relocation information (**R**), and addresses of generated routines (**A**). The code is written to the executable file by a lightweight process *Writer*. **R** and **A** are stored in tables which are used by a lightweight process *Updater*, operating concurrently with *Writer*, to update the file.

tion 8.4.4 regarding the performance of the dPaster's Phase 3 indicate that the current load balancing algorithm works adequately for programs with a small amount of inlining. The algorithm performs less well, however, when there is a large number of inline calls and the number of calls differ significantly between modules. The most important future developments of the dPaster will therefore be the introduction of new and more accurate load balancing schemes.

Load balancing algorithms fall in two categories, *static* and *dynamic* [108]. Static algorithms are characterized by the fact that processing tasks remain at the processor to which they were initially assigned for the lifetime of the program. Dynamic algorithms allow tasks to migrate between processors whenever this improves efficiency. Since our current inadequate load balancing algorithm is static, it is natural to consider whether dynamic algorithms may fare bet-

procedure will be substantially longer after the integrations and therefore more expensive to process during code generation.

ter. There are several possible dynamic algorithms, but we will restrict our presentation to two methods which present themselves naturally.

The first method, which we will call *Global Scheduling*, dispenses with the static load balancing method altogether and gives the Master full dynamic control over the module-to-slave allocation. Under the Global Scheduling strategy the Master employs a similar approach for Phase 3 as it currently uses for Phase 1: a pool of hitherto unprocessed modules is maintained and used to assign modules to Slaves on an as-needed basis. In other words, when a Slave has finished a module's Phase 3 processing, a remote call is made back to the Master to retrieve a new module. Unlike the present static load balancing algorithm which only partially replicates CET values and inline procedures (on the assumption that a module will remain at the Slave to which it was originally assigned during Phase 1), the Global Scheduling method will require full replication of all global data.

The second method, *Module Migration*, extends the basic static algorithm with facilities to dynamically handle cases in which the original module-to-slave assignment is found to be unbalanced. The basic idea is to let the Slaves perform Phase 3 processing of the modules assigned to them without intervention, unless one or more Slaves finish long before the others. When that happens the Master, or the Slaves themselves, orchestrate a redistribution of the remaining unprocessed modules. In other words, during the first part of Phase 3 the Slaves work independently of each other and the Master, but towards the end of the phase they cooperate in order to allow modules to *migrate* from busy to idle Slaves. The advantage of this method compared to Global Scheduling is that when the original Phase 1 module distribution is accurate there will be no unwarranted extra communication. Furthermore, unlike the Global Scheduling method which requires *full replication* of global data, the Module Migration method only requires migrating modules to bring along local expression values and inline procedures in the move.

Determining which of these methods will prevail will be the subject of future investigations. We will be particularly interested in finding out whether the full data replication required by Global Scheduling and the local data migration required by Module Migration will cancel out the advantages gained from the more precise load balancing afforded by these methods.

6.10.2 Functional Decomposition Solution

The distributed Paster heretofore presented is modeled on parallelization by *data decomposition*. In this section we will examine an alternative approach, the *fPaster*, which in its design resembles the *functional decomposition* model.

The analogy with functional decomposition is not completely accurate since processes do not communicate solely through pipelines, but the fPaster does consist of a number of units with dedicated functionality. The fPaster division of labor is as follows (see also Figure 6.7):

Finder The Finder reads the Module Table of the object files and recursively finds all modules which are to take part in the bind. When a new module is found its name is passed on to the Allocator process and to one of the Data processes.

Data Reader The Data processes (max one per site) read the data and code segments of the modules received from the Finder and send them to the Writer.

Relocation Reader The Relocation Reader processes (max one per site) read the relocation segments of the modules received from the Finder and send them to the Updater.

Expression Reader The Expression Reader loads the constant expressions and sends them on to the Expression Evaluator.

Inline Reader The Inline Reader loads the inline procedures and sends them on to the Inline Expander.

Expression Evaluator The Expression Evaluator computes the values of the expressions and sends the result on to the Coders and the Inline Expander using multicast calls.

Inline Expander The Inline Expander performs the inline-in-inline expansions and sends the result to the Coders.

Allocator The Allocator receives the names of the modules from the Finder and information pertaining to the cost of code generation from the object file headers and from the Expression Evaluator. On the basis of this information the Allocator informs the Coders of which modules to process.

Coders The Coders (normally one per site) receive expression values and inline procedures from the Expression Evaluator and Inline Expander, and names of modules for which they should generate code from the Allocator. Generated code and relocation information is passed on to the Writer and Updater processes.

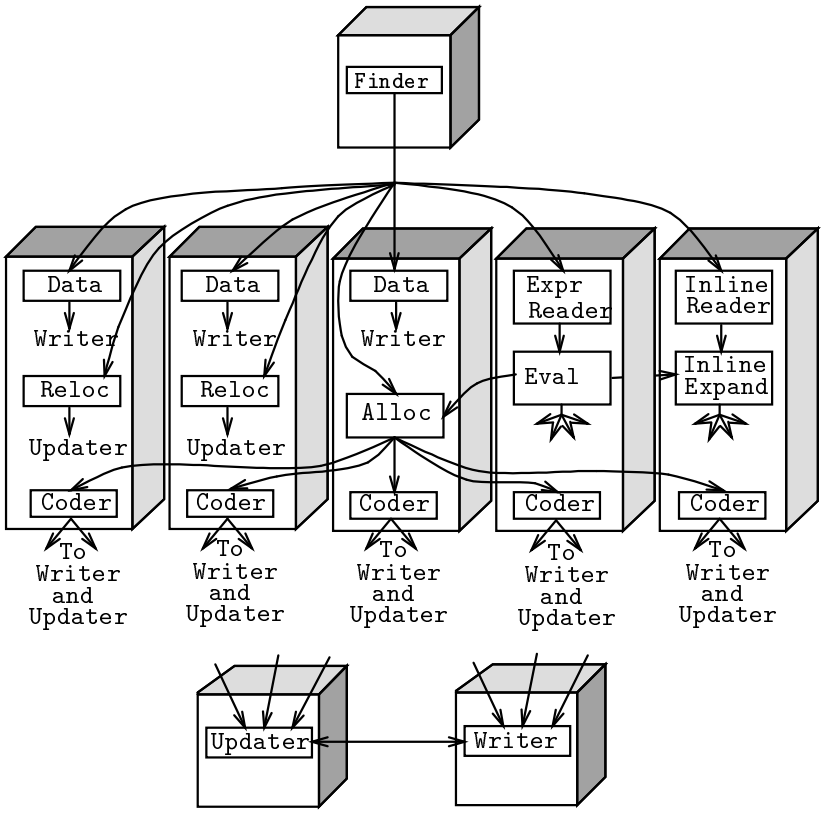


Figure 6.7: The structure of the fPaster.

Writer The Writer receives code from the Data Reader and Coder processes and writes it to the executable file.

Updater The Updater receives relocation information from the Relocation Reader and Coder processes and updates the executable file accordingly. File operations are synchronized with the Writer.

There are advantages as well as disadvantages to this configuration. On the minus side can be noted that the system contains more processes than the dPaster, and that, in contrast to the dPaster where each object file is opened only twice, the fPaster opens each file a total of seven times, albeit concurrently. On the plus side is increased concurrency and possibly faster expression evaluation and inline expansion.

6.11 Evaluation

Having given algorithms and arguments for the design of the dPaster, we will spend the rest of the chapter evaluating this design. We will first give the particulars of the current implementation and then attempt an analysis of the dPaster's message complexity.

6.11.1 Implementation Particulars

There are some discrepancies between the algorithms described earlier in the chapter and the actual implementation whose characteristics we will be discussing. Some algorithms have not been fully realized, some have suboptimal implementations, others have been somewhat optimized. This means that in some cases there is less message-passing in the implementation than might be anticipated from the algorithms; in other cases the implementation spawns more communication than would be necessary. The major sources of concern are outlined below:

- The distributed Paster currently does not check for cycles in the inline graph.
- The termination detection algorithm is controlled by the Master instead of Slave S_1. This leads to an extra message per phase, and possibly a few extra phases since in Algorithm 6.10 S_1 does not initiate a new round until it itself is passive, whereas the Master has to start a new round as soon as the previous one has failed to establish termination.
- From Algorithms 6.3 and 6.17 one would conclude that there is one *BinaryCode* call made to the Servant for each module processed. The Slaves are, however, able to save some communication by gathering enough data to fill a packet before a call is made.
- In Algorithm 6.8 the data sent between Slaves is in the form of sets of 4-tuples (m, i, a, V). However, a global renumbering scheme which gives each expression $e_{m,i}$.a a unique number j, allows Algorithm 6.8 to reduce the amount of transmitted data by sending 2-tuples (j, V) instead.
- In the present implementation of Phase 1 the Master initializes the module map with the main program module and the modules the main program module imports directly. This information is taken from the main program module's Module Table. This allows the Master to immediately get several of the Slaves started, rather than just one as in Algorithm 6.2.

While these discrepancies are minor, one aspect of the implementation which is likely to significantly influence the efficiency of the dPaster is the realization of the RPC protocol. The present implementation uses an extension of Edenbrandt's [68] UDP/IP implementation of Birrell and Nelson's [25] protocol. We give some of the limitations and performance characteristics of the protocol implementation:

- A remote call without parameters is about 80% slower than Sun's own remote procedure call facility,[11] and about 8 times slower than the corresponding low-level send and receive calls. The inferior performance can be attributed mainly to the fact that the `Unix` operating system does not have kernel support for lightweight processes.
- As in Birrell and Nelson's original implementation, large messages are sent as a sequence of packets where each packet is individually acknowledged by the server. This means that the distribution of large data items (such as inline procedures) will generate considerably more network traffic than necessary.
- Since the Internet protocol does not have a true multicasting facility, the present implementation realizes multicasting in terms of the Internet broadcast protocol.

It would be interesting to compare the current implementation of the dPaster with one using a simpler communication protocol or one running on an operating system with kernel support for lightweight processes. Such comparisons have not yet been possible.

6.11.2 Message Complexity

As a prelude to the empirical tests of Chapter 8 we will examine the proposed algorithms in terms of their message complexity. It must be noted that although message-passing accounts for a large proportion of the execution-time of many distributed programs, there are many other factors involved.

For the most part, the number of messages generated by the different phases is completely deterministic. The exception is Phase 2, which we will discuss shortly. The table below shows the number of messages generated by the different calls performed by the dPaster processes. Note that, for reasons discussed in Section 6.11.1, there may be fewer or more *BinaryCode* calls than indicated.

[11] Sun's RPC protocol does not support multi-packet calls and does not allow calls to take an arbitrary long time to complete. Furthermore, servers cannot handle several calls simultaneously.

Phase	From		To	Call	Messages
0	Master	$\Rightarrow$	Slave	FindSlaves	$1+s$
	Master	$\Rightarrow$	Slave	GlobalData	$1+s$
	Master	$\Rightarrow$	Servant	Create	2
1	Master	$\Rightarrow$	Slave	ProcessModule	$2m$
	Slave	$\Rightarrow$	Master	ModuleData	$2m$
	Slave	$\Rightarrow$	Servant	BinaryCode	$2m$
2	Master	$\Rightarrow$	Slave	Evaluate	$1+s$
	Slave	$\Rightarrow$	Slave	IsInline	$s(s+1)$
	Slave	$\Rightarrow$	Slave	Values	
	S_1	$\Rightarrow$	$S_2, \ldots, S_s$	Detect	
	Master	$\Rightarrow$	Slave	Expand	$1+s$
	Slave	$\Rightarrow$	Slave	InlineProc	
3	Master	$\Rightarrow$	Slave	DoPhase3	$1+s$
	Slave	$\Rightarrow$	Master	MachineCode	$2m$
	Slave	$\Rightarrow$	Servant	BinaryCode	$2m$
4	Master	$\Rightarrow$	Servant	Close	2

Phase 2

The analysis of the message complexity of Algorithm 6.14 is far from trivial. There are three major sources of trouble: First of all, the structure of the expression DAG is determined by the relationships between the hidden objects in the source program. For example, a program whose hidden types are largely independent of each other will exhibit a "flat" expression DAG, whereas a program where hidden types are often implemented in terms of each other will yield a deeper and more complex DAG. Secondly, the distributed expression DAG is built more or less at random; the load balancing algorithm of Section 6.6.2, which is responsible for the assignment of modules to Slaves, takes into account only the *size* of each module's expression DAG, not its structure. Thirdly, Algorithm 6.14 is asynchronous; each Slave executes its own compute-send cycle independently of the other Slaves. In other words, it is not possible to determine the number of values a Slave can compute during each round of the cycle since this ultimately depends on the speed with which the other Slaves are executing and the speed of message-passing.

So, clearly, in order to be able to say anything about the message complexity of Algorithm 6.14 it is necessary to make some simplifications. We will assume a synchronous version of Algorithm 6.14 (called Algorithm 6.14') where the compute-send cycle has been replaced by a compute-send-wait cycle, such that a Slave does not start a new computation round until it has received a *Values*-message from the other Slaves. This presupposes that a Slave always sends a *Values*-message even if it has not computed any new values. We will also

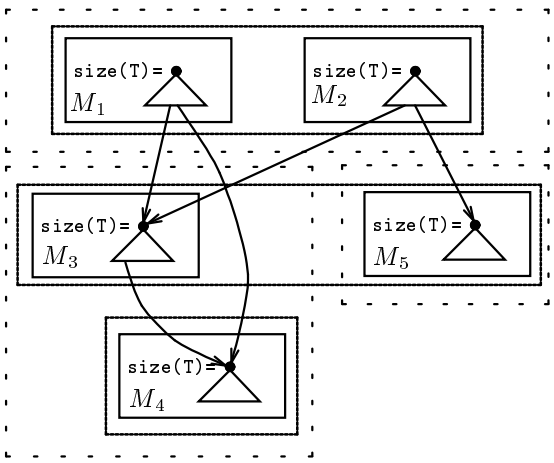


Figure 6.8: Type dependencies for a set of 5 modules

assume that the assignment of modules to Slaves is done solely to balance the number of expressions per Slave, i.e. $cost(m, S) = m.e$.

Furthermore, we will restrict ourselves to considering only the graphs generated by the translation of hidden types. Let us consider a set of modules $M_1, \ldots, M_m$ where each module implements one hidden type T. Each type $M_i`T$ is implemented in terms of a subset of the other types, subject to the restriction that recursive types are disallowed. The relationships between the types are reflected in the expression graph through the dependencies between the nodes which represent the size of the $M_i`T$'s. Figure 6.8 shows an expression graph generated from 5 modules, where each (possibly complex) expression is represented by a triangle ($\triangle$). We will term such a schematic expression graph a *dependency graph*.

The number of compute-send-wait rounds which Algorithm 6.14' will have to make depends on two factors: The height of the dependency graph,[12] and the module to Slave assignment. If, for example, the 5 modules in Figure 6.8 are grouped on 3 Slaves as indicated by the dotted boxes, Algorithm 6.14' will make 3 compute-cycles, whereas if the grouping is done according to the dashed boxes 2 cycles will suffice.

We see that the number of compute-cycles performed by Algorithm 6.14'

[12]The *height* of a directed acyclic graph is the length of the longest directed path in the graph (see McKay [147]). Note that this is different from the *diameter* which is the length of the longest *geodesic*, the shortest path between any two nodes.

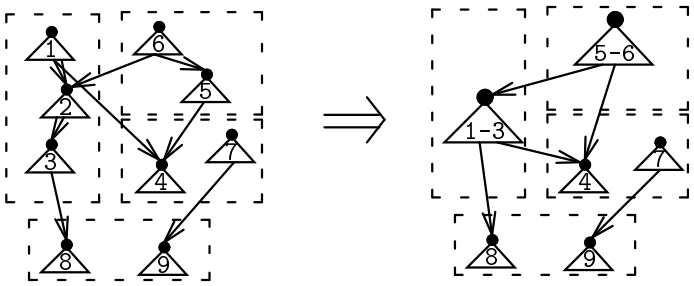


Figure 6.9: Connected subgraphs residing on the same Slave are coalesced into supernodes.

is given by the height of the "supergraph" which results when all connected subgraphs made up of nodes residing on the same Slave are replaced by a supernode (see Figure 6.9). Let $\mathcal{H}$ be the height of the original expression graph, and assume (1) that the nodes of the original graph are evenly distributed among the Slaves, and (2) that the probability of any two nodes being connected is constant. Then the probability of a given edge connecting two nodes on the same site is $1/s$ and the average height of the supergraph $\mathcal{H}' = (1 - 1/s)\mathcal{H}$. In the worst case, of course, $\mathcal{H}' = \mathcal{H}$, corresponding to the case when no two adjacent nodes reside on the same Slave.

In Algorithm 6.14' each round of the compute-cycle generates s^2 messages (s Slaves each multicast 1 message and receive $s - 1$ replies) and therefore the total number of messages generated is[13]

$$s^2\mathcal{H}' = s^2(1 - 1/s)\mathcal{H} = (s^2 - s)\mathcal{H}$$

We must now ask ourselves what kind of dependency graphs we are likely to encounter, and what the height of these graphs might be. A dependency graph of height zero corresponds to the case when all hidden types are implemented in terms of primitive types, whereas a $\mathcal{H}' = n$ implies that the n types depend on each other in a sequential fashion and that no adjacent nodes in the dependency graph reside on the same Slave. While the height of a *random DAG* is approximately $0.764334n$ (McKay [147]), and the height of a *random binary tree* is $\mathcal{O}(\sqrt{n})$ (Vitter and Flajolet [212]), our intuition tells us (and this is supported by the – admittedly meager – empirical results of Section 8.3.3) that none of these types of average graphs are very likely candidates for actual

[13]The fact that the message size is bounded has not been accounted for.

dependency graphs. Rather, we suspect that the height of most graphs will be bounded by a small constant, which would tend to make Algorithm 6.14 run in a small number of rounds. This conjecture is corroborated by the sample runs reported in Section 8.4.3.

The number of control messages used by the termination detection algorithm is dependent on the number of times the sites carrying out basic computation enter a passive state. Let us consider a simpler version of the algorithm (called Algorithm 6.10') where the detection of termination is carried out by the Master rather than by one of the Slaves. Furthermore, let P_i be the number of times site S_i goes passive. Then, in the worst case, Algorithm 6.10' runs in $\min_{i=1}^{s} P_i$ passes[14] using a total of $(s+1)\min_{i=1}^{s} P_i$ messages. If we also use the synchronized evaluation algorithm we see that the Slaves will go passive $1+\mathcal{H}'$ times, yielding a total of

$$(s+1)(1+\mathcal{H}') = (s+1)(1+(1-1/s)\mathcal{H}) = \mathcal{H}s(1+1/\mathcal{H}+1/s^2)+1$$

control messages for the termination detection algorithm.

To conclude this discussion we will calculate the number of messages generated by the alternative expression evaluation algorithms. The **sequential** algorithm runs in two phases. First, sites $S_2, \ldots, S_s$ send their expressions to S_1. This requires

$$2(s-1)\lceil n/s/\alpha_e \rceil$$

packets, since each site has $\lceil n/s \rceil$ expressions, and α_e expressions can be sent in each packet. In the second phase, S_1 broadcasts the results to sites $S_2, \ldots, S_s$ using

$$s\lceil n/\alpha_v \rceil$$

packets. Adding up we get

$$2(s-1)\lceil n/s/\alpha_e \rceil + s\lceil n/\alpha_v \rceil.$$

In the **redundant** algorithm each site broadcasts its expressions to the other $s-1$ sites. The total number of messages sent will be

$$s^2\lceil n/s/\alpha_e \rceil.$$

The number of messages used by the **depth-first** algorithm is twice the number of external edges in the graph. The probability that an edge is external is $(1-1/s)$, and the number of edges is dn, which gives a total of

$$2(1-1/s)dn$$

[14]The Master does not start a new round of termination detection until all Slaves have reported in. Therefore the number of rounds is determined by the Slave which enters a passive state the least number of times.

messages, assuming that we do not have to worry about possible cycles.

It must again be noted that we are only taking into account the *number* of messages, not their *size*. Depending on the implementation of the message-passing system, the cost of sending a large message may be virtually the same as sending a small one, or substantially higher. Further, we are assuming that broadcasts and multicasts are implemented as one outgoing message and one return message per recipient, and not simply as repeated simple calls. We must also keep in mind that message-passing only represents part of the total time used in evaluating the expressions. In algorithms 6.4 and 6.5 the expressions are evaluated sequentially, whereas in the other algorithms evaluation proceeds in parallel on the different sites.

6.12 Summary

There have been several attempts at parallelizing the assembly and compilation stages of translation, but until now there have been no attempts at parallel linking, perhaps because link-editors are considered to be inherently sequential and system-specific. We believe that the design presented in this chapter, together with the empirical results of Chapter 8, will make it clear that distributed module binding is indeed feasible.

Although the dPaster was designed specifically to provide efficient support for the flexible abstraction primitives of Zuse, we also see it as a suitable framework for the implementation of expensive inter-modular optimizations. At the present time the only such optimizations[15] performed by the dPaster is inline expansion and the removal of unreachable procedures. In future versions we intend to include other optimizations such as global register allocation [49] and global constant propagation [38]. We believe that it is possible to modify the CET and the distributed CET evaluation algorithms presented in this chapter, to efficiently compute the global data flow graph [86] and other kinds of inter-modular data which will be necessary in order to perform such inter-modular optimizations.

[15] See, for example, Richardson [176] for a discussion of various inter-procedural optimization techniques.

Chapter 7

Incremental High-Level Module Binding

Perfection is the enemy of the dissertation.
Ivan Sutherland

abstraction inversion: *Layering inappropriate amounts of complex interfaces to obscure a simple system.*
Reported by Alex Blakemore

7.1 Introduction

This chapter will examine the use of incremental techniques as a further avenue towards fast high-level binding. While the sequential techniques of Chapter 5 will be sufficient for small programs, the hierarchical techniques (Section 5.11) will be useful when some parts of a program are rarely changed, and the distributed techniques in the previous chapter promise fast turn-around times for major changes to large programs, they are likely to be outperformed by the incremental techniques presented here when only small and local changes have been made between two consecutive binds.

7.2 Incremental Translation Techniques

Most translators for languages in the Algol family are *batch-oriented*; they are essentially mathematical functions which take a number of arguments as input and produce a result as output. The input to the sequential binder of Chapter 5, for example, is the contents of the object files of the bound program's

modules, and the output is an executable file. An important characteristic of such batch-oriented translators is that they are *history-less*; they do not retain any information between successive invocations. This can be seen as an attractive feature since it greatly simplifies the design of the translator, but it fails to capitalize on the observation that between two consecutive invocations of a translation tool, it is often the case that most input arguments and hence the major part of the output value remain virtually unchanged.

A different class of translation systems has emerged over the last decade which, rather than recomputing the output value from scratch for every invocation, retains the result of the previous invocation (and any intermediate data used in calculating this value) and uses it as a basis for the new calculation. Such techniques are termed *incremental*, since they perform changes to the output value in small steps. An incremental semantic analyzer, for example, may store the attributed abstract syntax tree of a module, and when the module's source code is changed only recalculate the affected attributes.

It is important to note that translation tasks can benefit to a greater or lesser degree from incremental techniques. The most important factors to consider may be summarized in the following questions:

- Are the most frequent kinds of changes to the input arguments likely to affect a small or a large part of the output value?
- What amount of state information needs to be stored between invocations in order to make the incremental updates efficient?
- How does one determine the parts of the output value which need to be recalculated after a particular change to the input arguments, and how expensive is this analysis?

It is obviously the case that there exist situations in which it is faster to evaluate a function value from scratch rather than incrementally, even for very slight changes to the input arguments. It should furthermore be clear that if a function needs to store exorbitant amounts of state information between invocations, incremental updating may become infeasible. Yellin's INC [237] language for incremental computations addresses these points in a systematic way by attaching to each construct the computational cost of *static*[1] and *incremental* evaluation. This scheme makes it possible to predict when a particular argument change should result in an incremental recomputation and when evaluation from scratch is preferable.

[1] Evaluation from scratch.

Static semantic analysis can serve as an example of a translation task which will often benefit from incremental evaluation. In most block-structured languages a change to a declaration on a particular level will often only affect the semantic correctness of uses of the declaration (Hedin [92]), which can only occur in blocks on the same and lower levels. Similarly, a change to a statement will often only affect the static correctness of the statement itself. If changes on lower levels are more frequent than global changes, incremental updating will almost always be faster than exhaustive reevaluation.

Inter-procedural optimization and code generation, on the other hand, are examples of translation tasks which do not exhibit this kind of "local-effect-to-local-change" behavior. On the contrary, the work by Cooper [58], Pollock [168, 169], and Bivens [26] indicates that it is a non-trivial and potentially expensive task to determine which parts of a globally optimized program are affected by editing changes. One can imagine a worst-case scenario in which an edit to a single statement would invalidate all the collected global optimization information. This might, for example, occur in an incremental version of Wall's [214, 216, 217] inter-procedural register allocation algorithm.

As a general rule we might thus say that if small changes to a function's arguments result in major changes to its result, then incremental techniques are unlikely to be beneficial. We also note the importance of not having to store an unreasonable amount of information between successive invocations and of being able to easily determine which parts of a computed value will be affected by a change to the arguments. If computing the affected parts is very complicated and expensive, then these computations may outweigh the benefits of incremental updating.

7.2.1 Incremental Translation Tools and Environments

Translation tools which make use of incremental techniques are often, but not always, part of integrated programming environments. Such environments encourage an interactive, exploratory style of programming, which through the close coupling between different incremental tools (structure editors, semantic analyzers, code generators, optimizers, linkers, and loaders) can achieve fast turn-around times. Examples include the R^n [44, 57] FORTRAN programming environment, *Magpie* [191], Kaplan and Kaiser's [120, 119] incremental distributed environment, the *Rational* [222, 12, 71] Ada environment, *Gandalf* [161], and the *Synthesizer Generator* [175]. We refer to Dart [62] for a more detailed overview.

The incremental systems which are of most interest to the work we will be presenting in the next section are R^n, which supports inter-procedural opti-

mizations, and *Inclink* [171], which does incremental linking. We will therefore examine these more closely.

Inclink is based on a study by Linton and Quong [139, 172] which concludes that in a tool-based programming environment (such as `Unix`), on the average only one or two modules have been changed and recompiled between two successive links. They also note that the sizes of modules increase very slowly between links. The first time *Inclink* is called it performs a regular link-edit session, after which it enters a dormant state. When it is later awakened for a new linking session, it first rereads and inserts the object code files of any changed modules in-place in the old executable. Thereafter, it relocates any new or modified references. In order to accommodate future increases in module-size, some extra space (called the *slop*) is allocated after the code and data segments of each module in the executable program. If a changed module overflows its allocated area, all subsequent modules are shifted towards the end of the file. Hence, if there are no overflows *Inclink* runs in time proportional to the size of the changed modules; otherwise it runs in time proportional to the size of the resulting program. Quong reports that for typical relinks *Inclink* runs 3 to 70 times faster than the UNIX batch linker, *ld*, while consuming 6 to 12 times as much memory (five megabytes for a 17000 line program divided into 26 modules and using 9 libraries). Part of the speedup can be attributed to the fact that *Inclink* keeps symbols, relocation information, and object code in memory between links, which obviates the need to reread the object code of unchanged modules.

R^n is an integrated, incremental programming environment dedicated to scientific programming in FORTRAN. The environment consists of a structure-oriented *module editor*, a *program compiler*, a *module compiler*, and an *execution monitor*. The program compiler is responsible for computing the inter-procedural information and passing it on to the module compiler. The module compiler applies traditional code generation and optimization techniques (aided by the collected inter-procedural information) to a single module or a collection of modules. The inter-procedural optimizations performed include inline expansion [55, 56] and constant propagation [38], but the gathered inter-procedural information also helps traditional optimization techniques such as common subexpression elimination. The module editor aids the compilation process by building an abstract syntax tree for each compiled module and collecting local data-flow information. The module editor also informs the program compiler of which changes the user has made to the program, and based on this information the program compiler can determine which inter-procedural information needs to be (incrementally) updated and which modules need to be recompiled. One complication with the R^n approach to inter-procedural op-

timization is that a module's object code becomes dependent on the program in which it is included. In other words, if a module is to be a part of several different programs, each program must have its own specialized version of the module's object code.

7.3 The Incremental Paster

In this chapter we will describe an incremental Paster, called an *iPaster*, which works along the same lines as *Inclink*. There is, however, a fundamental reason why incremental pasting will be considerably more difficult (and potentially less profitable) than plain incremental linking: the inter-module data handled by the iPaster is much more complicated than that handled by *Inclink*. *Inclink* works with one kind of inter-module data only: the addresses of the routines and data items exported by each module. When the address of a routine changes, *Inclink* will simply overwrite all references to the routine with the new address. The iPaster, on the other hand, will have to accommodate not only changes to the addresses of compiled routines, but also changes to the values of expressions and the code of inline procedures. We give two examples:

1. A change to the realization of an inline procedure P will make it necessary for the iPaster to retrieve the unexpanded code of all inline procedures which call P (and any which call them, etc.) and redo all the expansions. All Phase 3 processing of every procedure which calls any of the affected inline procedures will have to be redone. If the new version of P is larger than the old one, the size of the code of the deferred procedures is likely to increase.

2. A change to the realization of an abstract type T which affects its size may have far-reaching consequences. All procedures which reference a variable of type T, or a variable whose type depends on T, will have to be reprocessed. If one of these procedures happens to be an inline procedure, we have a situation like the one described above.

Our intention, therefore, is for the iPaster to be employed primarily when there has been a small number of local changes to the modules which make up the program, and to defer major updates to the dPaster.

7.3.1 Preliminaries

Similarly to *Inclink*, the iPaster will keep note of the state of the bound program between sessions. The saved information will include: the names of the modules

which make up the program, the location of their object files, and the addresses in the executable of their code and data segments; the expressions and their values; the inline procedures in their expanded as well as unexpanded form; the address and relocation information of each deferred procedure; and two *use-maps* which indicate the CET expressions and the deferred and inline procedures which will be affected by changes to individual expressions. Contrary to *Inclink*, which keeps the module's object code in memory between links, the iPaster will not keep the Binary Code, Data, and Deferred Code Sections in memory since we believe this will be prohibitive for large programs.

The iPaster will maintain the four phases of the sPaster with some modifications. Phase 1 will load the expressions and inline procedures of the changed modules and update their binary code and data segments in the executable. Phase 2 will recalculate the values of affected expressions and expand inline procedures as necessary. Phase 3 will use the call-graph and the *use-maps* to determine the deferred procedures which have been affected, and will read and reprocess them. To allow for the growth of individual deferred procedures, *slop* will have to be appended to the code of each procedure. Slop-management will be discussed further in Section 7.3.3.

The incremental nature of the iPaster makes it well suited to situations in which a program has undergone a small number of local changes. The algorithms presented below will be able to handle changes with global consequences as well, but to make the iPaster design simple and efficient we institute three restrictions. Between two consecutive invocations of the iPaster:

- no new modules may be added to the program.
- no modules may be deleted from the program.
- no module specification units may be changed and recompiled.

The last point is essential, since it guarantees that the cross-module references (which all go through the CET) do not change between runs of the iPaster. For example, we can be assured that between two consecutive calls to the iPaster, $e_{m,i}$.size will always refer to the same type. A consequence of the rules above is that the iPaster is restricted to changes to the realization of abstract items and the hidden part of semi-abstract items, i.e. changes which occur in the implementation unit. We do not see this as unduly restrictive, since we believe that these are the kinds of changes which occur most frequently.

We have already discussed the major advantages that *Inclink* has over the iPaster, but we have yet to mention the one aspect in which the iPaster has the upper hand: while *Inclink*, like other link-editors, has to cater to the needs of many different compilers, the iPaster works in symbiosis with one compiler (zc)

and can demand that it do some extra work. Determining the changes which have been made to a module since the last bind is an important and time-consuming task for the iPaster, but it can be made more efficient with some help from the compiler. We therefore add an integer CET attribute *checksum* to all expressions associated with deferred and inline procedures and make the compiler compute it in such a way that it depends on every part of the corresponding Z-code block. To determine if a procedure has been altered since the last bind the iPaster simply checks if the old and the new *checksum*-attributes differ. In a similar vein, we give each module M the CET expressions $e_{M,0}$.b_checksum, $e_{M,0}$.d_checksum, and $e_{M,0}$.r_checksum, which are checksums for the Binary Code, Data, and Relocation Sections, respectively.

Lastly, we adopt the notion of a "dirty" expression, procedure, and module. An entity is said to be dirty if it needs to be reprocessed during an invocation of the iPaster, or if (in the case of modules) it contains an object which needs to be reprocessed. For convenience we give each dirty CET expression the marker "$*$" (as in $e^*_{m,i}$.a).

7.3.2 The Phases of the iPaster

We are now ready to give the iPaster algorithms. The main program (Algorithm 7.1) starts with a regular, sequential bind by using the algorithms developed for the sPaster. It then enters a dormant state, and when awakened prompts the user for modules which have changed since the last run. These modules are processed by the iPaster's own Phases 1 to 4 to build a new version of the executable file, after which the iPaster reenters its dormant state and waits for the process to repeat.

In the algorithms below we will make use of two maps E and P which store the expressions and procedures which will be affected by changes to individual CET expressions. The maps are built as part of Phases 2 and 3 of the first sequential bind and are kept up-to-date during the subsequent incremental updates.

Algorithm 7.1 (iPaster)

1. Let E be a map from CET expressions to sets of expressions such that if $(e_{m,i}.\mathrm{a} \mapsto \{\cdots, e_{n,j}.\mathrm{b}, \cdots\}) \in E$ then $e_{n,j}.\mathrm{b} \triangleq f(\cdots, e_{m,i}.\mathrm{a}, \cdots)$ is a CET expression.

2. Let P be a map from expressions to sets of procedures such that if $(e_{m,i}.\mathrm{a} \mapsto \{\cdots, e_{n,j}, \cdots\}) \in P$ then $e_{m,i}.\mathrm{a}$ is referenced in the Z-code block of the procedure associated with expression $e_{n,j}$.

3. Perform the sPaster's Phase 1 (Algorithm 5.8), but add slop to the Data and Binary Code Sections.

4. Perform the sPaster's Phase 2 (Algorithm 5.9), but add slop to every generated variable, structured constant, and object template. Also construct the E-map from the CET and the P-map from the inline procedures.

5. Perform the sPaster's Phase 3 (Algorithm 5.10), but add slop to the code of every deferred procedure. Also construct the P-map from the deferred procedures.

6. Perform the sPaster's Phase 4.

7. Execute the following code:

```
LOOP
    Perform iPaster's Phase 1 (Algorithm 7.2);
    Perform iPaster's Phase 2 (Algorithm 7.3);
    Perform iPaster's Phase 3 (Algorithm 7.4);
    Perform iPaster's Phase 4;
END;
```

The Incremental Phase 1

One of the iPaster's most important Phase 1 tasks is to detect the modules which have changed since the last run. Quong [172] notes that this is difficult to do efficiently, especially in a network file system, and that the obvious method (comparing the time-stamp of each module's object file to the last time of modification) is much too inefficient. We therefore opt for a solution similar to Quong's, namely to leave it to the user to inform the binder of the modules which have changed.

Algorithm 7.2 (Phase 1)

NewModules ← *Prompt the user for changed modules*;
FOR EACH $n \in$ *NewModules* **DO**
  Reopen n's object file and load the preamble;
  Reload n's Expression Section*;*
  Mark all $e_{n,i}$.a *whose definition is different from* old-$e_{n,i}$.a *dirty;*
  FOR ALL i **SUCH THAT** ($e_{n,i}$.inline $\triangleq \mathbb{T}$) $\wedge$
    (old-$e_{n,i}$.checksum$\neq e_{n,i}$.checksum) **DO**
    Load inline procedure $e_{n,i}$ *from n's* Inline Section*;*
    Mark $e_{n,i}$.inline *dirty;*
  END;
  Update the E-map from the CET, and the P-map from the inline procedures;
  IF (old-$e_{n,0}$.d_checksum$\neq e_{n,0}$.d_checksum) **THEN**
    If n's Data Section *fits in its old spot in the executable file,*
      insert it in-place. Otherwise append it to the end of the file;
  END;
  Same as above, but for n's Binary Code Section*;*
  Same as above, but for n's Relocation Section*;*
  Close n's object file;
END;

When a new Data or Binary Code Section overflows its slop region, it is necessary either to shift the following code to make room for the new section or (as in Algorithm 7.2) to append the section to the end of the executable file. Quong rejects the latter method on the grounds that it makes it impossible to separate the code and data segments in order to make the code write-protected. While this is a valid objection we will ignore this complication in this presentation.[2]

The Incremental Phase 2

We are now ready for the incremental Phase 2, which is very similar to its sequential counterpart:

Algorithm 7.3 (Phase 2)

1. Evaluate every "dirty" CET expression $e^*_{m,i}$.a, as well as any expression which, according to the *E*-map, (transitively) depends on a dirty expression.

[2]The current paster implementations produce the obsolete `Unix` *impure* format which mixes code and data, and hence appending to the end of the executable file causes no problem.

2. Rewrite any changed structured constant data objects (such as object type templates and literal records, strings, sets, or arrays) to their original spots on the executable file, or to the end of the file if they do not fit.

3. Update any changed relocation information.

4. Fill in newly calculated expression values from the CET in the Z-code of any – according to the P-map – dirty inline procedure.

5. Redo inline expansions as necessary according to the E-map.

The only part of Algorithm 7.3 which may need further explanation is the incremental inline expansion. The main question is whether we keep the results of intermediate expansions between consecutive runs of the iPaster, or whether we only store the original inline routines and the final results. In the former case (which is memory expensive) we only have to redo the expansion of dirty procedures, while in the latter case all expansions in all procedures which call a dirty procedure have to be redone.

While there exist a variety of incremental algorithms for keeping a set of expressions (and the values they compute) up-to-date in response to changes (see, for example, Alpern [8], Cohen [51], and Yellin [236]), they are overly general for this application. In particular, it would be unnecessarily expensive for the iPaster to keep the values up-to-date after each new expression is read and inserted in the CET during Phase 1. Rather, it is more efficient first to perform all necessary changes, and then to recompute any affected values.

The Incremental Phases 3 and 4

As we have already mentioned, for the sake of memory conservation the saved state of the iPaster does not include deferred procedures. Therefore it becomes necessary to reread (part of) the Deferred Code Section of dirty modules; i.e. modules which contain one or more dirty, referenced, deferred procedures. The iPaster's Phase 3 is otherwise similar to that of the sPaster:

Algorithm 7.4 (Phase 3)

PROCEDURE *Phase3* ();
  FOR EACH for every dirty module n **DO**
    Reopen n's object file;
    FOR ALL i **SUCH THAT** $e_{n,i}$ *is a dirty, referenced procedure* **DO**
      Read $e_{n,i}$ *from n's deferred code section;*
      Fill in newly calculated expression values from the CET;
      Expand calls to inline procedures;

```
            IF some calls were expanded THEN
               Recompute basic blocks and next-use information;
            END;
            Optimize;
            Generate machine code;
            Store generated relocation information;
            Write the code to the executable file at the old position
              or append to the end of the file;
         END;
         Close n's object file;
      END;
   END Phase3;
```

The iPaster's Phase 4 is also similar to the sPaster's, only there is no need to update addresses which have not changed during the present invocation.

7.3.3 Slop Management

Quong examined and evaluated a number of sophisticated slop management strategies, but settled for a simple algorithm which allocates 100 bytes of slop to each data and code segment initially and after each overflow. Since the iPaster's unit of incrementality is the procedure rather than a module segment as in the case of *Inclink*, we believe that in our case a slightly more complex strategy will be necessary. We propose that the slop assigned to a procedure P the n^{th} time it overflows its allocated area be given by:

$$\texttt{slop}_n^P = \begin{cases} C + \sum_{i=1}^{i=k} \texttt{slop}_0^{Q_i} & \text{if } n = 0 \\ \\ C + \texttt{slop}_{n-1}^P & \text{if } n > 0 \end{cases}$$

where $< Q_1, \ldots, Q_k >$ are the inline procedures called by P. In other words, the amount of slop initially given to a procedure is the sum of the slop given to the inline procedures it calls and a small constant C. Each time P overflows its allocation its slop is increased by a small constant. The reasoning behind this scheme is of course that the size of the binary code of a deferred procedure is more likely to grow quickly if the procedure depends (directly or indirectly) on a large number of inline procedures than if it only calls non-inline procedures.

7.4 Summary

Contrary to Quong's *Inclink*, which under almost all reasonable circumstances is faster than its batch counterpart *ld*, we conjecture that the iPaster will not

always outperform the sPaster and the dPaster. The reason is, of course, that for a language like Zuse with flexible encapsulation facilities and a translating system like zts which emphasizes global optimizations, it will often be the case that a seemingly small and local change (such as adding a field to the hidden part of an extensible record type, or changing the realization of an abstract inline procedure) will have far-reaching global consequences. We envision a system which, based on an analysis of the changes made to the program, invokes either the iPaster (if the changes are small and localized) or the dPaster (if the changes have global effect). Such as system would have the potential to be fast regardless of the kinds of changes which are made between binds.

A possibility which has yet to be discussed is to instrument the dPaster with incremental capabilities to form a ${}^{\mathrm{d}}_{\mathrm{i}}$Paster: a distributed incremental paster. The question is, of course, whether there exist any circumstances under which such a contraption would be likely to outperform both the dPaster and the iPaster. We believe, at this point, that the answer is "no." Distributed applications, in general, need to work on large data sets in order to make up for their high start-up costs. Incremental applications, on the other hand, perform best when given a small amount of data to process. It would therefore seem that the combination of distributed and incremental high-level binding is unlikely to be profitable.

The iPaster differs from other incremental translators for modular languages which engage in inter-modular optimization in that it hampers neither modularity nor separate compilation. A Zuse module can easily be part of several programs at the same time without having to be recompiled, in contrast to the R^n system, in which each program needs its own compiled version of each FORTRAN module.

Chapter 8

Evaluation

> **2 + 2 = 5–ISM:** *Caving in to a target marketing strategy aimed at oneself after holding out for a long period of time.* "Oh, all right, I'll buy your stupid cola. Now leave me alone."
>
> *Douglas Coupland [60]*

8.1 Introduction

The following statement summarizes some of the objections which may be raised against the translating system designs presented in the previous chapters:

> The benefits afforded by ZTS over traditional translating systems (such as improved control over encapsulation, improved global code quality, and the reduced need for recompilations) are limited, and do not warrant either the addition of another (complicated) piece of software or the extra link-time cost.

We have already noted that the benefits of flexible encapsulation can only be conclusively determined by putting it to the test in a wide variety of programming situations. There is still some controversy over the potential profits of inter-modular optimizations (see Chow [48], Cooper [55, 56], and Richardson [176, 178]), and one might argue that other methods for reducing the number of unnecessary recompilations (see Cooper [58], Tichy [209], Pollock [168], and Schwanke [190]) may be more effective than the one presented here. In this chapter we will put these and all other considerations aside and concentrate on

the last – and in our mind the most serious – objection: that ZTS-style high-level binding is inherently slow and will be unsuitable when applied to large programs.

This chapter will thus be concerned with determining the cost of pasting when applied to programs of different characteristics. In Section 8.2 we develop a model of modular programs which allows us to generate a large family of ZUSE programs differing in size, structure, and degree of encapsulation. In Section 8.3 we describe the experimental methods used in Section 8.4 where we study the behavior of ZTS when applied to a variety of programs generated by the model. We also observe its behavior when applied to some real programs. The chapter concludes with a summary (Section 8.5).

8.2 A Model of Modular Programs

There are several different ways of evaluating the efficiency (the time and space used) and effectiveness (the speed and size of the produced code) of translation systems for a language:

1. Run each translation system on a representative collection of well-known programs in the language and measure the resources used by the translator and the quality of the code it has produced. The programs should ideally be of varying size, from different application domains, and of different authorship.

2. Collect a representative set of programs and use this set to construct a model which adequately describes salient aspects of programs in the language. This model might include both quantitative and qualitative measures. Run each translation system on a set of programs constructed so that their statistical properties reflect those suggested by the model, and evaluate the results as in the previous case.

3. In the absence of a large collection of real programs, one might still be able to construct a model such as the one above. Such a model will have to be based on the assumption that the language lends itself to a certain style of programming, and that this style will be the most prevalent.

The first approach is the one most commonly employed, and for many languages there exist well-known sets of programs which have been used to test the translation times and code quality of a variety of compilers. The advantage of the second method, however, is that one is not restricted to testing only the programs at hand, but can generate as many programs as is deemed necessary.

The second method would make it possible, for example, to test the translator's behavior with the growth of program size. This is done by generating programs with statistical properties similar to the ones in the original set, but which are much larger than any of these programs.

The first method assumes the existence of an adequate sample of "real" programs, while the second method requires, at the very least, a body of knowledge of the structure of such programs. Unfortunately, relatively little is known about the statistical properties of programs: Knuth [128] studied FORTRAN programs, Wanvik [54] presents the results of a study of a fairly small CHILL program, Berry [23] contains some rather trivial statistics of C programs, Borison [33] gives quantitative data on three C and three Ada programs, and Linton and Quong [139] studied changes in module and program size over successive compiles and links.

It has not been possible to use either of the first two methods in the evaluation of the Zuse translating system. First of all, Zuse is a new language, and only a few programs have been written from scratch in Zuse. Furthermore, apart from Borison's and Linton and Quong's work which provide some insight into the average size of modular programs, there have not been any studies on the properties of modular programs. In particular, there is no body of work which sheds light on the way abstract and semi-abstract types, constants, and inline procedures – concepts which are important in Zuse and whose use must be an integral part of a model of Zuse programs – might be used in real programs.

This evaluation of the Zuse translating system must therefore rely on the third of the approaches outlined above: we will construct a simple and artificial model of Zuse programs based on our intuitions of what programs will probably be like. Programs will then be generated based on the model, using "reasonable" values of the model's parameters. These programs will be submitted to the Zuse translating system and the behavior of the system will be reported. We cannot argue that the generated programs are representative of real Zuse programs, but we will argue that when submitted to the translating system they will provide some insight into the system's behavior.

8.2.1 The Model

One can think of many interesting quantitative and qualitative measures of modular programs. Examples include: the average number of modules in a program, the average number of exported items in a module, the structure of the import graph formed by the modules (k-nary tree, DAG, sparse, full, etc.), and the structure of the graph formed by the exported items (such as the

call-graph of exported procedures or the dependency graph of exported types).

The primary parameters of the simple model we propose are the structure of the import graph (particularly its height), the module-size (the number of exported items per module), and the item-size (the number of references each item makes to imported items). In order to exclude programs containing illegal circular declarations (see Section 4.2), we restrict the import graph to a hierarchical level-organized structure, where each module may only import modules on lower levels. Although the design of the model is primarily motivated by our intuition of what a representative Zuse program should look like, it is also influenced by the translating system designs of the previous chapters. Since the running times of some of the proposed algorithms are proportional to the height of the import graph, some to the height of the CET, and others to the size and number of deferred and inline procedures, it is important that the model allows these parameters to be varied easily.

Formally, a program is described by an 8-tuple $\mathcal{T}$:

$$\mathcal{T} = (items_{conc}, items_{semi}, items_{abs}, ref, work, import, diff, modules)$$

The first three entries of a $\mathcal{T}$-tuple describe the number of concrete, semi-abstract, and abstract items generated by the model. The next two entries describe the structure of the generated items, and the last three entries determine the structure and size of the module import graph. See Table 8.1 for a more detailed description. Each value in a $\mathcal{T}$-tuple may take on any one of four different forms: a, $[a \cdots b]$, $N(m, \sigma)$, and $[a, N(m, \sigma), b]$, where a and b are integers and m and σ are real numbers. These different forms allow us either to produce programs which are almost completely deterministic (by using the a-form) or to introduce a level of non-determinism by using any of the other three forms, which select values using either a rectangular or normal distribution. Table 8.2 below shows how an actual value for a particular module or item is selected.

As we can see, our model is deliberately vague on the subject of the realization of the exported items. In particular, the model does not say anything about the structure of the exported types or the actions which should be taken by the exported procedures. It would, of course, be possible to parameterize the model with respect to these and other aspects of modular programs (static nesting depth of procedures, local items, variable usage, etc.), but we have, for the sake of simplicity, elected not to do so. Instead, we will make all types and structured constants records, make all scalar constants integers, and limit the statements in procedures and module bodies to procedure calls and a set of simple assignment, loop, and conditional statements operating on integers. While this simplistic view of the internal structure of modules may generate

$items_{conc}$	: A tuple (*type*, *variable*, $const_{scalar}$, $const_{struct}$, $proc_{not\text{-}inline}$, $proc_{inline}$) describing the number of concrete items of each kind each module should export.
$items_{semi}$	: Similar to $items_{conc}$ but for semi-abstract items.
$items_{abs}$	: Similar to $items_{conc}$ but for abstract items. $items_{abs}$ also has an additional entry, *module-body*, giving the number of module bodies (between 0 and 1) per module.
ref	: A tuple, similar to the *item*-tuples, describing the number of references an item may make to similar imported items.
work	: A tuple ($proc_{not\text{-}inline}$, $proc_{inline}$, *module-body*) describing the amount of work (in number of statements) performed by these items.
import	: The total number of modules a module may import.
diff	: The possible differences in levels between an importer and an exporter.
modules	: A list $< l_{\mathcal{L}-1}, l_{\mathcal{L}-2}, \ldots, l_0 >$ of the number of modules on each of $\mathcal{L}$ levels. Level $\mathcal{L}-1$ contains only the top-level program module and, hence, $l_{\mathcal{L}-1} = 1$. Level 0 contains only leaf modules, i.e. modules which do not import anything.

Table 8.1: Semantics of the entries in a $\mathcal{T}$-tuple.

a	: Constant distribution.
$[a \cdots b]$	: Rectangular distribution of integer values between a and b.
$N(m, \sigma)$	: Integer normal distribution with mean m and standard deviation σ.
$[a, N(m, \sigma), b]$	: Truncated normal distribution.

Table 8.2: Different forms of values in a $\mathcal{T}$-tuple.

programs inadequate for any tasks other than the one at hand,[1] for our present purposes it works well.

8.2.2 An Example

A simple example is given by the tuple $\mathcal{T}_0$ below:[2]

$$\begin{aligned}\mathcal{T}_0 \quad = \quad & (\equiv 0, \equiv 0, (1,0,0,0,0,1), \\ & \equiv [0, N(2,1), 4], \equiv [0 \cdots 2], \\ & [0 \cdots 2], 1, <1,2,[3 \cdots 5] >)\end{aligned}$$

Any program P generated from $\mathcal{T}_0$ will contain between six and eight modules: one main program module (M_2_0),[3] between three and five leaf modules (M_0_0, M_0_1, M_0_2, M_0_3, M_0_5), and two modules on the intermediate level (M_1_0, M_1_1). Each module (except M_2_0) will export 2 abstract items (an abstract type and an abstract inline procedure), and may import modules on the immediately lower level only. Each module may import between 0 and 2 modules. The number of imported types referenced by the realization of an exported abstract type is selected using a truncated normal distribution with mean 2 and standard deviation 1. The number of procedure calls in the inline procedures is given by the same distribution. In addition to these calls, extra work (between 0 and 2 statements) is added to each inline procedure and module body. Figures 8.1 and 8.2 illustrate a program generated from $\mathcal{T}_0$.

8.3 Experimental Approach

In preparation for the next section in which the results of the sPaster and dPaster test runs will be analyzed, this section will present statistics regarding the input programs and describe our experimental set-up.

8.3.1 Experimental Configuration

The tests were carried out on the network of Sun-3 workstations shown in Figure 6.1. The sites were connected by a 10Mb/s Ethernet network and the workstations ran the `Unix` operating system and the NFS distributed file system. During the tests the workstations were otherwise unloaded, and the Slaves' load

[1] Such as evaluating module compilers or structure-oriented editors, for example.

[2] $\equiv X$, where X is any one of the four value forms described above, is an abbreviation of a tuple consisting of all X's.

[3] Each module is named M_x_y, where x is the level of the module and y is the module's sequence number within its level.

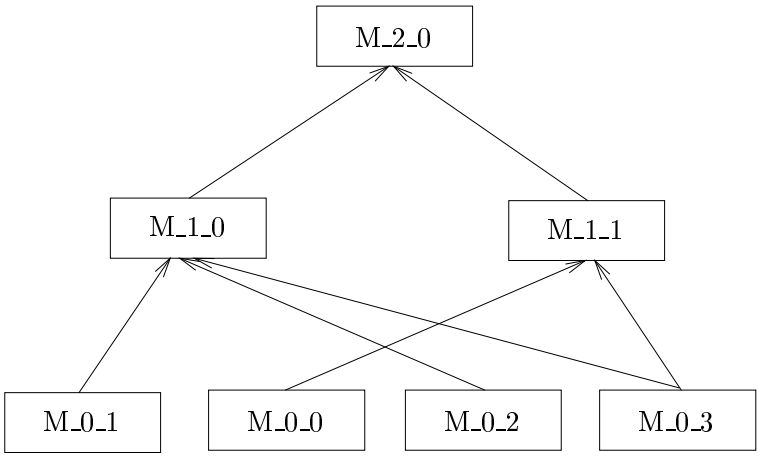


Figure 8.1: A program generated from $\mathcal{T}_0$.

balancing function (see Section 6.6.2 and the discussion in Section 8.4.4 below) was selected so as to balance Phase 3 processing ($cost(m, s) = m.c$). Each program was processed at least 30 times, and the time spent in each phase (as perceived by the Master and each of the Slaves) was recorded. All times given are averages of all the runs in wall time seconds.

One problem we had to deal with during the tests was client file caching (see Section 6.9). A program which is run several times on the same input data will often run faster the second and following times than the first time. The reason is that the second time the file cache already contains the program's load image and any files read by it during the previous run, and therefore these files will not need to be read from secondary storage. While this is in general the preferred behavior of a distributed file system, it renders unreasonable the results of test runs of IO-intensive programs such as the sPaster which presumably will never be run more than once on the same input data! We therefore emptied the file-cache on all machines prior to each test.

8.3.2 dPaster Process Allocation

During the tests the dPaster processes (Master, Servant, and Slaves) were placed on the available sites so as to minimize the total execution-time. The six possible allocation strategies are shown in Table 8.3. For a small number of sites ($2 \cdot \cdot 5$) Strategy-B (where the Master and Servant each share sites with

```
PROGRAM M_2_0;
  IMPORT M_1_0, M_1_1;
  TYPE T0 == RECORD [x_1_0 : M_1_0`T0; x_1_1 : M_1_1`T0 ];

  INLINE PROCEDURE I0 : (REF f : T0) +=
  BEGIN M_1_0`I0 (f.x_1_0); M_1_0`I0 (f.x_1_1); END I0;

  VARIABLE f : T0;
BEGIN I0 (f); END M_2_0.
```

```
SPECIFICATION M_1_0;
  TYPE T0 =    ;

  PROCEDURE I0 : (
     REF f : T0) =;
END M_1_0.
```

```
IMPLEMENTATION M_1_0;
  IMPORT M_0_1, M_0_2, M_0_3;
  TYPE T0 += RECORD [
                    x_0_1 : M_0_1`T0;
                    x_0_2 : M_0_2`T0;
                    x_0_3 : M_0_3`T0 ];

  INLINE PROCEDURE I0 : (REF f : T0) +=
  VARIABLE i : INTEGER;
  BEGIN
     M_0_1`I0 (f.x_0_1);
     IF i = 10 THEN i := 20;  ENDIF;
     M_0_3`I0 (f.x_0_3);
     CASE i OF 5 : i := 20;   | 13 : i := 9;   ENDCASE;
  END I0;
END M_1_0.
```

```
SPECIFICATION M_0_0;
  TYPE T0 =    ;
  PROCEDURE I0 : (
     REF f : T0) =;
END M_0_0.
```

```
IMPLEMENTATION M_0_0;
  TYPE T0 += RECORD [x : CARDINAL ];

  INLINE PROCEDURE I0 : (REF f : T0) +=
  BEGIN f.x := 7;  END I0;
END M_0_0.
```

Figure 8.2: Three of the modules of the program whose import graph is given in Figure 8.1

Strategy	Master	Servant	Slaves	Preferred
Strategy-A	Site 1	Site 2	Sites $3\cdot\cdot$	For 6 or more sites.
Strategy-B	Site 1	Site 2	Sites $1\cdot\cdot$	For $2\cdot\cdot5$ sites.
Strategy-C	Site 1	Site 1	Sites $2\cdot\cdot$	
Strategy-D	Site 1	Site 1	Sites $1\cdot\cdot$	For 1 site.
Strategy-E	Site 1	Site 2	Sites $2\cdot\cdot$	
Strategy-F	Site 2	Site 1	Sites $2\cdot\cdot$	

Table 8.3: The six different allocation strategies for the dPaster processes. The table shows that Strategy-A (which is the preferred strategy for 6 or more sites) allocates the Master and the Servant each on their own sites and the Slaves on the remaining sites. Strategy-B, on the other hand, places the Master and the Servant on sites together with one Slave.

one Slave) turned out to be the best one, while Strategy-A (where the Master and Servant each have a site of their own) is to be preferred for six sites or more. The reason is that the amount of communication between the Slaves and the Master and the Slaves and the Servant increases with the number of slaves, and at some point it becomes advantageous to reserve two sites for exclusive use by the Master and Servant rather than using those sites for optimization and code generation.

It is likely – although we are not able to present any hard evidence for this – that the break-off point between Strategy-A and Strategy-B (the number of slaves when Strategy-A becomes better than Strategy-B) is also dependent on the program being bound. The preferred allocation strategies in Table 8.3 were arrived at through tests using the input program `sPaster-B` (see the next section), and we believe that slightly different results would have been obtained using some other input. However, even if that were to be the case, the dPaster could not itself determine the optimal allocation policy for a given program, since that would require it to know details of the program even *before* having looked at it. It is therefore reasonable to use a "typical" program, as we have done, to determine a "best average" process allocation strategy.

8.3.3 Characteristics of the Test Programs

In order to be able to relate the running time of the zts to different characteristics of the input program, we have instrumented the compiler and binder with operations to determine some statistical properties of the program being

M	: Number of modules.
LOC	: Total size in thousands of lines of code.
S	: Total number of statements (in thousands).
P	: Number of procedures.
T_a	: Number of abstract types.
C_a	: Number of abstract constants.
I_a	: Number of abstract inline procedures.
P_d	: Number of referenced deferred procedures (procedures for which code is generated during Phase 3).
S_d	: Number of deferred context conditions.
I_2	: Number of inline expansions performed during Phase 2.
I_3	: Same as I_2 but for Phase 3.
I_c	: Percentage of dynamic calls expanded, for typical input.
E	: Number of constant expressions, excepting call-graph attributes.
H_e	: The height of the CET, excepting call-graph attributes.
$\overline{H_e}$	: Average height of all CET nodes, excepting call-graph attributes.
H_c	: The height of the call-graph, excepting recursive nodes.
$\overline{H_c}$	: Average height of all call-graph nodes.
H_i	: The height of the inline call-graph.
$\overline{H_i}$	: Average height of all inline call-graph nodes.

Table 8.4: Semantics of the measurements given in table 8.5.

translated. Our intention is also to use these functions to provide insight into the average and extreme properties of real programs. This information can then be used to determine reasonable values for the parameters of the model presented in the previous section, or to design a more accurate model. Table 8.4 gives the list of properties that ZTS can currently calculate along with their abbreviations.

At this point in time there only exist a few "real" programs written in Zuse. The only large program is the sPaster itself, which has been translated more or less mechanically into Zuse from the language in which it was originally written, Modula-2. Since the sPaster was not written in Zuse from scratch, we cannot argue that it is a good approximation of what a real Zuse program might look like. Nevertheless, five versions of the sPaster were constructed – differing in their amount of encapsulation and inlining – and used as input to

the sPaster and dPaster. A summary of the characteristics of each version is given in Table 8.5.

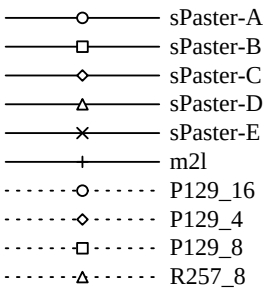
In addition to the five versions of the sPaster, we constructed four artificial programs (`P129_4`, `P129_8`, `P129_16`, `R257_8`) from the model in the previous section. The programs in the P-family (`P129_4`, `P129_8`, `P129_16`) all consist of 129 modules organized in 4, 8, and 16 levels, respectively. Their dominant characteristics are a rectangular import graph, a large amount of encapsulation and inlining, large modules, and a large amount of work per procedure. `R257_8` differs from these programs by being larger (more modules, more lines of code) but having fewer and smaller inline procedures and a much shallower inline call-graph. Furthermore, while the modules in the P-family of programs are all very similar, `R257_8` modules can vary significantly in size and structure. Table 8.6 lists the $\mathcal{T}$-tuples of each program. More detailed statistics can be found in Table 8.5.

8.4 Empirical Results and Analysis

We conclude this chapter by reproducing the running times of the sPaster and the dPaster[4] for five versions of one real program (the sPaster itself) and four artificial programs generated by the model presented in Section 8.2. We also analyze this empirical data and compare it to the analytical results from earlier chapters.

8.4.1 Total Binding Times and Speedup

Figure 8.3 shows the execution-times (wall time in number of seconds) for the dPaster and its sequential counterpart when binding the five versions of the sPaster and the four artificially generated programs. All graphs in this section use the following legend unless otherwise specified:



[4]The hPaster and iPaster have regrettably not been implemented at this point.

Program	M	LOC	S	P	T_a	C_a	I_a
sPaster-A	108	36	14	2024	28	34	0
sPaster-B	108	36	14	2024	28	33	431
sPaster-C	108	36	14	2024	0	0	0
sPaster-D	108	36	14	2024	0	0	431
sPaster-E	108	36	14	2024	16	27	202
P129_4	129	51	62	1905	471	448	653
P129_8	129	55	64	1905	471	448	653
P129_16	129	56	64	1905	471	448	653
P257_8	257	167	203	5158	800	775	647

Program	P_d	S_d	I_2	I_3	I_c
sPaster-A	879	5	0	0	0%
sPaster-B	953	15	41	1325	55.6%
sPaster-C	0	0	0	0	0%
sPaster-D	760	10	41	1325	55.6%
sPaster-E	1024	15	10	724	34.8%
P129_4	986	0	133	1815	N/A
P129_8	1129	0	268	3101	N/A
P129_16	1222	0	361	4040	N/A
R257_8	3889	0	14	2907	N/A

Program	E	H_e	$\overline{H_e}$	H_c	$\overline{H_c}$	H_i	$\overline{H_i}$
sPaster-A	533	29	3.6	24	3.6	0	0
sPaster-B	578	29	3.5	24	3.4	3	0.2
sPaster-C	55	0	0	24	3.6	0	0
sPaster-D	99	3	0.6	24	3.4	3	0.2
sPaster-E	497	28	3.6	24	3.5	2	0.2
P129_4	7542	26	6.1	4	1.4	3	0.9
P129_8	8645	48	14.4	8	3.2	5	1.3
P129_16	8360	87	32.1	16	7.0	7	1.7
R257_8	13629	33	7.9	8	2.5	2	0.4

Table 8.5: Source code statistics regarding five versions of the sPaster and four artificially generated programs. See table 8.4 for an explanation of the different measurements. The slight difference in average call-graph height ($\overline{H_c}$) between the different sPaster versions is due to intra-modular inlining. Local (non-exported) inline procedures do not take part in the binding process and do not show up in these statistics.

$$
\begin{array}{lcl}
\texttt{P129\_4} & = & (\equiv 0,\\
 & & \equiv 0,\\
 & & ([1\cdots5],0,[1\cdots5],[0\cdots1],[5,N(10,3),20],[0,N(5,2),20],[0\cdots1]),\\
 & & ([0,N(3,2),10],0,[0,N(3,2),10],[0,N(2,1),3],\\
 & & \quad [0,N(10,3),20],[0,N(2,1),3],[0\cdots50]),\\
 & & ([0,N(10,3),20],[0,N(2,1),3],[0,N(10,3),20]),\\
 & & [5,N(15,5),40],\\
 & & [1,N(2,0.5),3],\\
 & & <1,32,32,32,32>)\\
\\
\texttt{P129\_8} & = & (\&,\&,\&,\&,\&,\&,\&,<1,16,16,16,16,16,16,16,16>)\\
\\
\texttt{P129\_16} & = & (\&,\&,\&,\&,\&,\&,\&,<1,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8>)\\
\\
\texttt{R257\_8} & = & (\equiv 0,\\
 & & \equiv 0,\\
 & & ([1\cdots5],0,[1\cdots5],0,[5\cdots30],[0\cdots5],[0\cdots1]),\\
 & & ([0\cdots5],0,[0\cdots5],0,[0\cdots20],[0\cdots1],[0\cdots50]),\\
 & & ([0\cdots20],[0\cdots1],[0\cdots20]),\\
 & & [0\cdots40],\\
 & & [1\cdots8],\\
 & & <1,32,32,32,32,32,32,32,32>)
\end{array}
$$

Table 8.6: $\mathcal{T}$-tuples for the artificially generated programs. An ampersand (&) indicates that a tuple entry is the same as the one in the previous tuple.

For comparison we also give the execution-time of the SUN Modula-2 linker `m2l` (a pre-linker which calls the UNIX systems linker *ld* as part of the linking process) when linking the sPaster. Figure 8.4 shows the speedup of the dPaster relative to the sPaster and relative to itself running on one site. It is evident from Figure 8.4 that the speedup of the dPaster varies with the amount of encapsulation and inlining: `sPaster-C` – a completely concrete program – results in no speedup at all, while `sPaster-B` and `P129_16` yield a speedup of 2.6 and 3.5, respectively.

It is interesting to note that for all five versions of the sPaster, the dPaster (given enough sites) will always out-run `m2l`. This is of course an unfair comparison since the two programs differ in many ways: `m2l` is a pre-linker, while the dPaster performs every part of the linking process itself; `m2l`, like most system link-editors, has to resolve external symbols through name lookup, while ZTS binds all names already at compile-time; The dPaster performs inter-modular

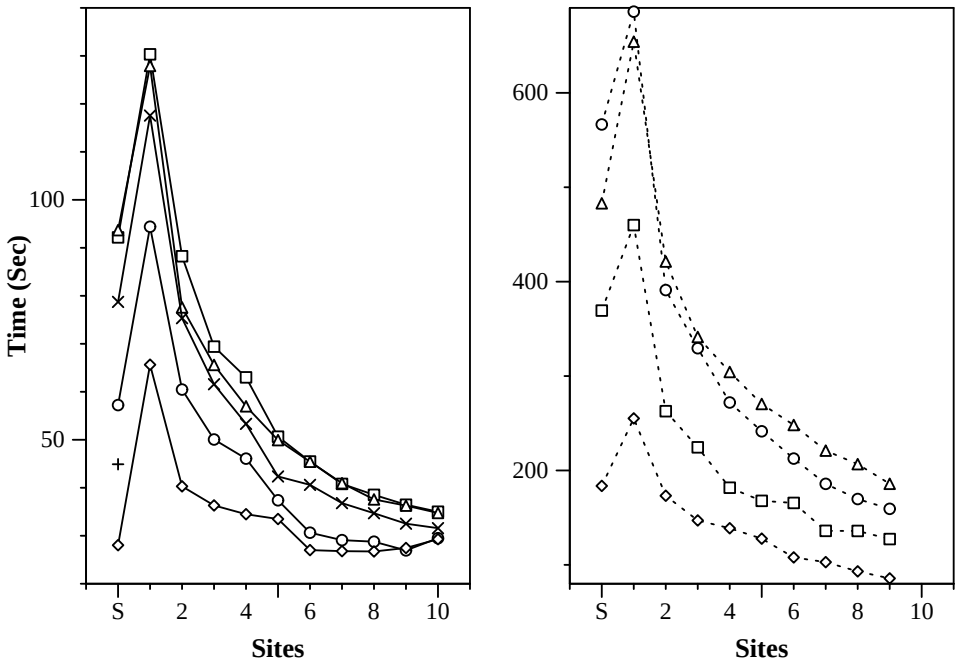


Figure 8.3: On the left is shown the running time of the sPaster (point S) and the dPaster running on 1–10 sites when binding five versions of the sPaster itself. The timing for the SUN Modula-2 linker `m2l` linking the sPaster is also given. To the right is shown the total running time of the sPaster and the dPaster when binding the four artificial programs. Only 9 sites were available for the test runs of the generated programs, which explains the missing data points.

optimizations, while `m2l` only does relocation. Furthermore, the SUN Modula-2 translation system produces better local code than ZTS, although inter-modular inlining can in some cases make up for ZTS' poor code quality (`sPaster-B` is only 14% slower than the sPaster produced by the SUN Modula-2 system with full optimization).

Also interesting is that the dPaster is never more than marginally faster than the sPaster for programs with no encapsulation and inlining, such as `sPaster-C`. This would seem to indicate the infeasibility of distributed link-editing. However, the current implementation of the dPaster is tuned to perform well for programs with large amounts of encapsulation and inlining, and its poor performance for other kinds of programs is by no means a proof that some level

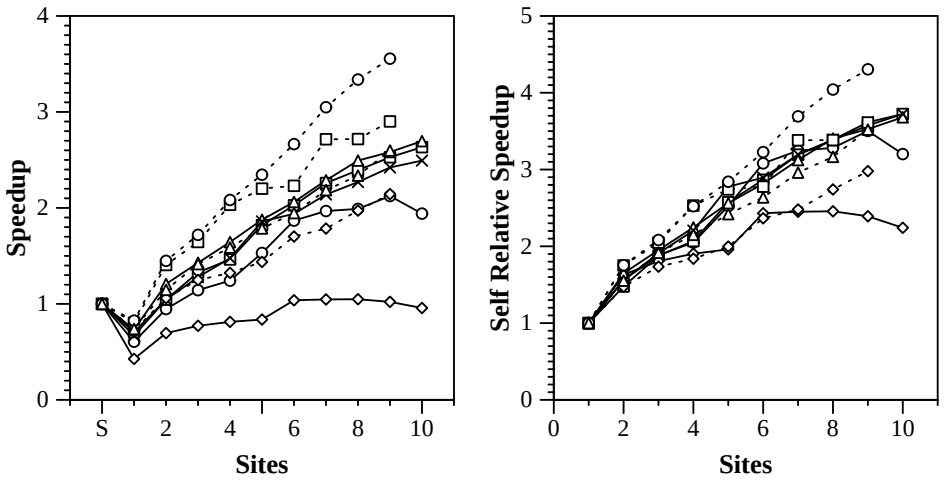


Figure 8.4: Speedup of the dPaster relative to the sPaster (left) and self-relative speedup of the dPaster (right).

of speedup cannot be achieved by some other techniques.

Figure 8.3 affords a further important observation, namely that `sPaster-D` (which has no encapsulation but full inter-modular inlining) and `sPaster-B` (which has full encapsulation and inter-modular inlining) result in very similar binding times. The conclusion must be that a system which performs inter-modular inlining might as well support full flexible encapsulation, since the extra translation-time cost will be close to negligible.

Figure 8.5 shows how the total sPaster time is distributed over the four phases for some of the test programs. Figure 8.6 shows the same results, but for the dPaster running on 9 sites. One thing we notice from these figures is that the dPaster spends a proportionally smaller part of its time on Phase 3 and 4 than the sPaster, but more of its time on Phase 1 and 2. This is because the dPaster achieves higher levels of speedup for Phase 3 and 4 than for Phase 1 and 2. We will examine this further in the next four sections, which we devote to discussions of each phase in turn.

Figure 8.7 also gives the time spent by the sPaster in each of the four phases, but this time we also show the percentage of the time that is actually spent performing useful computation. This data is interesting because it indicates whether the sPaster's four-phase design is a reasonable one, or if a different design (which would interleave the actions of some of the phases in order to take advantage of unused CPU power) would be likely to perform better. One could,

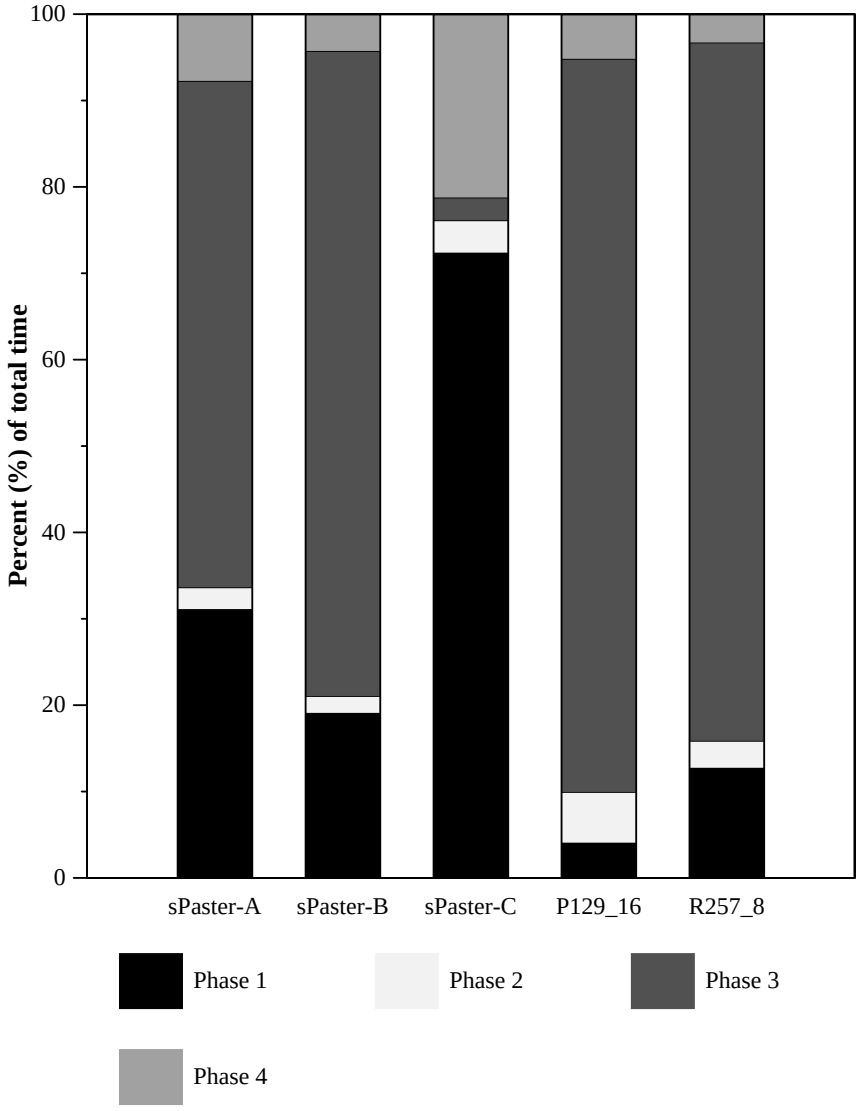


Figure 8.5: Percentage of the total sPaster time spent in each of the four phases.

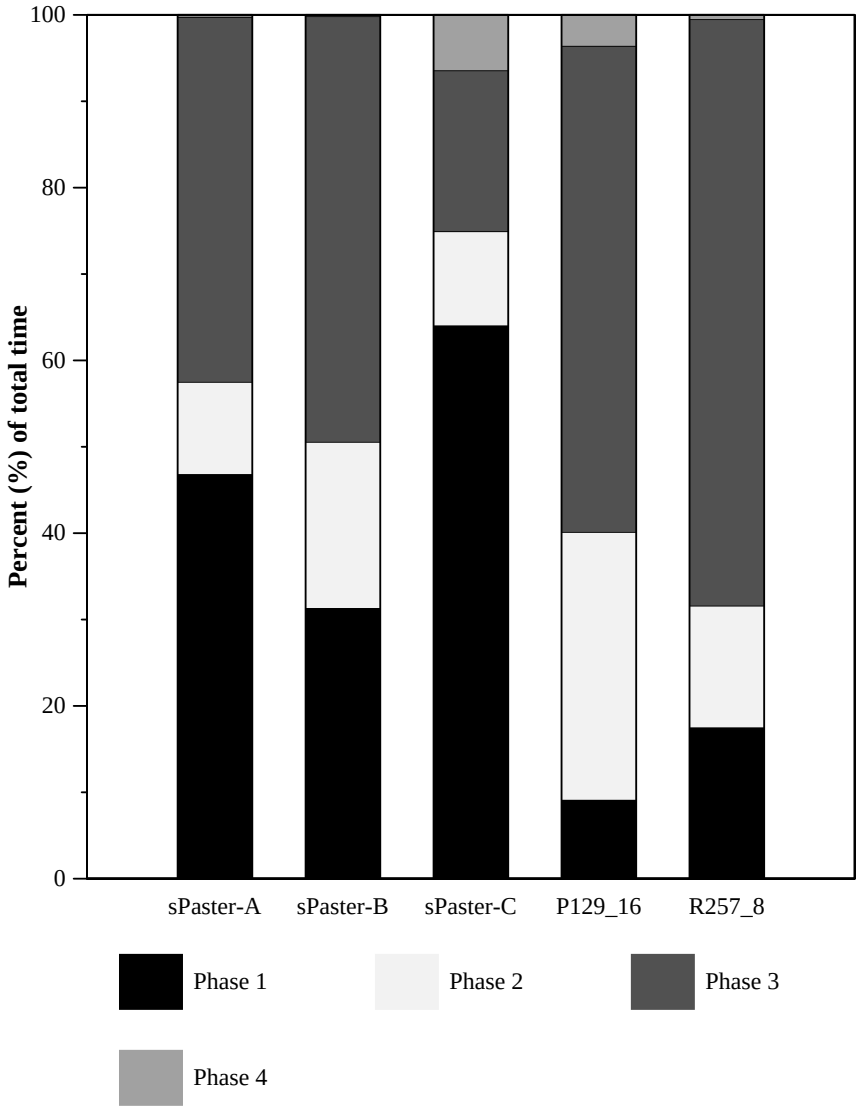


Figure 8.6: Percentage of the total dPaster time spent in each of the four phases when running on 9 sites.

for example, imagine a design in which Phase 2 runs in parallel with Phase 1, evaluating expressions and expanding inline calls in inline procedures as these become available. Or, one might let Phase 4 execute concurrently with Phase 1 and Phase 3, much as is done in the dPaster. We can deduce from Figure 8.7 that for some programs such designs may indeed perform better than the current one, but only marginally so. For example, the sPaster's Phase 2 spends most of its time in an idle state when processing `P129_16`,[5] and it may prove beneficial to spend this idle time doing Phase 1 or Phase 3 computations. Similarly, very large programs such as `R257_8` may have enough idle time during Phase 3 (due to the writing of the generated code) that it would pay off to perform relocations (Phase 4) in parallel. However, we believe that the greatest potential gain will come from concurrent execution within Phase 1. The source of almost all the idle time in Phase 1 is locating and opening new object code files, and it could be an improvement if this was done in parallel with the processing of the data read from the current file.

One must, however, keep in mind that there may be instances when the overhead of having several concurrent threads of control may very well cancel out much of the advantage of reducing the sPaster's idle time. Running Phase 2 and 3 in parallel may work well for programs like `P129_16` in which Phase 2 contains much idle time, but is likely to perform less well for programs like `sPaster-B`[6] in which Phase 2 is completely CPU-bound.

8.4.2 Analysis – Phase 1

Figure 8.10 shows that during Phase 1 the sPaster spends most of its time searching for and opening object files. This is particularly true when the sPaster is given long search-paths or when the object files reside deep in the directory hierarchy (this was the case for the sPaster versions). For programs such as `sPaster-C` which have none or very little encapsulation and inlining and hence few deferred procedures, the copying of code and data from the object files to the resulting executable file (the Process Data-entry in Figure 8.10) also becomes time-consuming. Similarly, loading the Expression Section is expensive for programs such as `P129_16` that contain a significant amount of encapsulation. The Phase 1 timings and the dPaster's Phase 1 speedup are given in Figures 8.8 and 8.9.

[5] The reason is that `P129_16` contains a large number of abstract structured constants which, once they have been constructed, have to be written to the executable file. The idle time is simply spent waiting for these write-operations to complete.

[6] `sPaster-B` is the program for which the current implementation was tuned, which explains why it has the least idle time of all the test programs.

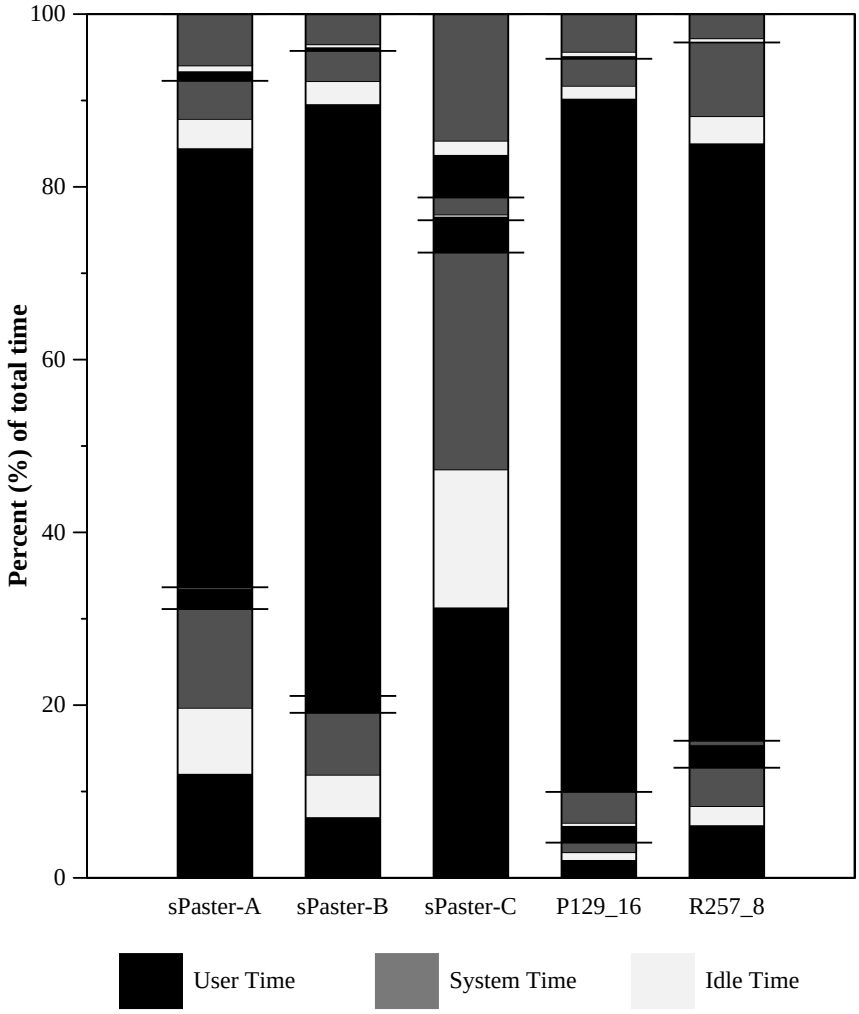


Figure 8.7: Percentage of the total sPaster time spent in each of the four phases. Each bar is divided into four parts, one per phase, with Phase 1 at the bottom and Phase 4 at the top. Each phase is divided into three categories: User Time (the CPU time used by the phase), System Time (the CPU time spent in operating system routines), and the Idle Time (the time spent in a suspended state waiting for system routines to finish executing).

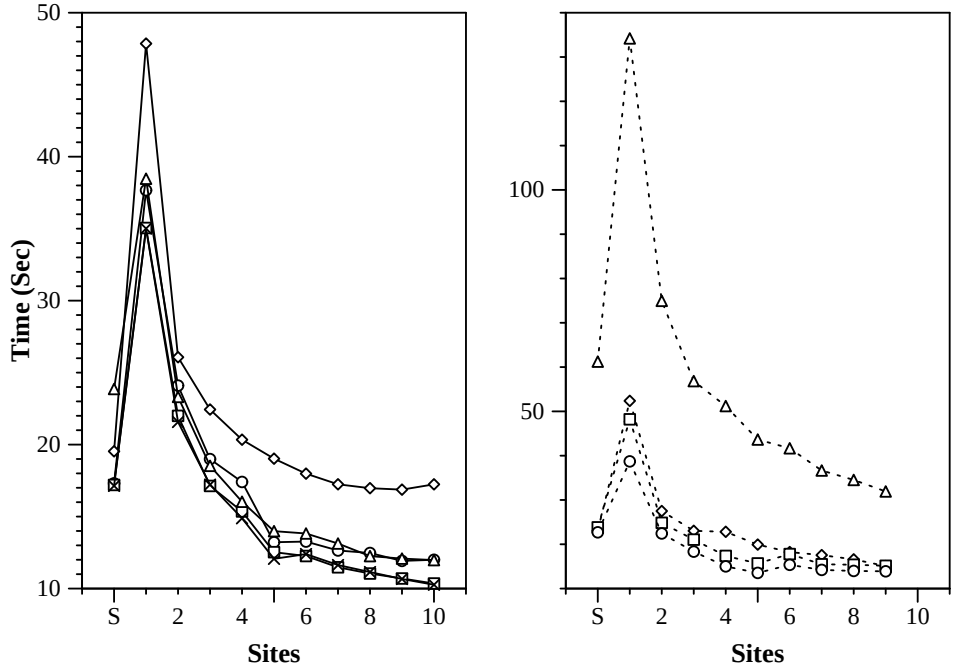


Figure 8.8: Phase 1 processing times.

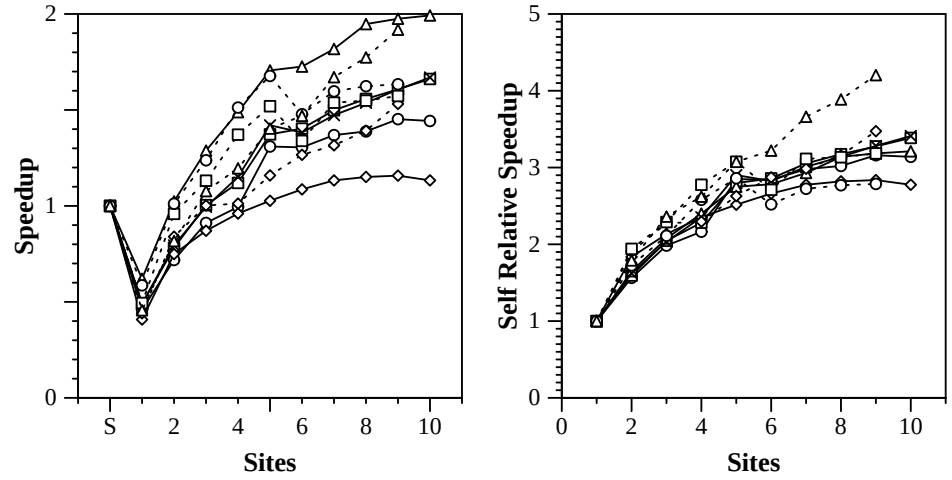


Figure 8.9: Speedup of Phase 1.

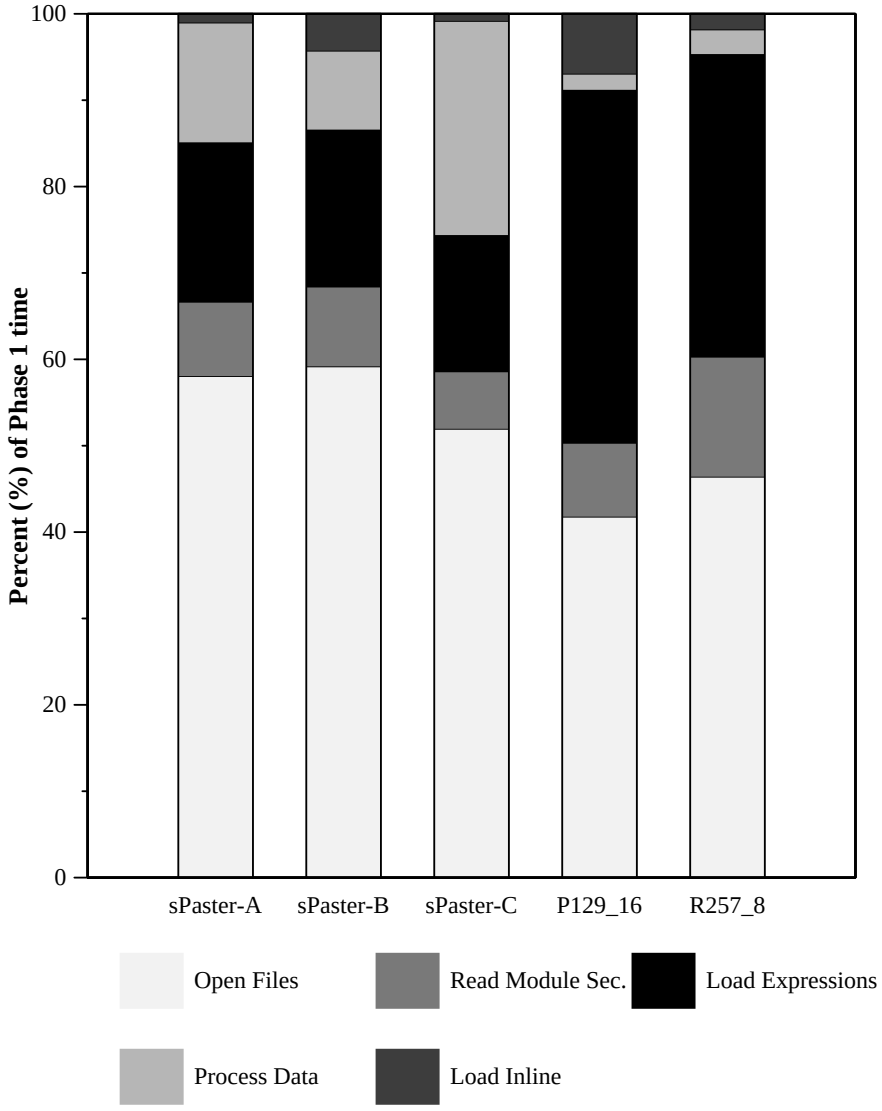


Figure 8.10: Breakdown of the sPaster's Phase 1 activity. Consult Algorithm 5.8 on page 146 for a description of the various operations. The Open Files-entry includes the time for locating the files and loading the preambles. The Process Data-entry is the combination of all the Algorithm 5.8 operations concerned with the Data, Binary Code, and Relocation Sections.

8.4.3 Analysis – Phase 2

Already in Chapter 6 we noted that Phase 2 is the dPaster phase that is the most challenging to parallelize. This is also evident from Figures 8.11 and 8.12, which show that, in most cases, the distributed Phase 2 is between a factor 2 and 5 *slower* than its sequential counterpart. The reason is, of course, that in addition to evaluating expressions and expanding inline procedures, the dPaster's Phase 2 also performs the distribution of the resulting values and intermediate code. Particularly expensive is the replication of inline procedures and the values of abstract structured constants and object type templates, since these tend to be large.

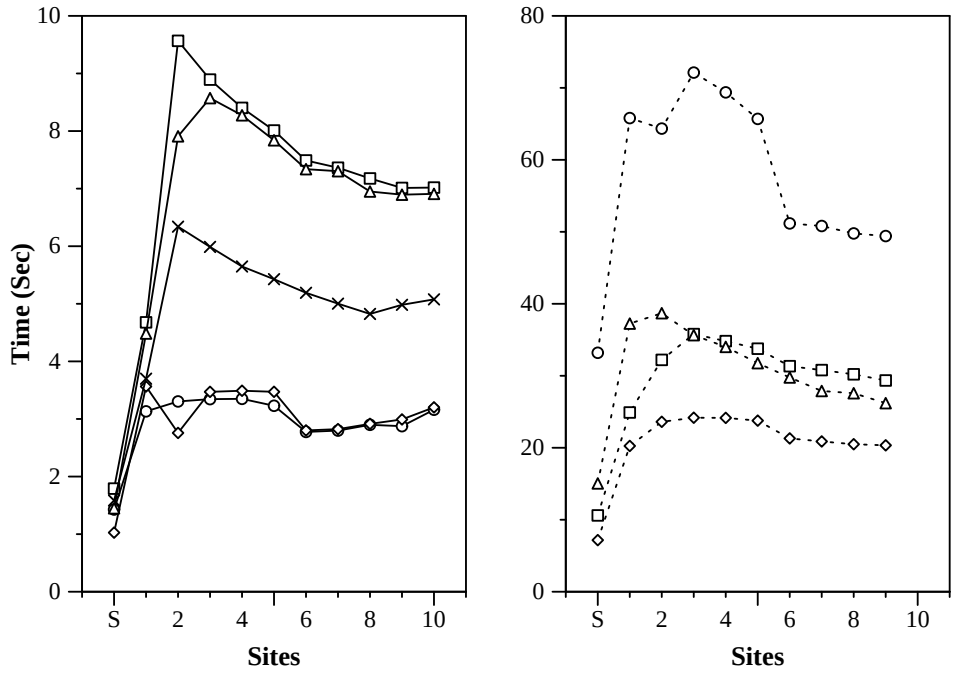


Figure 8.11: Phase 2 processing times.

Figures 8.13, 8.14, 8.15, and 8.16 break up the Phase 2 timings into evaluation and inlining. These figures show that the main cost of the dPaster's Phase 2 comes from the distribution of inline procedures, not from the evaluation of expressions. It is even the case that the dPaster (running on 6 sites or more) achieves a small speedup (1.1) when evaluating `P129_16`'s expressions.

In Section 6.11.2 we analyzed the expected behavior of synchronized ver-

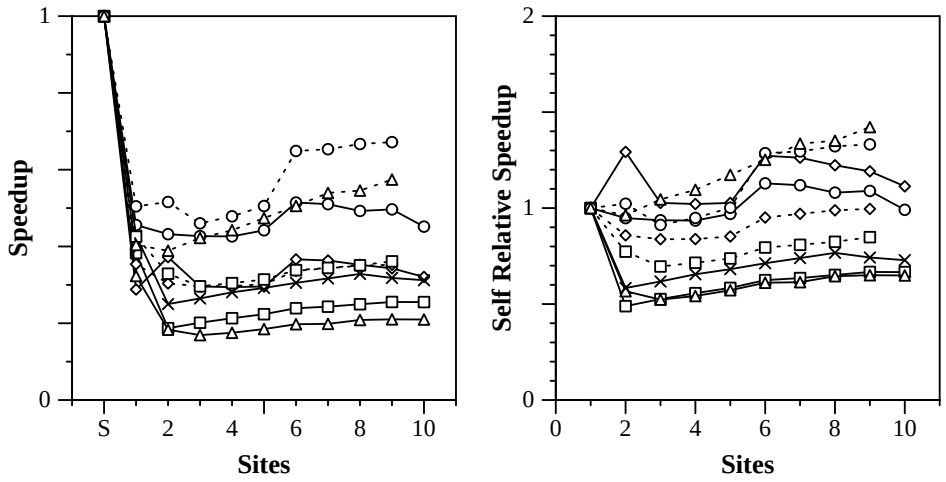


Figure 8.12: Speedup of Phase 2.

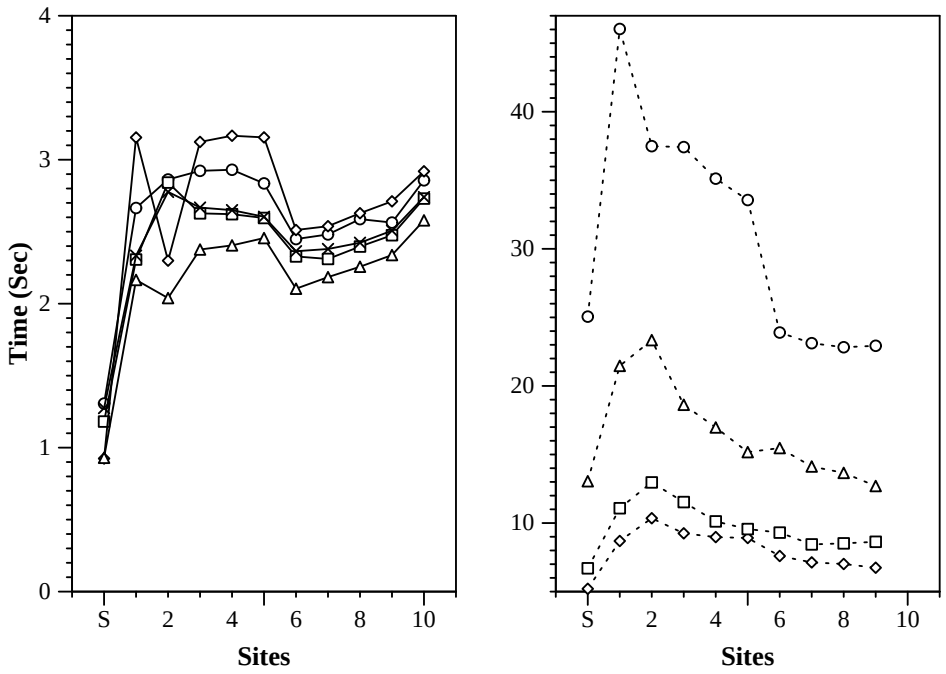


Figure 8.13: Phase 2 expression evaluation times.

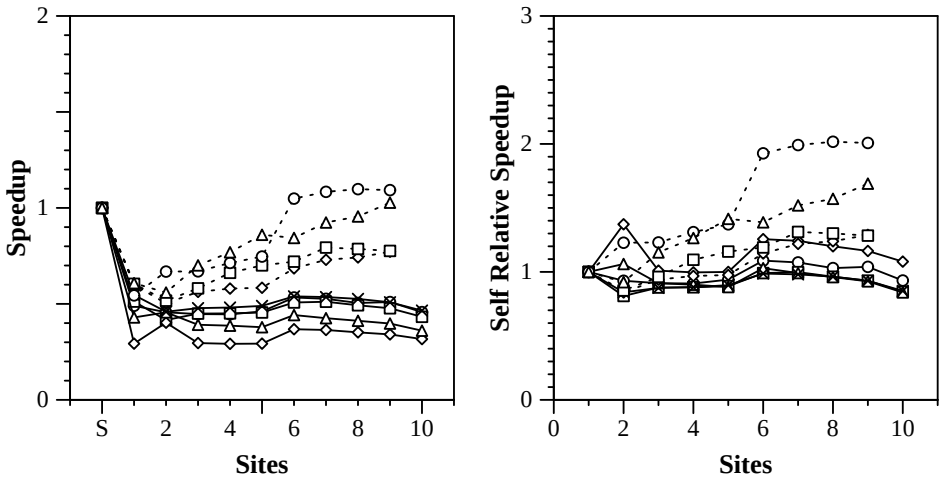


Figure 8.14: Phase 2 expression evaluation speedup.

sions of the expression evaluation and termination detection algorithms. We found that for both algorithms the number of rounds grows with the number of slaves and with the height of the supergraph formed when internal paths in the expression graph are collapsed. This analysis is partially supported by Figures 8.17 and 8.18. These graphs show that programs with high expression graphs (such as `P129_16`) require more termination detection rounds and expression sends than programs of lower height (such as the sPaster versions). However, while the number of expression sends grows with the number of slaves as predicted, this does not seem to be true for termination detection rounds. Rather, the number of such rounds *decreases* ever so slightly with the number of slaves. This is because the predictions in Section 6.11.2 relate to a *synchronized version* of the expression evaluation algorithm, while the one actually implemented is *asynchronous*. In other words, a slave does not have to wait for the other slaves to finish a round before it starts its next one. Consequently, when there are many Slaves involved in the expression evaluation, each Slave is likely to receive a steady stream of incoming values to process and will be less likely to have to enter a passive state.

8.4.4 Analysis – Phase 3

The previous section shows that the reason that the distributed Phase 2 yields a speedup below 1 is that it has to do a fair amount of replication of global data.

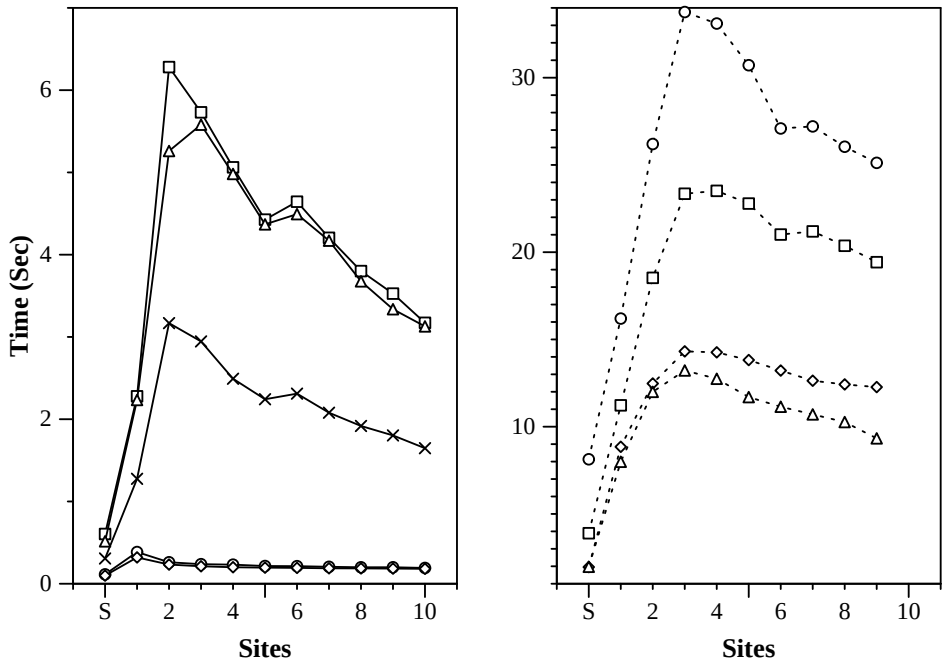


Figure 8.15: Phase 2 inline expansion times. This includes the time for updating the intermediate code of inline procedures with the newly computed CET values.

But thanks to this replication, Phase 3 is freed of all inter-Slave communication and makes up for the excessive cost of Phase 2 by yielding for most programs a speedup ranging from 3 to 6 (see Figures 8.19 and 8.20).

Figure 8.21 shows the relative cost of each of the sPaster's Phase 3 operations. Obviously, code generation takes up most of the sPaster's time during Phase 3. This is particularly true for programs with a significant amount of Phase 3 inlining, since inline expansion creates large procedures for which code generation is expensive. The data for the Read Deferred-entry in Figure 8.21 should be taken with a grain of salt: since ZTS uses memory mapping techniques for the reading of the object code files (see Section 5.8.3), the actual reading of the deferred code will take place when the code is needed.

There is still one major problem with Phase 3, and that is load balancing. This is evident from Figure 8.22 (left) which shows that although Phase 3 will perform quite well on the average, it will on occasion perform very poorly. The problem is that the simple load balancing scheme which runs during Phase 1 (see

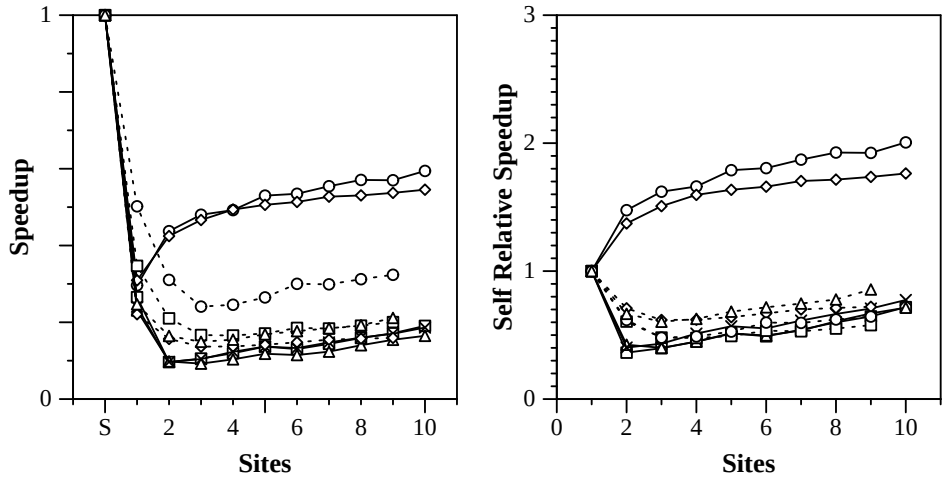


Figure 8.16: Phase 2 inline expansion speedup.

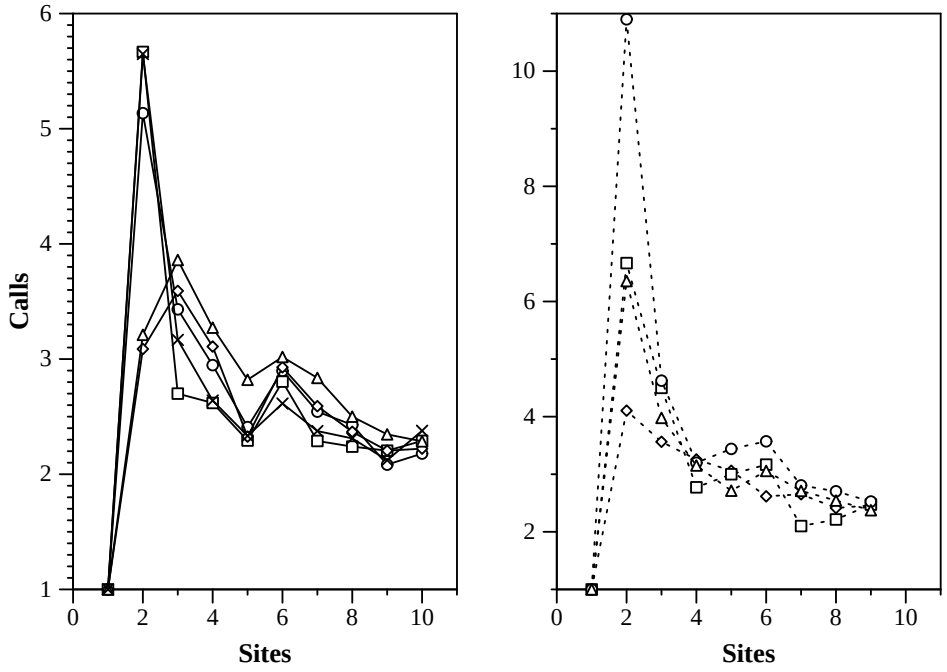


Figure 8.17: Average number of termination detection rounds initiated by the Master during Phase 2.

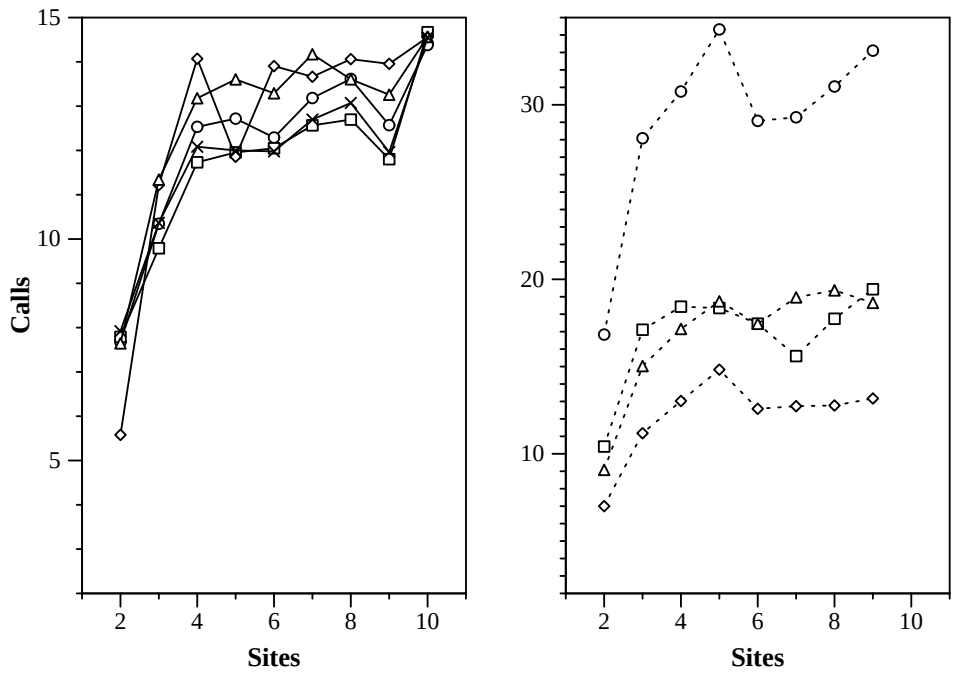


Figure 8.18: Average number of expression sends per Slave during Phase 2.

Sections 6.6.2 and 6.10.1) cannot take into account the amount of inlining which will be performed during Phase 3, since this is not known until after Phase 2. As a result, the load balancing algorithm will sometimes unknowingly assign several modules which are high in inlining to the same Slave, and that Slave will end its Phase 3 processing much later than the other Slaves. This is clearly shown in Figure 8.22 (right). We are currently considering alternative load balancing algorithms which allow modules to migrate from busy to unoccupied slaves.

8.4.5 Analysis – Phase 4

While the sPaster performs all its relocation during Phase 4, the dPaster performs its relocation concurrently with the Slaves' Phase 1, 2, and 3 processing. Therefore, all the dPaster's Master has to do during Phase 4 is wait for the Servant to finish the relocation. The length of this wait depends on the total amount of relocation information, when that information is produced, and the amount of processing resources available to the Servant:

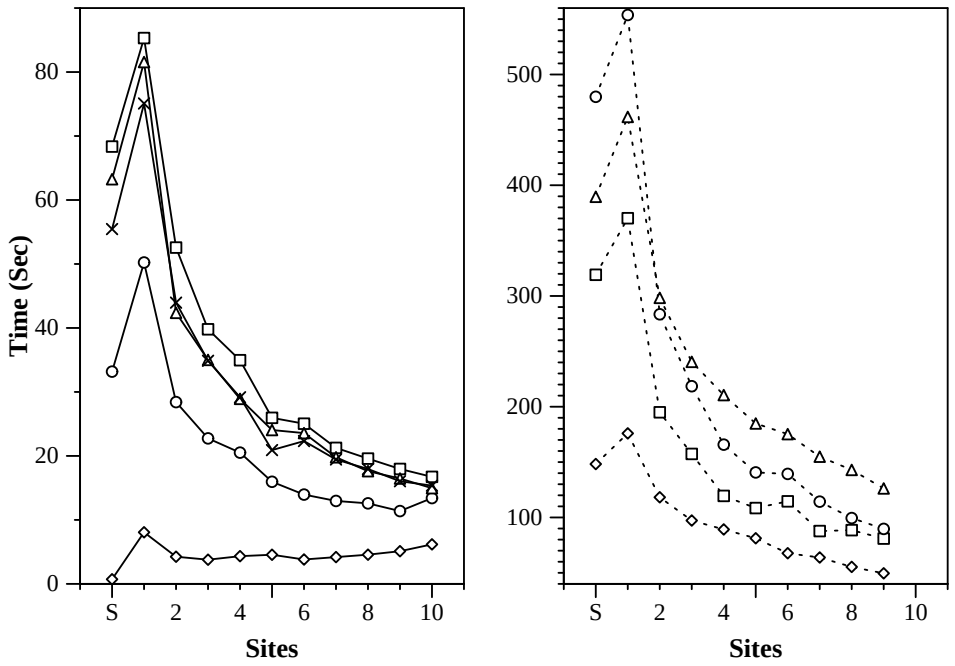


Figure 8.19: Phase 3 processing times.

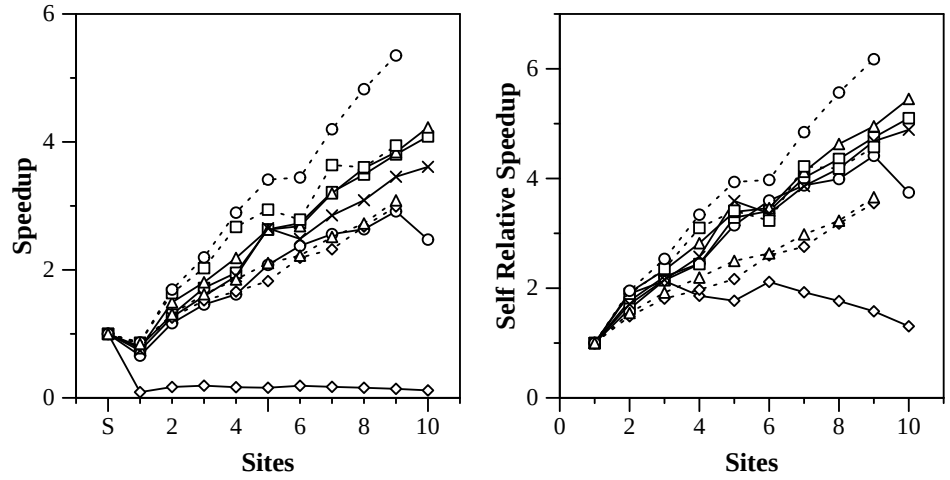


Figure 8.20: Speedup of Phase 3.

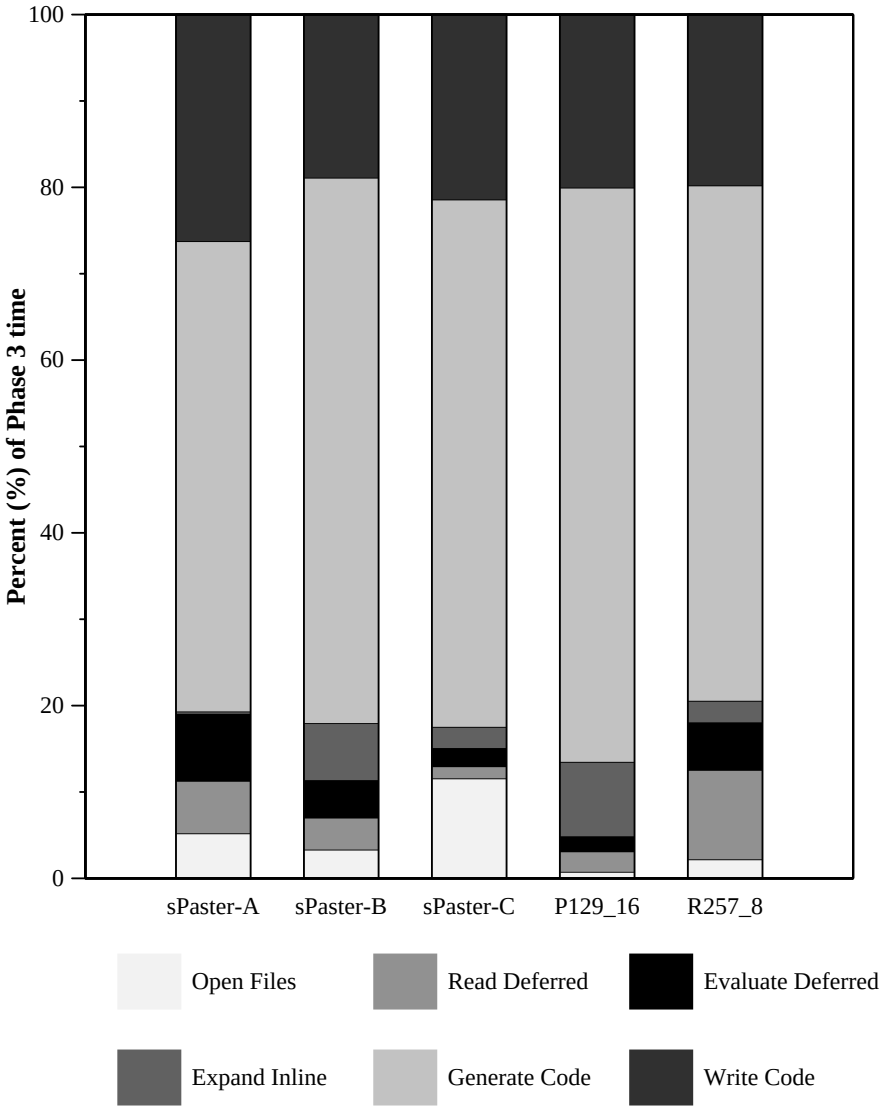


Figure 8.21: Breakdown of the sPaster's Phase 3 activity. Consult Algorithm 5.10 on page 148 for a description of the various operations. The Evaluate Deferred-entry corresponds to the operation in Algorithm 5.10 that fills in CET values. The Generate Code-entry includes the time for recomputing basics blocks and next-use information. The Write Code-entry includes the time for generating binary code from an internal symbolic machine code format.

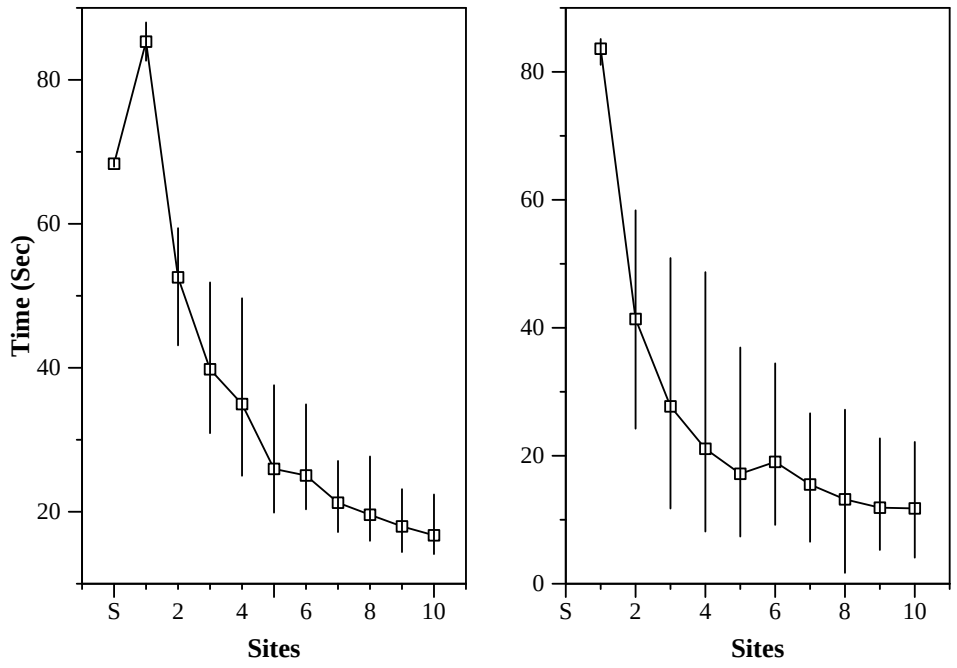


Figure 8.22: The graph to the left shows the average, shortest (lowest point of vertical line), and longest (highest point of vertical line) Phase 3 processing times over all the test runs. The graph to the right shows the longest and shortest processing time of *any individual Slave*.

- For programs such as `P129_16` with many deferred procedures and where each procedure makes many calls, the Slaves may produce more relocation information during Phase 3 than the Servant can consume. The net result is that the Servant has to continue the relocation after the Slaves are finished with Phase 3.

- Similarly, programs such as `sPaster-C` which produce all their relocation information during Phase 1 and need very little Phase 3 processing, leave the Servant with little time during Phase 3 to perform relocation.

- If the Servant is allowed to execute on its own site (as in Strategy-A and Strategy-F), it will have more resources available and will be much more likely to finish quickly than if it would share sites with the Master or one of the Slaves.

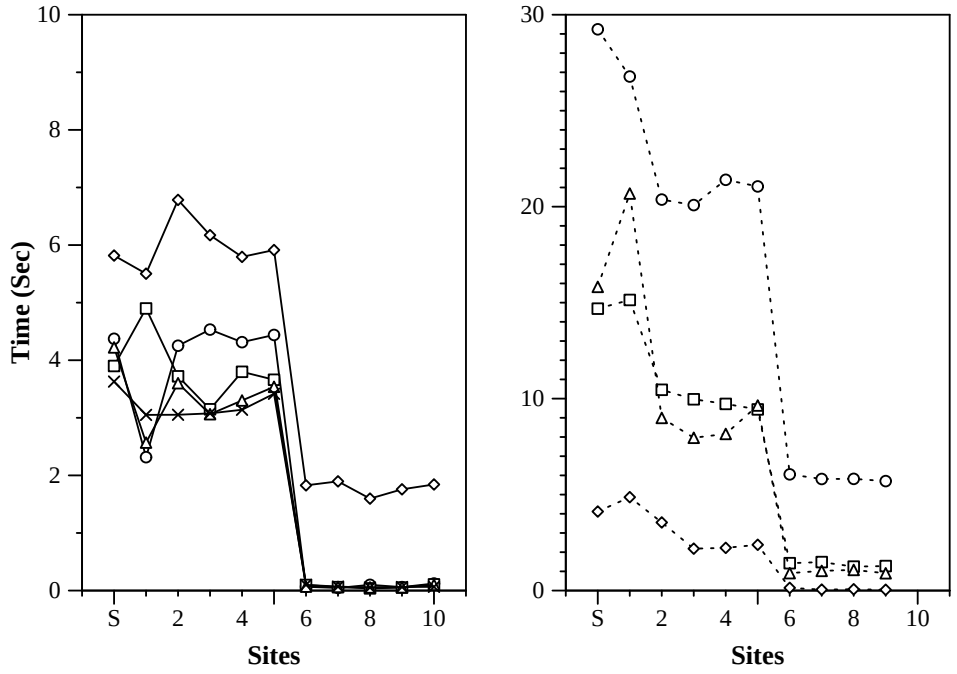


Figure 8.23: Phase 4 processing times.

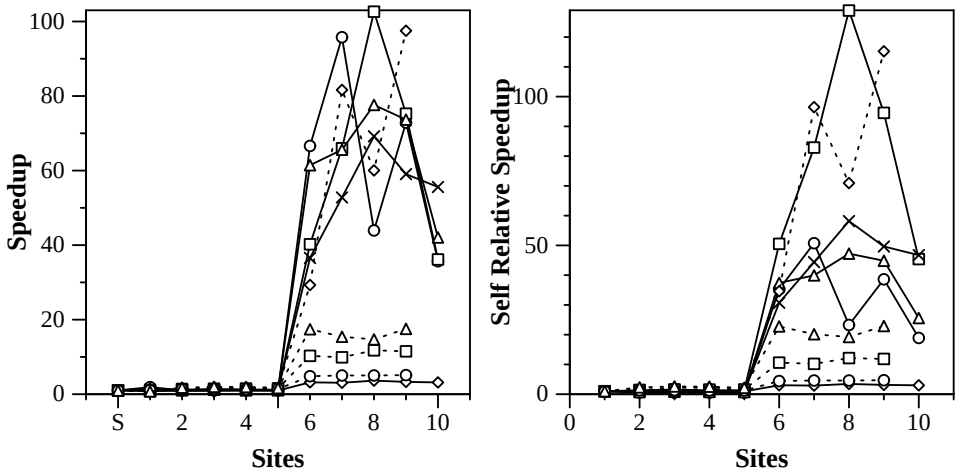


Figure 8.24: Speedup of Phase 4.

The last point is illustrated clearly in Figures 8.23 and 8.24, which show little Phase 4 speedup for Strategy-B and Strategy-D ($1 \cdot \cdot 5$ sites) but substantial speedup for Strategy-A (more than 6 sites).

8.5 Summary

The results of the measurements presented in this chapter indicate that for programs with medium to large amounts of encapsulation and inlining, the dPaster achieves speedup factors between 2 and 3.5, given 6 or more processors. Even with as few as 3 processors available, the dPaster is almost always faster than the sPaster. Furthermore, when enough processors are available the dPaster will (for many programs) run *faster* than the SUN Modula-2 linker `m2l`. This is in spite of the fact that the SUN Modula-2 translating system does not support ZTS-style flexible encapsulation and inter-modular optimization. While this may be an unfair comparison, it does indicate that one can expect ZTS to achieve turn-around times similar to traditional translating systems. This supports the main claim of this thesis, namely that distributed high-level binding is an attractive solution to the problems inherent in modularization and separate compilation.

In addition to the measurements of the sPaster and the dPaster, this chapter makes one important contribution: a model of modular programs. Although this model is still in an embryonic stage, without it we would have been unable to say anything about the behavior of the ZTS for many classes of programs. To be a truly useful tool for other types of applications, however, the model needs to be extended in many ways. First of all, there needs to be much additional research into the structure of real programs. Not until such statistics are available will we be able to select reasonable values for the model parameters. Translating systems such as ZTS that are able to collect source statistics are a step in this direction. It will also be necessary to provide better control over the internal characteristics of modules and procedures. A test data generating system such as Maurer's [145], based on context-free grammars, seems ideal for this purpose. Finally, we believe that the tuple value forms of Table 8.2 are overly simplified. A more reasonable approach would make each form a function of the module's height in the import graph. This would account for our intuition that lower level modules often implement abstract data types (and therefore export many abstract items, particularly abstract types and abstract inline procedures), whereas modules at a higher level are more likely to be abstract state machines and therefore mostly export non-inline abstract procedures.

Chapter 9

Conclusions and Future Research

Q: *How many graduate students does it take to change a light bulb?*
A: *Only one, but it takes 9 years.*

Reported by Jan Yarnot

In this thesis we have pursued two different avenues of research: *language design* for better control over abstraction and encapsulation, and *translation system design* for better flow of inter-modular information in the presence of separate compilation. It should come as no surprise, however, that the pursuit of these two interests has gone hand in hand: the design of Zuse has provided motivation for the design of the zts, whose capabilities in turn have inspired further additions to Zuse.

Although it was the possibility of true *orthogonal encapsulation* that first sparked our interest in pursuing the research presented here, we believe that the spin-off results are of equal interest. In fact, one might, not unreasonably, view the thesis from three different perspectives: that it is all about language design, all about code optimization, or all about efficient translation. We will consider each perspective in turn.

9.1 Language Design

Like others before us, we have strived to design a language of "least surprise," in our case with respect to encapsulation. For example, a beginning Modula-2 programmer is surprised to learn that types may be hidden but not constants,

and he is even more surprised when he is told that the realization of hidden types is restricted to pointers. A beginning Ada programmer is in for similar surprises: the realization of procedures may be hidden, but the realization of types and constants may only be protected. The encapsulation system presented in this thesis is simple and regular and should be met without surprise by beginning programmers: all aspects of all exported items may be hidden or revealed; all exported items may be protected or unprotected.

How do the Zuse encapsulation concepts compare to those found in other languages? We have already mentioned that encapsulation in Modula-2 and Ada is confusing and arbitrary and influenced by implementation concerns more than anything else. Encapsulation in Modula-3 is much more interesting and regular, although basically restricted to object types. Object type encapsulation in Zuse is very similar to the Modula-3 design; in fact, Zuse's object-oriented features are more or less a subset of those found in Modula-3. The main difference is instead the way object type encapsulation is implemented in zts and SRC Modula-3. We will discuss this more thoroughly in the next section.

There is one important difference between Zuse and Modula-3 in regard to object type encapsulation, namely the ability of Modula-3 to present different views of a particular object type to different kinds of clients. This is a result of the Modula-3 many-to-many module system and the Modula-3 type system which uses structural type equivalence. We are currently investigating the possibility of extending Zuse with a multiple view capability, possibly through a module system more general than one-to-one but more restricted than Modula-3's many-to-many, which we find overly flexible and potentially confusing.

The language most closely related to Zuse is Milano-Pascal (Section 2.8.6), which (like Zuse) supports hidden types and scalar constants. There are many differences, however. First of all, Milano-Pascal – unlike all modern modular languages – uses import-by-structure rather than import-by-name. From a purely theoretical point of view import-by-structure appears superior to import-by-name, since it grants modules complete autonomy: no module needs to know about the existence of any other module. Unfortunately, import-by-structure is completely unusable in practice, since it requires each client to give a complete description of all imported items it needs to use. A further difference between Zuse and Milano-Pascal is, of course, Zuse's flexible encapsulation, which extends the rudimentary facilities found in Milano-Pascal with type protection and various forms of semi-abstract export. Furthermore, Milano-Pascal does not support abstract inline procedures, a concept which ought to be an integral part of any language concerned with efficient execution in the presence of strict encapsulation.

One question that is fair to ask at this point is whether the full flexibility afforded by Zuse in regard to encapsulation is really necessary. Could it not be that the wide range of choices available to a programmer might be more a source of confusion than a source of elation? It should be clear that Zuse as it is presented in this thesis is an *experimental* design, intended to extend flexible encapsulation to its limits, not a language engineered towards production programming. We would not be surprised if future research were to show that a simpler bag of encapsulation primitives will be sufficient for most programming situations. It must be made clear, however, that from an implementation perspective full flexible encapsulation as presented here comes at little extra cost compared to a language with scaled down encapsulation, consisting, for example, of abstract types and constants.

9.2 Code Optimization

It might seem that the pursuit of new and more effective code optimization techniques and the invention of new programming language constructs would be two diametrically opposed activities. This is, however, not the case. On the contrary, if new language constructs are to take root and blossom, it is imperative that their conception be followed by the development of efficient and effective translation techniques.[1] Many programmers are reluctant to use concepts which they feel (rightly or not) would seriously hamper the efficient execution of their programs. Therefore, any attempt at introducing new encapsulation concepts had better be accompanied by ample evidence that these concepts are indeed efficiently implementable.

We believe that this thesis has shown that full and flexible encapsulation is useful, desirable, and can be implemented efficiently. Furthermore, we have shown that inter-procedural optimization and data encapsulation are simply different sides of the same coin, and that techniques which support the implementation of one may be used to implement the other. In fact, the evaluation results of Chapter 8 show that a translating system which performs binding-time inter-modular procedure integration might as well support full data encapsulation – the additional binding-time cost is negligible. Procedure integration is in itself a particularly important companion optimization to encapsulation,

[1] Unfortunately, the availability of efficient implementation techniques is no guarantee that new and otherwise interesting language constructs will be immediately embraced by the programming language design community. This is evident from the sad history of CLU's excellent iterator construct, whose introduction was immediately followed by the description of an efficient implementation (Atkinsson [13]). Still, no other systems programming language has to date included a similar concept.

since it nullifies many of the inefficiencies introduced by procedural interfaces to abstractions.

As a matter of fact, the ZTS design guarantees that – in spite of separate compilation – the code produced for Zuse programs will be as efficient as that produced by compilers for monolithic languages. Furthermore, statically allocated abstract types, inter-modular inline expansion, smart linking, and translation-time construction of object type templates guarantee that ZTS will produce *better* code than traditional translating systems for modular and object-oriented languages with strong encapsulation concepts.

It is interesting to compare ZTS to the SRC Modula-3 translating system since, as we noted in the previous section, Zuse and Modula-3 have essentially the same encapsulation features in regard to object-oriented programming. As we described in detail in Section 2.8.9, the run-time system in the SRC Modula-3 implementation is responsible for performing many of the tasks performed at binding-time in ZTS. The result is, of course, that (other things being equal) a Zuse program translated by ZTS will always perform better than the equivalent Modula-3 program translated by the SRC system.

9.3 Translation Efficiency

The introduction of a new programming language construct requires, as we have said, proof that programs which use the construct will execute efficiently. Equally important, however, is to show that there exist translating system techniques such that the construct may be efficiently processed. Programmers may very well spurn an otherwise desirable programming language feature if its use results in unacceptably long turn-around times. We have devoted the second half of this thesis to the presentation of techniques for the efficient processing of modular imperative and object-oriented languages with flexible encapsulation concepts. Our results show that distributed translation techniques produce efficient high-level module binders which support the concepts introduced in Zuse.

What is interesting, however, is that regardless of the translating techniques used, languages with strict encapsulation have an advantage with respect to translation efficiency over "realization protection (Section 2.5)" languages such as Ada: since they require less information to be given in specification units they run a much smaller risk of suffering from trickle-down recompilations. Thus, even a Zuse translating system which sports an unsophisticated sequential module binder and hence may suffer long module binding times may very well do better on the whole than a traditional translating system for Ada, since it

is likely to suffer fewer complete recompilations.

Furthermore, a Zuse translating system with a sophisticated module binder (such as the dPaster) will, as we have seen in Chapter 8, combine a reduced risk of trickle-down recompilations with module binding times which are sometimes *shorter* than what can be expected from traditional link-editors. Zuse and Zts are in essence able to combine the best of several worlds: flexible encapsulation, excellent code quality through inter-modular optimization and static allocation, and fast turn-around-times through distributed binding and fewer complete recompilations.

9.3.1 Sequential vs. Distributed vs. Incremental Translation

One of the implicit long-term goals of the research presented here is to compare the benefits of sequential, distributed, and incremental translation techniques. We are especially interested in the relative performance of these techniques when combined with inter-procedural optimization and when subjected to minor and major source code changes.

The strength of incremental translation tools lies in their superior performance in the presence of small and local changes to the source code. Incremental translation techniques in fact rely on the assumption that local source code changes will have local effects on the object code. This is often true for tools such as syntactic and semantic analyzers, but may not hold true for aggressive optimizers and code generators. For example, a code generator which does inter-procedural register allocation may, in the worst case, need to reprocess the entire program as a result of a small change to the body of a single procedure. The same is true of the expansion of inline procedures: a local change to the body of an inline procedure will affect (at least) all procedures which call it. While distributed tools are more immune to these kinds of worst-case scenarios, they will most likely be outperformed by their incremental counterparts in the cases when local changes only have local effect.

If we apply these reflections to the translating system designs presented in the thesis we might say that the dPaster will have good *worst-case* behavior but poor *best-case* behavior, while the iPaster will have good *best-case* behavior and poor *worst-case* behavior. In other words, when only local changes have been made to a program (the *best case*) the iPaster will perform well, but the dPaster will perform poorly. When changes with global effects have been performed (the *worst case*), however, the opposite will be true. It remains an open question whether it is possible to combine incremental and distributed techniques to efficiently handle situations with local as well as global changes.

9.4 Contributions

The most important contribution of the first half of this thesis is, of course, the flexible encapsulation primitives of Zuse. We find it interesting that it has proven possible to blend type protection with abstract and semi-abstract export to yield a coherent and regular type system.[2] A further interesting (and to us somewhat surprising) discovery has been the wide range of static semantic conditions which cannot be tested at compile-time.

The most novel and in our mind the potentially most profitable contribution of the second half of the thesis is the distributed module binder of Chapter 6. We believe that distributed high-level binding provides a framework not only for the support of data encapsulation in modular programming languages but also for the application of inter-modular optimizations which have hitherto been deemed prohibitively expensive. We will pursue this avenue of research, examining how the dPaster can be extended to support various inter-procedural optimization techniques (e.g. inter-procedural constant propagation and register allocation) as well as the efficient implementation of new language constructs (inline iterators/generators, and code sharing among instances of generic modules). We also intend to investigate different techniques for speeding up the current dPaster implementation, particularly new load balancing algorithms and techniques for less rigorous separation into phases.

Until now, parallel and distributed translation has been met with comparatively little interest, and the number of fully implemented systems is very small. We believe this to be the result of triply misdirected efforts: most practical as well as theoretical work in the field has been geared towards tightly coupled architectures (which are not yet generally available), the compilation of monolithic languages (which are obsolete), and the early phases of translation such as lexical and syntactic analysis (which have very efficient sequential solutions). Furthermore, with the advent of separately compiled modular languages, the amount of parallel work available within a compilation unit is reduced to the extent that the application of coarse-grained parallelism is not warranted.[3] We believe that distributed high-level module binding opens up interesting new areas of application of distributed translation techniques since it is geared towards loosely coupled distributed systems (which are generally available), modular languages (which are in widespread use), and the late phases of translation (which are expensive for sequential architectures).

[2] However, much of the syntax leaves something to be desired. Type protection clauses, for example, have been described as "rather baroque."

[3] The work of Wortman and Junkin [118, 231, 233], however, shows that successful parallel compilation of modular languages on shared memory architectures is possible.

Bibliography

"I've got to think up bigger things.
I'll bet I can, you know.
I'll speed my Thinker-Upper up
As fast as it will go!"

Then BLUNK! Her Thinker-Upper thunked
A double klunker-klunk.
My sister's eyes flew open
And she saw she'd thunked a Glunk!

He was greenish. Not too cleanish.
And he sort of had bad breath.
"Good gracious!" gasped my sister.
"I have thunked up quite a meth!"

Dr. Seuss [193]

[1] Accredited Standards Committee X3, Information Processing Systems, The American National Standards Institute (ANSI), CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001. *Draft Proposed American National Standard for Information Systems — Programming Language C++*, x3j16/91-0115 edition, September 1991. Cited on page 45.

[2] Ada 9X mapping – mapping rationale. ftp from ajpo.sei.cmu.edu, August 1991. Cited on page 45.

[3] Ada 9X project report – Ada 9X requirements. ftp from ajpo.sei.cmu.edu, December 1990. Cited on page 45.

[4] Ada 9X project report – Ada support for software reuse. ftp from ajpo.sei.cmu.edu, October 1990. Cited on page 45.

[5] Rolf Adams, Annette Weinert, and Walter Tichy. Software change analysis *or* Half of all Ada compilations are redundant. In *2nd European Software Engi-*

neering Conference, pages 203–221, Coventry, UK, September 1989. Cited on page 19.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6. Cited on pages 133 and 149.

[7] F. E. Allen, J. L. Carter, J. Fabri, J. Ferrante, W. H. Harrison, P. G. Loewner, and L. H. Trevillyan. The experimental compiling system. *IBM Journal of Research and Development*, 24(6):695–715, November 1980. Cited on page 138.

[8] Bowen Alpern, Roger Hoover, Barry K. Rosen, and Peter F. Sweeney F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *First Annual acm-siam Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, January 1990. Cited on page 216.

[9] Allen A. Ambler and Charles G. Hoch. A study of protection in programming languages. Technical Report ICSCA-CMP-3, University of Texas at Austin, 1976. Cited on page 8.

[10] Allen L. Ambler, Donald I Good, James C. Brown, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, March 1977. ACM Conference on Language Design for Reliable Software. Cited on page 62.

[11] M. Ancona, L. De Floriani, D. Dodero, and P. Thea. Program development by using a source linker. In *4th Jerusalem Conference on Information technology (JCIT). Next Decade in Information Technology*, pages 251–259, Jerusalem, Israel, May 1984. Cited on page 31.

[12] James E. Archer and Michael T. Devlin. Rational's experience using Ada for very large systems. In *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages 251–259, Houston, Texas, USA, June 1986. Cited on pages 33 and 209.

[13] Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler. Aspects of implementing CLU. *Proceedings ACM National Conference*, pages 123–129, December 1978. Cited on pages 15, 45, 152, 155, and 253.

[14] Erik H. Baalbergen. Parallel and distributed compilations in loosely-coupled systems: A case study. In *Proc. Workshop on Large Grain Parallelism*, 1986. Cited on pages 27 and 166.

[15] Erik H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1:135–158, 1988. Cited on pages 27, 166, and 168.

[16] Erik H. Baalbergen, Kees Verstoep, and Andrew S. Tanenbaum. On the design of the amoeba configuration manager. *ACM SIGSOFT Software Engineering Notes*, 17, November 1989. Proc. 2nd ACM International Workshop on Software Configuration Management. Cited on pages 27 and 166.

[17] Henri. E. Bal, R. van Renesse, and Andrew. S. Tanenbaum. Implementing distributed algorithms using remote procedure calls. In *Proc. National Computer Conference, AFIPS*, pages 499–505, 1987. Cited on page 164.

[18] J. Eugene Ball. Predicting the effects of optimization on a procedure body. In *Proceedings of the '79 Symposium on Compiler Construction*, pages 214–220, 1979. Cited on page 137.

[19] F. L. Bauer and H. Wössner. The "plankalkül" of Konrad Zuse: A forerunner of today's programming languages. *CACM*, 15(7):678–685, July 1972. Cited on page 48.

[20] Leland L. Beck. *System Software – An Introduction to Systems Programming.* Addison-Wesley, second edition, 1990. ISBN 0-201-50945-8. Cited on pages 29 and 32.

[21] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, Georgia, USA, June 1988. Cited on pages 30 and 135.

[22] Marc Benveniste. Operational semantics of a distributed object-oriented language and its Z formal specification. Publication Interne 532, INRISA/INRIA-Rennes, Rennes Cedex, France, April 1990. Cited on page 88.

[23] R. E. Berry and B. A. E. Meekings. A style analysis of C programs. *CACM*, 28(1):80–88, January 1991. Cited on page 221.

[24] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation – Numerical Methods.* Prentice-Hall, 1989. ISBN 0-13-648759-9. Cited on page 167.

[25] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. Cited on pages 164 and 200.

[26] Mary P. Bivens and Mary Lou Soffa. Incremental register reallocation. *Software-Practice and Experience*, 20(10):1015–1047, October 1990. Cited on page 209.

[27] Dines Björner and Cliff B. Jones. *Formal Specification & Software Development.* Prentice-Hall, 1982. ISBN 0-13-329003-4. Cited on page 87.

[28] Dines Björner and O. N. Oest. *Towards a Formal Description of Ada.* LNCS 98. Springer Verlag, 1980. ISBN 3-540-10283-1. Cited on pages 45 and 87.

[29] Günter Blaschek, Gustav Pomberger, and Alois Stritzinger. A comparison of object-oriented programming languages. *Structured Programming*, 4(10):187–197, 1989. Cited on page 8.

[30] Hans-Juergen Boehm and Willy Zwaenepoel. Parallel attribute grammar evaluation. In R. Popescu-Zeletin, editor, *International Conference on Distributed Computing Systems*, pages 347–354, Berlin, West Germany, September 1987. Cited on pages 168 and 191.

[31] Grady Booch. *Software Engineering with Ada*. The Benjamin/Cummings Publishing Company, 1987. ISBN 0-8053-0604-8. Cited on page 22.

[32] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991. ISBN 0-8053-0091-0. Cited on page 272.

[33] Ellen Ariel Borison. *Program Changes and the Cost of Selective Recompilation*. PhD thesis, Carnegie Mellon University, 1989. Cited on page 221.

[34] Gabriel Bracha and Sam Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127–138, 1987. Cited on page 179.

[35] Paul Branquart, Georges Louis, and Pierre Wodon. *An Analytical Description of CHILL, the CCITT High Level Language*. LNCS 128. Springer Verlag, 1982. ISBN 3-540-11196-4. Cited on page 45.

[36] Gary Bray. Sharing code among instances of Ada generics. *SIGPLAN Notices*, 19(6):276–284, June 1984. ACM Sigplan '84 Symposium on Compiler Construction. Cited on page 152.

[37] British Standards Institution. *Third Working Draft Modula-2 Standard*, d106/n336 edition, October 1989. Cited on pages 45 and 87.

[38] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium On Compiler Construction*. ACM, June 1986. Cited on pages 149, 206, and 210.

[39] Frank W. Calliss. A comparison of module constructs in programming languages. *SIGPLAN Notices*, 26(1):38–46, January 1991. Cited on pages 8 and 27.

[40] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42, August 1992. Cited on page 45.

[41] Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, DEC SRC, November 1989. Cited on pages 4 and 45.

[42] Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, January 1989. Cited on page 45.

[43] Luca Cardelli and Peter Wegener. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985. Cited on pages 8 and 17.

[44] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon, and Scott K. Warren. A practical environment for scientific programming. *IEEE Software*, 20(11):75–89, November 1987. Cited on pages 5 and 209.

[45] A. Celentano, P. Della Vigna, C. Ghezzi, and D. Mandrioli. Modularization of block-structured languages: The case of Pascal. In *Workshop on Reliable Software*, pages 167–79, Bonn, Germany, September 1978. Cited on pages 4, 31, 45, and 159.

[46] A. Celentano, P. Della Vigna, C. Ghezzi, and D. Mandrioli. Separate compilation and partial specification in Pascal. *IEEE Transactions on Software Engineering*, 6:320–328, July 1980. Cited on pages 4, 31, 39, 45, and 159.

[47] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992. Available via anonymous ftp from `self.stanford.edu`. Cited on pages 32 and 156.

[48] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *IEEE COMPCON 1986*, pages 132–137, 1986. Cited on pages 4, 30, 159, 169, and 219.

[49] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, Atlanta, Georgia, USA, June 1988. Cited on pages 149 and 206.

[50] Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, October 1991. Cited on page 151.

[51] Robert F. Cohen and Roberto Tamassia. Dynamic expression trees. Technical Report CS-90-35, Department of Computer Science, Brown University, December 1991. Cited on page 216.

[52] Christian S. Collberg and Magnus G. Krampell. Design and implementation of modular languages supporting information hiding. In *Proceedings of the Sixth International Phoenix Conference on Computers and Communications*, pages 224–228, Scottsdale, AZ, USA, February 1987. Cited on pages 4 and 159.

[53] Christian S. Collberg and Magnus G. Krampell. A property-based method for selecting among multiple implementations of modules. In *Lecture Notes in Computer Science No. 289*. AFCET, Springer Verlag, February 1987. Cited on page 15.

[54] Reidar Conradi and Dag Heiraas Wanvik. Mechanisms and tools for separate compilation. Technical Report 25/85, Division of Computer Systems and Telematics. University of Trondheim, Norway, 1985. Presented at IFIP WG.2, Maine, Oct 7-11, 1985. Cited on pages 8, 19, and 221.

[55] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. Technical Report TR90-128, Rice University, Houston, Texas, USA, September 1990. Cited on pages 138, 210, and 219.

[56] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software-Practice and Experience*, 21(6), June 1991. Cited on pages 210 and 219.

[57] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986. Cited on pages 5 and 209.

[58] Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilations. In *Proceedings of the SIGPLAN 86 Symposium of Compiler Construction*, June 1986. Cited on pages 5, 209, and 219.

[59] James R. Cordy and Richard C. Holt. Code generation using an orthogonal model. *Software-Practice and Experience*, 20(3):301–320, March 1990. Cited on page 135.

[60] Douglas Campbell Coupland. *Generation X*. Abacus, 1991. ISBN 0-349-10331-3. Cited on page 219.

[61] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989. Cited on page 135.

[62] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Haberman. Software development environments. *IEEE Software*, 20(11):18–28, November 1987. Cited on page 209.

[63] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984. Cited on page 135.

[64] Mark Day, 1991. Personal communication. Cited on page 15.

[65] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 122–127, San Francisco, California, May 1982. Cited on page 151.

[66] Edsger W. Dijkstra. Termination detection for diffusing computation. *Information Processing Letters*, 11(1):1–4, 1980. Cited on page 183.

[67] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., 1990. ISBN 0-471-92489 X. Cited on page 87.

[68] Anders Edenbrandt. RMC - a remote procedure call facility. Unpublished manuscript, Department of Computer Science, Lund University, 1990. Cited on page 200.

[69] R. S. Engelmore and A. J. Morgan. *Blackboard Systems*, chapter 30. Addison-Wesley, 1988. ISBN 0-201-17431-6. Cited on page 192.

[70] M. C. Er. A parallel computation approach to topological sorting. *The Computer Journal*, 26(4):293–295, 1983. Cited on page 192.

[71] Peter H. Feiler, Susan A. Dart, and Grace Downey. Evaluation of the Rational environment. Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, July 1988. Cited on pages 33 and 209.

[72] Michael B. Feldman. *Data Structures with Modula-2*. Prentice-Hall, 1988. ISBN 0-13-197666-4. Cited on page 70.

[73] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255–265, April 1979. Cited on page 27.

[74] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. Cited on page 135.

[75] Ray Ford and Mary Pfreundschuh Wagner. Incremental concurrent builds for modular systems. *Journal of Systems Software*, 13:157–176, 1990. Cited on page 166.

[76] David G. Foster. Separate compilation in a Modula-2 compiler. *Software-Practice and Experience*, 16(2):101–106, February 1986. Cited on page 26.

[77] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980. Cited on page 183.

[78] Christopher W. Fraser. A language for writing code generators. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 238–245, 1989. Cited on page 149.

[79] Mahadevan Ganapathi and Charles N. Fischer. Attributed linear intermediate representations for retargetable code generators. *Software-Practice and Experience*, 14(4):347–364, 1984. Cited on page 131.

[80] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985. Cited on page 162.

[81] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. Early experience with Mesa. *CACM*, 20(8):540–553, August 1977. Cited on page 45.

[82] Carlo Ghezzi, 1991. Personal communication. Cited on page 159.

[83] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, Inc., second edition, 1987. ISBN 0-471-82173-X. Cited on page 47.

[84] Alan Gibbons and Wojciech Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In *Foundations of Software Technology and Theoretical Computer Science, Sixth Conference*, pages 454–467, New Delhi, India, December 1986. Springer Verlag. LNCS 241. Cited on page 192.

[85] Thomas Gross, Angelika Zobel, and Markus Zolg. Parallel compilation for a parallel machine. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 91–99, Portland, Oregon, June 1989. Cited on page 169.

[86] Rajiv Gupta, Lori Pollock, and Mary Lou Soffa. Parallelizing data flow analysis. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990. Cited on page 206.

[87] Jürg Gutknecht. Separate compilation in Modula-2: An approach to efficient symbol files. *IEEE Software*, pages 29–38, November 1986. Cited on pages 25 and 26.

[88] Mary Wolcott Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston, Texas, USA, April 1991. Cited on page 138.

[89] Samuel P. Harbison. *Modula-3*. Prentice Hall, 1992. ISBN 0-13-596369-6. Cited on page 32.

[90] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986. Cited on page 45.

[91] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, March 1986. Cited on page 45.

[92] Görel Hedin. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Sciences, Lund University, Lund, Sweden, March 1992. Cited on page 208.

[93] Jean-Michel Hèlary, Claude Jard, Noël Plouzeau, and Michel Raynal. Detection of stable properties in distributed applications. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 52–63, Vancouver, British Columbia, Canada, August 1987. Cited on pages 179 and 183.

[94] Mark Himelstein, Fred C. Chow, and Kevin Enderby. Cross-module optimizations: Its implementation and benefits. In *Proceedings of the Summer 1987 USENIX Conference*, pages 347–356, June 1987. Cited on pages 4, 30, 137, 138, 159, and 169.

[95] Daniel Hoffman. Practical interface specification. *Software-Practice and Experience*, 19(2):127–148, February 1989. Cited on pages 11 and 17.

[96] Daniel Hoffman and Richard Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 14(9):1243–1255, September 1988. Cited on page 17.

[97] Anne M. Holler. *A Study of the Effects on Subprogram Inlining*. PhD thesis, University of Virginia, Charlottesville, Virginia, USA, March 1991. Computer Science Report No. TR-91-06. Cited on pages 138 and 150.

[98] Richard C. Holt. Data descriptors: A compile-time model of data and addressing. *ACM Transactions on Programming Languages and Systems*, 9(3):367–389, 1987. Cited on page 135.

[99] Richard C. Holt and James R. Cordy. The Turing language report. Technical Report CSRI-153, Computer Systems Research Institute. University of Toronto, August 1986. Cited on page 45.

[100] Richard C. Holt and James R. Cordy. The Turing plus report. Technical Report CSRI-214, Computer Systems Research Institute. University of Toronto, August 1988. Cited on page 45.

[101] Richard C. Holt and Philip A. Matthews. The formal semantics of Turing programs. Technical Report CSRI-182, Computer Systems Research Institute. University of Toronto, May 1986. ISSN 0834-1648. Cited on page 45.

[102] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language. Design and Definition.* Prentice Hall, 1988. ISBN 0-13-933136-0. Cited on page 45.

[103] Richard C. Holt and David B. Wortman. A model for implementing EUCLID modules and prototypes. *ACM Transactions on Programming Languages and Systems*, 4(4):552–562, 1982. Cited on page 45.

[104] Michael Lee Horowitz. *Automatically Achieving Elasticity in the Implementation of Programming Languages.* PhD thesis, Carnegie Mellon University, 1988. Cited on pages 4 and 159.

[105] Shing-Tsaan Huang. Detecting termination of distributed computations by external agents. In *International Conference on Distributed Computing Systems*, pages 79–84, Newport Beach, California, 1989. Cited on page 183.

[106] Jean D. Ichbiah. On the design of Ada. In *Information Processing 83*, pages 1–10. Elsevier Science Publishers B.V. (North Holland), 1983. Cited on page 45.

[107] D. C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford University Press, 1988. ISBN 0-19-859667-7. Cited on page 87.

[108] M. A. Iqbal, J. H. Saltz, and S. H. Bokhari. A comparative analysis of static and dynamic load balancing strategies. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 1040–1047, 1986. Cited on page 196.

[109] Anita K. Jones and Barbara H. Liskov. An access control facility for programming languages. Computation Structures Group Memo 137, MIT, April 1976. Cited on page 61.

[110] Anita K. Jones and Barbara H. Liskov. A language extension for controlling access to shared data. *IEEE Transactions on Software Engineering*, 2(4):277–285, December 1976. Cited on pages 61 and 65.

[111] Anita K. Jones and Barbara H. Liskov. A language extension for expressing constraints on data access. *CACM*, 21(5):358–367, May 1978. Cited on page 61.

[112] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, second edition, 1990. ISBN 0-13-880733-7. Cited on page xi.

[113] Cliff B. Jones and Roger C. F. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall, 1990. ISBN 0-13-116088-5. Cited on pages 87 and 88.

[114] Kevin D. Jones. LM3: A Larch interface language for Modula-3. a definition and introduction. version 1.0. Technical Report 72, Digital Systems Research Center, June 1991. Cited on pages 17 and 45.

[115] Simon L. Peyton Jones. *Parallel Graph Reduction*, chapter 24. Prentice-Hall, 1987. ISBN 0-13-453325-9. Cited on page 191.

[116] Mick Jordan. An extensible programming environment for Modula-3. *SIGSOFT '90*, 15(6):66–76, December 1990. Cited on page 45.

[117] Martin Jourdan. A survey of parallel attribute evaluation methods. In *Attribute Grammars, Applications and Systems*, pages 234–255, June 1991. LNCS 545. Cited on page 191.

[118] Michael D. Junkin and David B. Wortman. The implementation of a concurrent compiler. Technical Report CSRI-235, Computer Systems Research Institute. University of Toronto, December 1990. Cited on pages 45, 169, and 256.

[119] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, pages 58–67, November 1987. Cited on pages 167 and 209.

[120] Simon M. Kaplan and Gail E. Kaiser. Incremental attribute evaluation in distributed language-based environments. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 121–130, New York, USA, 1986. Cited on pages 167, 191, and 209.

[121] Howard P. Katseff. Using data partitioning to implement a parallel assembler. *SIGPLAN Notices*, 23(9):66–76, September 1988. ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages, and Systems. Cited on page 169.

[122] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report TR 91-11-04, University of Washington, Seattle, WA 98195, USA, November 1991. Available via anonymous ftp from `cs.washington.edu` (128.95.1.4) in `pub/pardo/rtcg-case.ps.Z`. Cited on pages 32 and 156.

[123] S. Khanna and A. Ghafoor. A data partitioning technique for parallel compilation. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990. Cited on page 166.

[124] R.B. Kieburtz, W. Barabash, and C.R. Hill. A type-checking program linkage system for Pascal. In *3rd International Conference on Software Engineering*, pages 23–28, Atlanta, GA, USA, May 1978. Cited on page 31.

[125] Eduard F. Klein. Attribute evaluation in parallel. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990. Cited on page 191.

[126] Eduard F. Klein and Kai Koskimies. Parallel one-pass compilation. In *International Workshop on Attribute Grammars and their Applications (WAGA '90)*, Paris, France, September 1990. LNCS 461. Cited on page 191.

[127] Jørgen Lindskov Knudsen, Ole Lehrmann Madsen, Claus Nørgaard, Lars Bak Petersen, and Elmer Sandvad. An overview of the Mjølner BETA system. Technical report, Mjølner Informatics ApS, Science Park Aarhus, Gustav Wiedersvej 10, DK-8000 Århus C, Denmark, March 1990. Cited on pages 35 and 45.

[128] Donald E. Knuth. An empirical study of FORTRAN programs. *Software-Practice and Experience*, 1:105–133, 1971. Cited on page 221.

[129] Peter Kornerup, Bent Bruun Kristensen, and Ole Lehrmann Madsen. Interpretation and code generation based on intermediate languages. *Software-Practice and Experience*, 10:635–658, 1980. Cited on page 131.

[130] Magnus Krampell. Information Hiding: Design and implementation of modular programming languages. Licentiate Thesis LUNFD6/NFCS-1004/1-48/1987, Department of Computer Sciences, Lund University, 1987. Cited on page v.

[131] Antoni Kreczmar, Andrzej Salwicki, and Marek Warpechowski. *LOGLAN '88 – Report on the Programming Language*. LNCS 414. Springer Verlag, 1990. Cited on page 45.

[132] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. Object oriented programming in the Beta programming language. Technical Report 32, Datalogisk Afdeling, Aarhus Universitet, January 1990. Cited on page 45.

[133] Matthijs F. Kuiper. *Parallel Attribute Evaluation*. PhD thesis, University of Utrecht, Padualaan 14, P.O. Box 80.089, Utrecht, The Netherlands, November 1989. Cited on page 191.

[134] Matthijs F. Kuiper and S. Doaitse Swierstra. Parallel attribute evaluation: Structure of evaluators and detection of parallelism. In *International Workshop on Attribute Grammars and their Applications (WAGA '90)*, pages 61–75, Paris, France, September 1990. LNCS 461. Cited on page 191.

[135] B. W. Lampson, J.J Horning, R.L. London, J. G. Mitchell, and G.J. Popek. Report on the programming language Euclid. *SIGPLAN Notices*, (0):1–79, February 1977. Cited on pages 37 and 45.

[136] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–31, January 1984. Cited on page 275.

[137] David B. Leblang and Robert P. Chase Jr. Parallel software configuration management in a network environment. *IEEE Software*, pages 28–35, November 1987. Cited on page 166.

[138] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *Computing Surveys*, 22(4):321–374, December 1990. Cited on page 162.

[139] Mark A. Linton and Russel W. Quong. A macroscopic profile of program compilation and linking. *IEEE Transactions on Software Engineering*, 15(4):427–436, April 1989. Cited on pages 5, 210, and 221.

[140] Barbara H. Liskov, Russell R. Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. LNCS 114. Springer Verlag, 1981. ISBN 3-540-10836-X. Cited on pages 15, 45, and 152.

[141] Barbara H. Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986. ISBN 0-262-12112-3. Cited on page 8.

[142] Barbara H. Liskov, A. Snyder, and Russell R. Atkinson. Abstraction mechanisms in CLU. *CACM*, 20(8):564–576, August 1977. Cited on pages 15, 45, and 152.

[143] M. D. Maclaren. Inline routines in VAXLN Pascal. *SIGPLAN Notices*, 19(6):266–275, June 1984. ACM Sigplan '84 Symposium on Compiler Construction. Cited on page 137.

[144] Friedermann Mattern. Experience with a new distributed termination detection algorithm. In *2nd International Workshop on Distributed Algorithms*, pages 127–143, Amsterdam, The Netherlands, July 1987. LNCS 312. Cited on page 183.

[145] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, pages 50–55, July 1990. Cited on page 250.

[146] Scott McFarling. Procedure merging with instruction caches. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 71–79, Toronto, Canada, June 1991. Cited on page 137.

[147] Brendan D. McKay. On the shape of a random acyclic digraph. *Mathematical Proceedings of the Cambridge Philosophical Society*, (106):459–465, 1989. Cited on pages 203 and 204.

[148] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer Verlag, 1984. ISBN 3-540-13641-X. Cited on page 188.

[149] Wen mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 246–255, Portland, Oregon, June 1989. Cited on page 138.

[150] Bertrand Meyer. Genericity versus inheritance. In *OOPSLA '86*, pages 391–405, September 1986. Cited on page 8.

[151] Bertrand Meyer. *Object Oriented Software Construction*. Addison-Wesley, 1988. ISBN 0-13-629049-3. Cited on pages 10, 11, 17, 36, and 45.

[152] Bertrand Meyer. From structured programming to object-oriented design: The road to Eiffel. *Structured Programming*, (1):19–39, 1989. Cited on page 45.

[153] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Addison-Wesley, 1990. ISBN 0-13-498510-9. Cited on pages 87 and 110.

[154] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992. ISBN 0-13-247925-7. Cited on page 45.

[155] Mjølner Informatics ApS, Science Park Aarhus, Gustav Wiedersvej 10, DK-8000 Århus C, Denmark. *The Mjølner BETA Compiler. Reference Manual*, mia-90-2(0.2) edition, September 1990. Cited on page 45.

[156] Mjølner Informatics ApS, Science Park Aarhus, Gustav Wiedersvej 10, DK-8000 Århus C, Denmark. *The Mjølner BETA Fragment System. Reference Manual*, mia-90-3(0.2) edition, October 1990. Cited on page 45.

[157] Hanspeter Mössenböck and Josef Templ. Object Oberon – a modest object-oriented language. *Structured Programming*, 10(4):199–207, 1989. Cited on page 45.

[158] Hanspeter Mössenböck and Niklaus Wirth. Differences between Oberon and Oberon-2. Technical report, Institute für Computersysteme, ETH, Zürich, 1991. Cited on page 45.

[159] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. Technical report, Institute für Computersysteme, ETH, Zürich, 1991. Cited on page 45.

[160] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991. ISBN 0-13-590464-1. Cited on pages 7, 41, 45, and 72.

[161] David Notkin. The Gandalf project. *The Journal of Systems and Software*, 5:91–105, 1985. Cited on pages 33 and 209.

[162] Stephen M. Omohundro. The Sather language. Technical report, International Computer Science Institute, 1947 Center Street, Suite 600, Berkely, California 94704, 1991. Cited on page 45.

[163] Jukka Paakki, Anssi Karhinen, and Tomi Silander. Orthogonal type extensions and reductions. *SIGPLAN Notices*, 25(7):28–38, July 1990. Cited on pages 45 and 155.

[164] M. Paganini, A. Sacco, L. Tartaglino, and A. Tazzari. A linker for a modularized Pascal[1] Technical report, Dipartimento di Elettronica, Politecnico di Milano, 1978. (in Italian). Cited on pages 45 and 159.

[165] David L. Parnas. Information distribution aspects of design methodology. In *Information Processing 71*, pages 339–344. North Holland Publishing Company, 1972. Cited on page 12.

[166] David L. Parnas. A technique for software module specification with examples. *CACM*, 15(5):330–336, May 1972. Cited on page 9.

[167] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance*. The Bodley Head Ltd., 1974. ISBN 0-552-10166-4. Cited on page 7.

[1] This work has not been available for dissemination.

[168] Lori L. Pollock and Mary Lou Soffa. Incremental global optimization for faster recompilation. In *IEEE 1990 International Conference on Computer Languages*, pages 281–290, New Orleans, Louisiana, 1990. Cited on pages 149, 169, 209, and 219.

[169] Lori L. Pollock and Mary Lou Soffa. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems*, 14(2):173–200, April 1992. Cited on page 209.

[170] Leon Presser and John R. White. Linkers and loaders. *Computing Surveys*, 4(3):149–167, September 1972. Cited on page 29.

[171] Russel W. Quong. *The Design and Implementation of an Incremental Linker.* PhD thesis, Stanford University, May 1989. Cited on page 209.

[172] Russel W. Quong. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems*, 13(1):1–20, January 1991. Cited on pages 5, 210, and 214.

[173] Eric S. Raymond, editor. *The New Hacker's Dictionary.* The MIT Press, 1992. ISBN 0-262-68069-6. Cited on page 47.

[174] Stephen A. Rees and James P. Black. An experimental investigation of distributed matrix multiplication techniques. *Software-Practice and Experience*, 21(10):1041–1063, October 1991. Cited on page 162.

[175] Thomas Reps and Tim Teitelbaum. Language processing in program editors. *IEEE Software*, 20(11):29–40, November 1987. Cited on page 209.

[176] Stephen Richardson and Mahadevan Ganapathi. Code optimization across procedures. *Computer*, pages 42–49, February 1989. Cited on pages 206 and 219.

[177] Stephen Richardson and Mahadevan Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, pages 137–42, August 1989. Cited on page 138.

[178] Stephen Richardson and Mahadevan Ganapathi. Interprocedural optimization: Experimental results. *Software-Practice and Experience*, 19(2):149–169, February 1989. Cited on page 219.

[179] M. W. Rogers, editor. *Ada: Language, Compilers and Bibliography.* Cambridge, 1984. ISBN 0-521-26464-2. Cited on pages 33 and 45.

[180] Jonathan Rosenberg. *Generating Compact Code for Generic Subprograms.* PhD thesis, Carnegie-Mellon University, 1983. Cited on page 152.

[181] Donald L. Ross. Classifying Ada packages. *SIGADA Letters*, 6(4):46–57, July 1986. Cited on pages 22 and 24.

[182] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, pages 46–57, November 1986. Cited on page 45.

[183] A. H. J. Sale. Optimization across module boundaries. *The Australian Computer Journal*, 19(3):167–173, August 1987. Cited on pages 4, 138, and 159.

[184] A. H. J. Sale, 1991. Personal communication. Cited on page 159.

[185] Donald Sannella. Formal specification of ML programs. Technical Report ECS-LFCS-86-15, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986. Cited on page 45.

[186] Vats Santhanan and Daryl Odnert. Register allocation across procedure and module boundaries. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39, White Plains, New York, June 1990. Cited on pages 30 and 169.

[187] M. Satyanarayanan. A survey of distributed file systems. *Annual Reviews of Computer Science*, (4):73–104, 1990. Cited on page 162.

[188] Robert W. Scheifler. The analysis of inline substitution for a structured programming language. *CACM*, 20(9):647–654, September 1977. Cited on pages 137 and 138.

[189] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development.* Wm. C. Brown Publishers, 1986. ISBN 0-697-06849-8. Cited on page 87.

[190] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988. Cited on pages 27 and 219.

[191] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental compilation in Magpie. *SIGPLAN Notices*, 19(6):122–131, June 1984. ACM Sigplan '84 Symposium on Compiler Construction. Cited on page 209.

[192] Robert W. Sebesta. *Concepts of Programming Languages.* The Benjamin Cummings Publishing Company, Inc, 1989. ISBN 0-8053-7011-0. Cited on page 18.

[193] Dr. Seuss. *I can Lick 30 Tigers Today! And Other Stories.* Collins, 1969. ISBN 0-00-195353-2. Cited on page 257.

[194] Mary Shaw, editor. *ALPHARD: Form and Content.* Springer Verlag, 1981. ISBN 0-387-90663-0. Cited on pages 18, 45, 62, and 152.

[195] Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *CACM*, pages 553–564, 1977. Cited on page 152.

[196] Mukesh Singhal. Deadlock detection in distributed systems. *COMPUTER*, pages 37–48, 1989. Cited on page 179.

[197] C. H. Smedema, P. Medema, and M. Boasson. *The Programming Languages Pascal, Modula, CHILL, Ada.* Prentice-Hall, 1983. ISBN 0-13-729756-4. Cited on page 45.

[198] Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research directions in object-oriented programming*, pages 564–576. MIT Press, 1987. ISBN 0-262-19264-0. Cited on page 71.

[199] J. M. Spivey. *The fuzz Manual*, 1988. Cited on page 87.

[200] J. M. Spivey. *Understanding Z.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988. ISBN 0-521-33429-2. Cited on pages 87 and 88.

[201] J. M. Spivey. *The Z Notation – A Reference Manual.* Addison-Wesley, 1989. ISBN 0-13-983768-X. Cited on page 87.

[202] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 1.3.9 edition, January 1991. Cited on page 135.

[203] Bjarne Stroustrup, 1988. Personal communication reported in Booch [32]. Cited on page 10.

[204] Richard E. Sweet. The Mesa programming environment. In *SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 216–229, Seattle, Washington, June 1985. ACM. Cited on pages 30 and 45.

[205] R. B. Tan and J. van Leeuwen. General symmetric distributed termination detection. Technical Report RUU-CS-86-2, Vakgroep Informatica, Rijksuniveriteit Utrecht, January 1982. Cited on page 183.

[206] Lars-Erik Thorelli. A linker allowing hierarchic composition of programs. In *Information Processing 83, IFIP.* Elsevier Science Publishers B.V., 1983. Cited on page 31.

[207] Lars-Erik Thorelli. A language for linking modules into systems. *BIT*, 25:358–378, 1985. Cited on page 31.

[208] Lars-Erik Thorelli. Modules and type checking in PL/LL. In *OOPSLA*, pages 268–274, October 1987. Cited on page 31.

[209] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986. Cited on pages 27 and 219.

[210] Mads Tofte. Four lectures on Standard ML. Technical Report ECS-LFCS-89-73, Laboratory for Foundations of Computer Science, March 1989. Cited on page 45.

[211] S. Vankatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, April 1989. Cited on page 183.

[212] Jeffrey Scott Vitter and Philippe Flajolet. *Average-Case Analysis of Algorithms and Data Structures*, chapter 9. Elsevier Science Publishers B.V., 1990. ISBN 0-444-88075-5. Cited on page 204.

[213] Kim Walden. Automatic generation of Make dependencies. *Software-Practice and Experience*, 14(6):575–585, June 1984. Cited on page 27.

[214] David W. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 264–275, New York, jul 1986. Cited on pages 30, 149, 159, 169, and 209.

[215] David W. Wall. Register windows vs. register allocation. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 67–78, Atlanta, Georgia, USA, June 1988. Cited on page 169.

[216] David W. Wall. Global register allocation at link time. Research Report 17, Digital Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, California 94301, USA, September 1989. Cited on pages 30, 149, 159, 169, and 209.

[217] David W. Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems*, 14(3):299–338, July 1992. Cited on pages 30, 149, 159, 169, and 209.

[218] David W. Wall and Michael L. Powell. The Mahler experience: Using an intermediate language as the machine description. Research Report 1, Digital Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, California 94301, USA, September 1987. Cited on pages 30 and 169.

[219] Mark N. Wegman and Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991. Cited on page 135.

[220] Peter Wegner. Learning the language. *BYTE*, pages 245–253, March 1989. Cited on page 12.

[221] Alan L. Wendt. Fast code generation using automatically-generated decision trees. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 9–15, White Plains, New York, June 1990. Cited on page 149.

[222] Thomas R. Wilcox and Howard J. Larsen. The interactive and incremental compilation of Ada using Diana. Technical report, Rational, Mountain View, California, USA, 1985. Cited on pages 33 and 209.

[223] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, second edition, 1983. Cited on page 45.

[224] Niklaus Wirth. From Modula to Oberon. *Software-Practice and Experience*, 18(7):661–670, July 1988. Cited on page 45.

[225] Niklaus Wirth. The programming language Oberon. *Software-Practice and Experience*, 18(7):671–690, July 1988. Cited on pages 4 and 45.

[226] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988. Cited on pages 45 and 81.

[227] Niklaus Wirth. The programming language Oberon. Technical Report 111, ETH, September 1989. Cited on page 45.

[228] Niklaus Wirth and Jürg Gutknecht. The Oberon system. *Software-Practice and Experience*, 19(9):857–893, September 1989. Cited on pages 32 and 45.

[229] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. A model of visibility control. *IEEE Transactions on Software Engineering*, 14(4):512–520, April 1988. Cited on pages 62 and 77.

[230] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics*. Pitman, 1988. ISBN 0-273-02673-9. Cited on page 87.

[231] David B. Wortman. A concurrent Modula-2+ compiler. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990. Cited on pages 45, 169, and 256.

[232] David B. Wortman and James R. Cordy. Early experiences with Euclid. In *5th International Conference on Software Engineering*, pages 27–32, San Diego, California, USA, March 1981. Cited on page 45.

[233] David B. Wortman and Michael D. Junkin. A concurrent compiler for Modula-2+. In *SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 68–81, 1992. Cited on pages 45, 169, and 256.

[234] W. A. Wulf. Abstract data types: A retrospective and prospective view. In *Proceedings of the 9th Symposium on the Mathematical Foundations of Computer Science*, pages 95–112, Rydzyna, Poland, September 1980. Springer Verlag. Cited on page 8.

[235] Xerox Corporation, 3450 Hillside Avenue, Palo Alto, CA 94304. *Mesa Language Manual*, xde-3.0-3001 edition, November 1984. Cited on pages 4, 30, 45, and 61.

[236] Daniel M. Yellin. Speeding up dynamic dynamic transitive closure for bounded degree graphs. Technical Report RC 13535, IBM T. J. Watson Research Center, October 1990. Cited on page 216.

[237] Daniel M. Yellin and Robert E. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, April 1991. Cited on page 208.

[238] Ignacio Trejos Zelaya. An experimental language definition. Master's thesis, Oxford University, 8-10, Keble Road, Oxford, OX1 3QD, United Kingdom, September 1990. Cited on pages 45 and 88.

[239] Konrad Zuse. Der plankalkül. In *Berichte der Gesellschaft Für Matematik und Datenverarbeitung*, volume 63. 1972. Cited on pages 48 and 121.

Index

The best index to a person's character is

a) *how he treats people who can't do him any good and*
b) *how he treats people who can't fight back.*

Abigail Van Buren

Neither abstraction nor simplicity is a substitute for getting it right.

Butler Lampson [136]