Incremental Code Generation in a Distributed Integrated Programming Environment

Michael James McCarthy

Thesis submitted for the degree of
Doctor of Philosophy
in
The University of Adelaide
(Department of Computer Science)

December 1995

To Julia,
who does nothing
incrementally.

Table of Contents

Table of Figures

Abstract

An incremental compiler recompiles a program, after an edit, in time proportional to the size of the edit; this incremental recompilation will include incremental generation of object code for the program. This thesis presents a new method for performing incremental code generation in a distributed integrated programming environment. A prototype implementation of such an incremental code generator is also described.

Retargetable instruction selectors are generated from a formal specification of the architecture of the target computer. This thesis derives a new retargetable incremental instruction selection algorithm from a non-incremental instruction selection technique in the framework of a precise model of the underlying program representation. The resulting algorithm incrementally regenerates locally optimal object code after the replacement of a subtree in an abstract syntax tree program representation.

A greedy incremental code generator recompiles the updated program immediately after an edit. The impact on the response time of the environment is reduced if the incremental code generator executes in parallel with editing. An efficient greedy parallel incremental code generation algorithm is derived from the retargetable incremental instruction selection algorithm; the derivation of the parallel algorithm is also based on an analysis of the propagation of static semantic information after the replacement of an arbitrary subtree in the abstract syntax tree.

The integration of a greedy incremental code generator into the MultiView distributed integrated programming environment is demonstrated via a prototype implementation. An instance of this environment for a particular programming language is generated from a formal specification of the language and the target computer architecture on which the programs in the language are to be executed. Experimental data gathered from the prototype verifies that the recompilation time after a program update depends approximately linearly on the number of recompiled nodes.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Michael J. McCarthy

June 16, 1995

Acknowledgements

I am indebted to my supervisor Chris Marlin who has encouraged and advised me on the tortuous road to the completion of this work. Without his support the research would never have been done. The quality of this thesis has been immeasurably improved by his red pen.

Thanks are also owed to Paul Calder for his advice and encouragement, to Jenny Harvey for suggesting that I rewrite Chapter Three, to Todd Rockoff for keeping me honest, and to David Jacobs for the loan of his dictionary.

I am grateful to the Department of Computer Science at The Flinders University of South Australia for providing the research facilities that I have used over the last few years.

Chapter 1

Introduction

The power of computer systems has increased dramatically over the last three or four decades, while the cost of these systems has dropped sharply. Over the same period of time, the range and complexity of applications for these computer systems has expanded considerably. However, the development and maintenance of the software that is critical to all computer systems remains a difficult and expensive task.

Many approaches to software development have been proposed in an attempt to reconcile the difficulty of software development with the demand for the timely delivery of safe and reliable software. Various models of the software life-cycle have been proposed, in which specific activities are codified. Such activities include

- requirements analysis,
- software design,
- configuration management and version control,
- implementation and unit testing,
- system integration, and
- maintenance.

Consideration of these activities has led to the provision of a range of tools for the software developer; these tools typically provide explicit assistance to perform a particular activity. Such a tool can be regarded as effective if it reduces the time taken to complete the activity and increases confidence in the resulting artifact.

The artifacts produced during implementation and testing are debugged programs. These programs are usually written in some high-level language. An effective tool to aid this activity must reduce the time taken to develop the code and provide a helpful environment in which this code can be executed and debugged. Implementation requires the repeated modification, recompilation and execution of the same program. During

the development of a single piece of source code, a programmer typically iterates over an edit-compile-execute cycle many times. After editing, the programmer will usually wish to execute the program, in order to evaluate and test any modifications. However, a delay typically occurs while the program source is recompiled into a new version of the executable object code. For software systems of the size and complexity required by many current applications, this delay may be a significant detriment to the productivity of the software developer.

The preparation time is defined to be the duration of the compilation part of the edit-compile-execute cycle; it is thus the delay between the initiation of compilation and the time at which the program is ready for execution. During this time period, the programmer is likely to be idle, waiting to be able to test and evaluate the most recent changes or additions before making any further changes or additions. Thus, systematic reduction of preparation times that cause perceptible delays will improve the productivity of software developers engaged in the implementation phase of the software life-cycle.

Incremental compilation reduces preparation time by recompiling only those parts of the program that have been affected by editing of the source code. If recompilation of affected parts of the program occurs immediately after the program source has been modified, but before execution is requested, then further reductions in the preparation time are possible. Indeed, many of the required recompilations may have occured before the programmer requests execution. Unfortunately, for reasons to be explained later in this thesis, such *greedy* recompilation can have an adverse affect on the response time of the program editor. However, if incremental recompilation takes place in parallel with editing, the preparation time can be reduced, with little effect on the response time of the editor.

Fast recompilation can be exploited to implement functionality that enhances the execution and debugging facilities of a programming environment. In particular, an incremental compiler is able to rapidly insert and remove fragments of object code in the compiled program. This ability can be exploited in the implementation of a range of facilities:

- the implementation of all the usual functionality of a source level debugger,
- the provision of sophisticated debugging operations and constraint checking, and
- instrumentation and monitoring of the executing program.

The implementation of any compiler is a considerable undertaking. The use of a code generator generator to translate a formal description of an instruction set into the architecture-dependent components of the code generator eases both the initial implementation and any subsequent retargeting of the compiler to a different computer architecture. Similarly, the implementation of an incremental code generator is a considerable undertaking. Ideally, the implementor of a programming environment employing incremental recompilation should have a tool which translates a formal specification of the source language and a formal architecture description into the architecture-dependent components of the incremental code generator, thus assisting both the initial implementation and any subsequent retargeting of the environment.

1.1 Contributions of this thesis

This thesis shows that an incremental instruction selection algorithm emitting locally-optimal object code can be systematically derived from a non-incremental retargetable instruction selection algorithm. Furthermore, a retargetable incremental code generator based on such an incremental instruction selector can be integrated into a distributed integrated programming environment in such a way that the incremental code generator executes in parallel with program editing.

The derivation of an incremental instruction selection algorithm from an existing non-incremental algorithm requires a precise characterisation of the subject program. This characterisation must encompass notions of program updates and the static semantics of the program source. An algebraic model of abstract syntax and abstract syntax trees is used as a theoretical framework within which to derive the incremental instruction selection algorithm. The model is equipped with a notion of program updates based on subtree replacement, and a notion of program semantics based on attribute grammars.

Bottom-up rewrite system (BURS) theory is applicable to instruction selection. A BURS-based instruction selector chooses locally-optimal object code in two traversals of an intermediate code tree: a bottom-up pattern matching traversal assigns states to each node of the subject tree, and a subsequent top-down traversal of the subject tree emits locally-optimal object code. The architecture-dependent components of such an instruction selector are generated from a formal specification of the instruction set of the target architecture.

If a BURS pattern matcher has assigned states to a subject tree and this tree is subsequently modified, then the state assignment is no longer consistent: new nodes in subject require initial state assignment and some existing nodes will require different state assignments. This thesis analyses the necessary perturbation of the BURS state assignments resulting from the replacement of a subtree in the subject tree.

An abstract syntax tree representation of a program in a typical high level language contains insufficient information for state labelling by a BURS pattern matcher. Previous BURS-based instruction selectors have generated object code from low-level intermediate code. The generation of such intermediate code from an abstract syntax tree is potentially expensive. An efficient transformation of abstract syntax trees into a form of intermediate representation is described. This transformation is based on the algebraic model of abstract syntax tree representations of programs. The transformed tree is suitable for state labelling by a BURS pattern matcher.

The semantic-based incremental transformation of abstract syntax trees and the analysis of BURS state perturbations after subtree replacement leads to the derivation of a BURS-based incremental instruction selection algorithm. Experimentation has shown that this algorithm regenerates locally-optimal object code in time proportional to the size of the new subtree.

Integrated programming environments directly support the edit-compile-execute cycle that is manifested during implementation and unit testing. Such an environment typically provides a language-sensitive program editor and some facilities for program execution and debugging. Program execution facilities can be based on the incremental generation of

executable object code. In a distributed integrated programming environment, editing and code generation may proceed in parallel. Such an architecture exploits any parallelism manifested by the host computer system – at worst, on a uniprocessor host, the code generator exploits the idle time between edits. A parallel incremental code generation algorithm is derived from the BURS-based incremental instruction selection algorithm. The algebraic model of program representations that are based on abstract syntax trees is used to justify the elimination of certain redundant recompilations. An approach to the integration of a code generator based on this algorithm into a distributed integrated programming environment is demonstrated.

The principal contributions of this work are:

- the systematic derivation, in the framework of a precise model of the subject program, of a retargetable incremental instruction selection algorithm from a non-incremental instruction selector,
- the synthesis of an efficient parallel incremental code generation algorithm based on this incremental instruction selection technique, and
- the demonstration of an approach to the integration of a parallel incremental code generator into a distributed integrated programming environment.

1.2 Outline of this thesis

Chapter 2 introduces the basic issues and definitions pertaining to incremental compilation. An enumeration of fundamental design choices characterises the structure of incremental code generators. A discussion of the individual choices, and the interaction between choices, provides insight into incremental compilation. The range of design alternatives is then used as the basis for a survey of existing incremental compilers and integrated environments that include an incremental code generation facility.

Various techniques have been used to provide execution facilities in integrated programming environments. Chapter 3 surveys several integrated programming environments to illustrate a range of architectures and execution paradigms. The MultiView integrated programming environment, which is based on a distributed client-server architecture, is

described in Section 3.2. If an incremental code generator is implemented as a client in the MultiView environment, then it executes in parallel with program editing.

In Chapter 4, an algebraic model of abstract syntax is presented: abstract syntax is defined in terms of heterogeneous algebras and abstract syntax trees as words freely generated from an abstract syntax. The model is equipped with a notion of program semantics that is based on attribute grammars, and a notion of program editing that is based on subtree replacement.

Chapter 5 commences with a survey of retargetable instruction selection techniques. An analysis, within the framework of the abstract syntax model, demonstrates that bottom-up rewrite system (BURS) based instruction selection is an appropriate base for an incremental instruction selection algorithm. An efficient transformation is developed that translates abstract syntax trees that are decorated with static semantic information into a form suitable for state assignment by a BURS pattern matcher. The remainder of Chapter 5 describes a BURS-based incremental instruction selection algorithm that is based on the earlier analysis of BURS and the semantic-based transformation of abstract syntax tree program representations.

In Chapter 6, a greedy parallel incremental code generation algorithm is described. This algorithm relies on the BURS-based instruction selection algorithm of Chapter 5 and a detailed analysis of the propagation of semantic information about the abstract syntax tree after a program update.

A prototype of the MultiView system described in Chapter 3 and the parallel incremental code generator described in Chapter 6 has been implemented. Chapter 7 describes this implementation. Experimental evidence is also presented in Chapter 7 to verify that the incremental code generator recompiles a program after the replacement of a subtree in the abstract syntax tree in time proportional to the size of the new subtree.

Finally, Chapter 8 describes possible future research that arises from the work presented in this thesis.

Chapter 2

Incremental compilation

increment
A small (or infinitesimal) positive or negative change in a variable quantity or function;
incremental
Of or pertaining to an increment or increments; advancing by increments;
incrementalism
Belief in change by degrees;

The Shorter Oxford Dictionary

During software development and maintenance, pieces of code will be repeatedly edited and tested. After modifying fragments of code, the programmer will wish to execute the new, slightly different, program. During debugging, trace statements and extra conditional tests may be inserted. In a conventional environment, these tasks require the complete recompilation of the larger units of code that surround the modified fragments. Then, the resulting object code is linked with project and system libraries. Finally, the new version of the program can be executed. The delay before the program is ready for execution, called the *preparation time*, is adversely affected by the amount of recompilation performed. Furthermore, much of the recompiled code will be unchanged from the previous version. Long preparation time is both frustrating to the programmer and wasteful of one of the most valuable of project resources: the programmer's time. Reducing the preparation time of a software development environment will improve the productivity of programmers.

One approach to reducing preparation time is to avoid recompiling unmodified parts of the code when preparing a new version of the program for execution. The goal of an incremental compiler is to accomplish this task by expending an amount of effort that is proportional to the *size of the change* [Earley72]. Newly generated object code, corresponding to the modified parts of the program source, is merged with the existing object code that corresponds to the unchanged source.

The *size of the change* depends on the context of the changed source code. This notion is formalised as the *extent of recompilation* (see p. 9), which is precisely defined in terms of a particular program representation in Chapter 5.

An incremental compiler can be used for more than just fast recompilation. For example, portions of programs may be instrumented by the selective compilation and insertion of diagnostic code. Object code to verify assertions may also be inserted at strategic locations. Fritzson, in [Fritzson83], describes how an incremental compiler can be used to implement the common features of source-level debuggers; for example, the code to implement conditional break points can be compiled and inserted into the object code. More recently, the DICE system (see Section 2.4.2, p. 30) has been extended as part of investigations into sophisticated debugging tools [Fritzson92, Fritzson94].

The practicality of applying compilation technology to source-level debugging is critically dependent on the speed of recompilation. Debuggers are typically used in a highly interactive manner and long preparation time would limit the usefulness of features that are implemented via compilation. Incremental compilation can provide the necessarily fast recompilation.

This chapter explores the notion of incremental compilation. After establishing the basic concepts, notation and issues, it surveys a number of existing systems that are based on incremental compilation.

2.1 Basic definitions

The *granularity* of an incremental compiler is the smallest syntactic unit that may be individually recompiled. Figure 2.1 shows the source code for a factorial function written in Ada. This function is incorrect: the "2" in the first return statement needs to be replaced with "1". After this modification, a conventional compiler would have to recompile the entire enclosing compilation unit (which might well consist of considerably more than just the function shown). An incremental compiler with procedure-level granularity would recompile the entire function declaration. Statement-level granularity causes

```
...
function FACTORIAL (N: in NATURAL) return NATURAL is
begin
   if N > 0 then
      return N * FACTORIAL (N - 2);
   else
      return 1;
   end if;
end FACTORIAL;
...
```

Figure 2.1 Granularity for incremental compilation.

the highlighted return statement to be recompiled. Expression-level granularity would recompile only the modified part of the expression.

Statement-level and finer grained incremental compilers are usually referred to as *fine-grained.* Procedure-level and coarser grained incremental compilers are referred to as *coarse-grained.*

The *extent* of recompilation is the set of source code fragments that require recompilation after an edit. The nature of such fragments is determined by the granularity of the incremental compiler; the size of the set will depend on the nature of the edit. Given statement-level granularity, changing the value of a constant in a source code expression will produce a recompilation with small extent, namely a single statement. However, the modification of a symbolic constant definition will cause recompilation with an extent that must include all statements that use the symbolic constant. Given procedure-level granularity, the extent of any recompilation will be a (possibly empty) set of procedures.

The dynamic interaction between source editing and recompilation is significant in an environment based on incremental compilation. Ford and Sawamiphakdi in [Ford85] distinguish between *demand-driven* and *greedy* approaches to incremental compilation. A greedy incremental code generator recompiles code after each edit to the source. Demand-driven environments mark the edited fragments of the source, but perform no recompilation until the object code is required, due to either a request for execution or an explicit compile command.

Greedy code generation minimises the preparation time before a program is ready for execution. However, as the programmer refines the program, the same code fragment may be repeatedly recompiled without ever executing. This continual, often useless, recompilation will adversely affect editing response time because the editor must wait until recompilation after a change is complete before a subsequent edit is possible. Demand-driven code generation avoids this, but at the expense of a longer preparation time. Hybrid approaches in which recompilation is triggered by editor cursor movements, such as movement out of a procedure body, are a useful compromise. The LOIPE environment (see Section 2.4.3, p. 35) is such a hybrid; recompilation is triggered by the motion of the editing cursor out of a modified procedure. Sawamiphakdi, in [Sawamiphakdi84], showed how parallelism can be exploited to attain the low preparation time of greedy code generation with minimal impact on response time (see Section 6.1, particularly Figure 6.1 on p. 147).

2.2 Goals and issues

The *incrementalism condition* for an incremental compiler is that recompilation after a source code change is proportional to the size of the change. Satisfaction of this incrementalism condition requires that, as far as possible, each component of an incremental compiler satisfies the condition.

When designing and evaluating incremental compilers, it is important to have a clear idea of what is required of such a tool. Fritzson, in [Fritzson82], identifies a number of design goals for incremental compilation:

- fast recompilation,
- consistent compilation,
- preservation of execution state,
- low storage overhead,
- good quality of the generated machine code, and
- separability.

Minimal preparation time is one of the prime reasons for the development of systems based on incremental compilation, and so *fast recompilation* is a primary goal. Speed of recompilation will, in general, impact on both the preparation time and the response time of the environment. If response time is poor, then programmers will probably not use the incremental system. If preparation time is poor, then the incremental system is unlikely to have any advantages over a functionally equivalent non-incremental system. Unfortunately, as stated, this criterion of fast recompilation is a little vague for a meaningful comparison of incremental compilation techniques. At best, authors have compared the performance of their incremental systems with the corresponding non-incremental systems. In [Bivens87], such a comparison is made by comparing the time for incremental register reallocation with the time to perform register reallocation for the complete program. In [Fritzson82], a ten times speed up is reported for statement-level incremental recompilation over procedure-level recompilation.

Compilation is *consistent* if code quality does not deteriorate after successive incremental recompilations. For example, new fragments of object code may be merged into the object code by replacing an existing machine instruction with a jump to a block of object code consisting of the new fragment, the instruction that was replaced in the original fragment, and finally a jump back into the main sequence. This is illustrated in Figure 2.2, where the transition from Figure 2.2(a) to Figure 2.2(b) involves the insertion of an assignment statement after the first statement on the left hand side of Figure 2.2(a); the object code is modified by replacing the instruction for the second assignment statement in Figure 2.2(a) by a jump to a block of instructions for the second and third assignment statements in Figure 2.2(b), followed by a jump back to the fourth assignment statement. After successive edit and recompilation cycles, the object code tends to become fragmented into small pieces of object code connected by unconditional jumps. The quality of such code thus degenerates after repeated changes, and such a compiler does not exhibit consistent recompilation.

```
A := 1;
C := 3;
D := 4;
```

1:	mov	1,%l0	*a := 1*
2:	mov	3,%l3	*c := 3*
3:	mov	4,%l4	*d := 4*

(a) Source and object code prior to an edit.

```
A := 1;
B := 2;
C := 3;
D := 4;
```

1:	mov	1,%l0	*a := 1*
2:	jmp	128	*to new segment of code*
3:	mov	4,%l4	*d := 4*

128:	mov	2,%l2	*b := 2*
129:	mov	3,%l3	*c := 3 (replaced by the jump)*
130:	jmp	3	*back to old segment*

(b) Source and object code after the edit.

Figure 2.2 Inconsistent incremental compilation.

A sufficient condition for consistency is that the generated object code is solely a function of the current state of the program source code, and not of the history of modifications leading to this state. That is, to maintain consistent recompilation, the new object code produced by the incremental compiler after an edit should be equivalent to that produced if the entire program was recompiled. Such a criterion requires a notion of the equivalence between two pieces of object code corresponding to the same source code. Absolute equivalence requires binary equivalence between the respective binary representations. Instruction level equivalence requires that the sequences of instructions used be identical, but that register allocation and storage assignments may differ. Fritzson presents a detailed discussion of consistency, along with several notions of equivalence, in [Fritzson84a].

Preservation of execution state is required if the environment is to support on-the-fly modification of code while a running program is suspended at a breakpoint. To preserve execution state, any changes to the program that invalidate the state of a suspended execution are forbidden. For example, allocation of storage for variables whose lifetime extends beyond the breakpoint must either be left unchanged or the execution state must be modified to allow for the new allocation.

Separability implies that code generated by the incremental compiler can be executed independently of the programming environment. Fritzson also observes that it is desirable

to be able to reconnect the environment to a running program.

The quality of code produced by an incremental compiler should be good enough for production use. In any compiler, however, there is a trade-off between code quality and compilation speed. Indeed, most compilers allow the selection of various optimisations when invoked. During program development, programmers will usually select fewer optimisations to gain compilation speed. Any optimisations provided by an incremental system are at the cost of both preparation and response times, and in the complexity of the data structures that must be incrementally manipulated.

The plethora of available computer architectures has led to considerable interest in constructing compilers that may easily be *retargeted* to different instruction sets. Ganapathi [Ganapathi82] presents an early survey of this area; Section 5.1 discusses more recent research into retargetable code generation.

Retargeting of non-incremental compilers may be assisted by the use of code generator generators that, given a machine description, are able to generate most of the components of a code generator that are specific to the machine's instruction set. For example, the TWIG system [Aho89] is able to generate code generators based on fast top-down tree-pattern matchers from a set of tree-patterns annotated with costs and code templates. The GNU C compiler [Stallman94] uses an instruction set description that includes an algebraic formula for each machine instruction.

To date, fine-grained incremental compilers have either used a naive approach to instruction selection or have adapted the code generator from an existing, non-incremental compiler. Retargetability has not been considered as a design goal. However, the investment required to produce an incremental compiler is no less than that required to produce a traditional compiler. Thus, the benefits of retargetability are as significant for incremental compilers as for non-incremental compilers. Consequently, the following are added to the set of possible design goals for an incremental compiler:

- retargetability, and
- the provision of tools to aid retargeting to new architectures.

Of the incremental compilers to be surveyed in Section 2.4, only the DICE system has been retargeted to new architectures. Santi [Santi89] describes the retargeting of DICE from the DEC-10 architecture to the MC68020. This is analogous to the retargeting of the Portable C Compiler (PCC) [Johnson78], as the DICE code generator is based on the code generator from this latter compiler (see Section 2.4.2, p. 30). DICE has also been ported to several other machines: PDP-11, DEC-20, and SPARC [Fritzson95]. The PDP-11 and DEC-20 versions of DICE also used the incremental code generator based on PCC. For the SPARC version, the DICE incremental code generator was rewritten to use the BEG code generator generator system [Emmelmann89]. The BEG-based incremental code generator exhibits similar properties to the PCC-based version: it provides statement-level granularity and register allocation at the statement level.

2.3 Incremental code generation

To a first approximation, compilation may be considered to consist of an analysis phase and a synthesis phase. The program source is read and its meaning, according to the language semantics, is extracted in the analysis phase. The synthesis phase emits object code according to the meaning inferred during analysis. In a non-incremental compiler, the analysis phase, or front-end, performs syntactic and static semantic analysis. Results are passed to the synthesis phase, or back-end, using a combination of an intermediate program representation and data structures that represent the semantic information inferred from the source. An incremental compiler must also perform both analysis and object code synthesis. Satisfaction of the incrementalism condition demands that this analysis is incremental. Incremental analysis is therefore required to incrementally update both the intermediate representation and the semantic data structure that form the input to the synthesis phase.

This work focusses on incremental compilation in integrated programming environments. More specifically, it concentrates on the synthesis, or code generation, phase of incremental compilation. No new results on incremental semantic analysis are presented, but the manner in which the results of such analysis are used is significant.

Several methods have been proposed for performing incremental semantic analysis in programming environments. Since one of the design principles expounded for the programming environment presented in Chapter 3 is language independence, attention is limited to those techniques that generate incremental analysers from a formal specification of the language. This precludes discussion of the hand-coded, language-specific incremental semantic analysers that are used in environments such as SEP [Sawamiphakdi84].

[Reps83] shows how incremental attribute evaluation in a decorated abstract syntax tree may be used for incremental semantic analysis. Incremental evaluators for arbitrary non-circular attribute grammars can be constructed using these techniques. More recently, the Synthesizer Generator [Reps89a] has exploited very efficient incremental attribute evaluators based on visit-sequence evaluators derived from ordered attribute grammar specifications [Kastens80].

Incremental static semantic analysis based on pure attribute grammars has a number of drawbacks. For example, the usual approach to implementing a symbol table involves constructing a list of identifier bindings in a declarative part of the program and propagating this to the use sites as an inherited attribute. Any change to a declaration modifies the symbol table attribute that is propagated to all points of the program in the appropriate scope. Not only can these "copy-chains" be expensive to propagate, but a naive incremental attribute evaluator will probably re-evaluate attributes at the use sites of all identifiers in scope. When coupled with an incremental code generator, unnecessary attribute re-evaluation could easily lead to the unnecessary incremental recompilation of program fragments. More specifically, in a fine-grained incremental compiler, the extent of recompilation after an edit could grow due to the unnecessary attribute re-evaluations, thus violating the incrementalism condition.

Several authors have proposed extensions to pure attribute grammars for use in incremental systems. The Synthesizer Generator allows non-local attribute references, through which the attribute expressions for a node may make use of the attributes of an ancestor node. Algorithms for incrementally maintaining aggregate values, and for restricting attribute propagation when information within an aggregate changes, are presented in

[Hoover86]. Such aggregates are useful for the implementation of symbol tables in which the symbol table is an aggregate of identifier definitions. Horwitz described a technique for using both attributes and relations in a symbiotic fashion for incremental semantic analysis [Horwitz85, Horwitz86]. In this scheme, tuples may be entered into, or removed from, a relation as a consequence of attribute evaluation; attribute equations may include relational expressions. The global nature of the relations allows semantic information to propagate without extended copy-chains of attribute values.

Furthermore, intermodule checking is difficult in systems based purely on attribute grammars. For example, Ada compilation units import context from other compilation units such as packages and, in turn, export information to further compilation units. For any reasonably sized project, it would be impractical to access fully attributed abstract syntax trees that provide the necessary context for all imported compilation units. The memory requirements alone would be crippling. Some mechanism for importing and exporting the externally visible context of modules is required to implement intermodule static semantic checking. The Synthesizer Generator provides an *external store* mechanism [Reps89b] that allows attribute values to be stored in data structures separated from the abstract syntax tree. This external store may be used to extract information from a module in a fashion which allows the information to be imported into another. Horwitz's relationally augmented attribute grammars suggest other techniques for propagating information between modules.

Hedin proposes *door attribute grammars* as a new technique for constructing incremental semantic analysers [Hedin92]. A door attribute grammar allows objects and references to objects as part of the attribution of syntax trees. However, a door attribute grammar is partitioned into a main grammar and a door package. The main grammar can be handled by automatic techniques, but the door package must be manually (but systematically) implemented. Within the framework of door attribute grammars, some of the deficiencies of traditional attribute grammars, when applied to incremental semantic analysis, can be addressed. For example, the intermodule checking problem may be handled by connecting syntax trees through door attributes. Door attribute grammars

have been used to successfully implement an incremental semantic analyser for much of the Simula programming language [Hedin92].

The research described in this thesis focusses exclusively on the synthesis aspects of incremental compilation. In particular, it concentrates on incremental code generation as part of an integrated programming environment. Such environments exploit the sharing of a common program representation by a collection of coupled tools. With a few notable exceptions, such as the relational approach in [Linton84], this shared program representation is an abstract syntax tree. The editor manipulates this tree, the incremental semantic analyser traverses the tree and the incremental code generator is driven by the abstract syntax tree and the semantic information. Consequently, a number of assumptions have been made that limit the scope of the investigation reported in this thesis:

- the source program is available as an abstract syntax tree, and
- the results of incremental semantic analysis are available to the code generator.

No assumptions on the nature of incremental semantic analysis are made. Conversely, the limitations due to a particular semantic analysis technique that may adversely affect incrementalism are tolerated. For instance, expansion of the extent due to unnecessary propagation of attribute evaluation along copy-chains is seen as an issue relating to incremental semantic analysis, rather than a problem in incremental code generation.

2.3.1 Design issues

The structure of an incremental compiler, or an incremental code generation facility for an integrated programming environment, can be considered in terms of a set of key design decisions. These decisions, enumerated as a set of choices, are as follows:

- choice of the granularity of recompilation,
- choice of the canonical program representation,
- choice of the semantic analysis strategy,
- choice of the intermediate representations,
- choice of an instruction selection algorithm,
- choice of a register allocation strategy,

- choice of target-dependent and target-independent optimisations, and
- choice of the editor–code generator interaction strategy.

These decisions are not independent. In some fashion each individual decision will impact on all other decisions. Recompilation granularity constrains the possible resolution of other choices, and has strong implications for the space and time complexity of their implementations. Incremental maintenance of the intermediate representations will depend on both the nature of the canonical representation and the semantic analysis strategy. However, before considering such interactions, it is worthwhile examining each choice in isolation.

Canonical representations

Program development requires the coordinated usage of a number of tools. An incremental code generator will be just one of these tools. At a minimum, it will interact with a program editor, a linker and a loader. It may also interact with symbolic debuggers, metric gathering tools, or even program animation utilities. Many such tools interact in terms of the source code of the program. Each such "source-level" tool requires access to some representation of the program source. The *canonical* program representation is the highest-level representation understood by all tools.

Traditionally, this representation has been the source text of the program. Text is attractive for numerous reasons. It is easily manipulated by standard editors, easily browsed, and may be processed and manipulated by any number of easily obtained tools. If a text based canonical representation is used, then the input to the incremental compiler will be in terms of text editing commands, such as those produced by the Unix `diff` utility. Earley's incremental compilation technique [Earley72] and, more recently, Gafter's proposal [Gafter90], shows how incremental compilers can be developed that accept sequences of textual changes, or "deltas", as input.

The use of text as the canonical program representation in an incremental compiler necessitates the repeated scanning and parsing of the program text. These operations

must also be incremental; otherwise, the incrementalism condition is violated. Structured representations can obviate this textual analysis. Tree-structured canonical representations have been investigated in systems such as the Cornell Program Synthesizer [Teitelbaum81], Mentor [DonzeauGouge80] and, more recently, in Centaur [Borras88], Pan [Ballance90] and the Synthesizer Generator [Reps89a]. Such projects have demonstrated the practicality of providing sophisticated editing environments that manipulate a structured representation of the program source instead of text. Such higher-level representations are attractive because they reduce the amount of replicated processing that must be performed by all of the tools. In particular, providing input to the incremental compiler in terms of tree operations, instead of the more primitive textual operations, avoids the overhead of incremental scanning and parsing of the source text.

The choice of a suitable canonical representation, be it plain text or tree-structured, may well be determined by considerations peripheral to the problem of code generation. This will invariably be the case when attempting to incorporate an incremental code generator into an existing programming environment for which the choice is already made. Whatever representation is chosen, it must be amenable to browsing and manipulation by all the tools in the environment. The specific requirements of one tool should not affect the representation in a manner detrimental to other tools. The diverse nature of tools such as text editors or symbolic debuggers suggests that the canonical representation should provide essentially the same information as the program text, but in a more tractable form. Abstract syntax trees have been a common choice for the structured representation (see Chapter 4, p. 71).

Intermediate representation

The canonical program representation will usually be unsuitable for directly generating object code. An abstract syntax tree alone, for example, does not contain sufficient information to resolve the identifier occurrence that denotes the use of some variable; thus, the generation of object code to access the variable could not proceed without additional information. Non-incremental compilers make use of intermediate representations such as

intermediate code trees (as in the Portable C Compiler [Johnson78]), DAGs, or quadruples [Aho86]. Usually, implementation decisions, such as storage layout, are made prior to the generation of the intermediate representation. While these representations are optimised for the generation of object code, they are unsuitable as a canonical representation. In fact, it is rare that the original program text can be reconstructed from the intermediate representation; certainly, this is not possible if an intermediate form such as quadruples is used. Such a reconstruction from the canonical representation is a precondition to the implementation of fundamental tools such as syntax-directed editors.

Prior to code generation, updates to the canonical program representation must be mapped into updates of the intermediate representation. This mapping must be incremental if the incrementalism constraint is to be satisfied.

Parts of the intermediate representation may be discarded between successive recompilations. It may be sufficient to store only the mapping between source code and object code in order to correctly handle later edits, as in the DICE system (see Section 2.4.2, p. 30). However, the presence of sophisticated incremental register allocation or source-level optimisations requires that additional information be stored. Transforming the canonical form to intermediate code requires knowledge of the program's semantics. Intermediate forms that encapsulate all the semantic information required for code generation are described in [Henry84]. The choice of intermediate representation must include consideration of the data that must persist between successive recompilations, the complexity of the transformation from canonical form to the intermediate representation and, most importantly, the requirements of instruction selection and register allocation.

Semantic analysis

Incrementalism demands that the semantic information required for incremental code generation be inferred incrementally. Thus, data structures encapsulating this information must be maintained incrementally.

In an integrated environment, tools other than the incremental code generator will also need to access the semantically derived information. At the very least, the environment

must provide feedback to the user on static semantic errors. Hence, the corresponding requirements of the environment should also influence the choice of an incremental semantic analysis strategy.

The nature of the languages to be supported is also an issue. Pure incremental attribute evaluation, as used in the prototype described in Chapter 7 of this thesis, is suitable for simple languages. However, as the complexity of the language features to be supported increases, more sophisticated techniques are required. Several alternatives have been cited in Section 2.3.

Optimisations

Production-quality compilers invariably perform optimisations on the program, effectively replacing the program with a functionally equivalent program that is better with respect to some metric such as size or execution speed. Such optimisations include both target-independent optimisations, such as constant folding or loop unrolling, and target-dependent optimisations, such as instruction scheduling. Optimisations performed before code generation operate on the intermediate representation, effectively replacing fragments of intermediate code with new, semantically equivalent fragments better than the original. Optimisations performed after code generation operate on the generated object code. Such optimisations are often performed during a separate phase of compilation or even by a discrete tool, as is often the case with peephole optimisation or instruction scheduling.

The application of any optimisations incurs a cost both in the execution time of the code generator and in the storage overhead for the analysis required for the particular optimisations. To perform optimisations incrementally, sufficient information must be maintained between successive compilations, so that when further changes occur the compiler can check both whether previously applied optimisations remain valid, and whether any new optimisations may be applied. If a previously applied optimisation becomes invalid, it must be undone. When a new optimisation is applied, sufficient information must be kept to enable it to be undone (see p. 42 in Section 2.4.7). Choosing what, if any,

optimisations to apply is a trade off between the quality of the generated code, the speed of incremental code generation, and the storage overhead for the information that must be kept to record the optimisations.

Instruction selection

Code generation can be viewed as the marriage of instruction selection and register allocation. The instruction selection algorithm is responsible for mapping the intermediate representation into the object code that implements the necessary functionality. Good instruction selection makes a significant contribution to object code quality. Non-incremental compilers use techniques such as pattern matching and dynamic programming [Aho89] to perform instruction selection. The cost of such sophisticated techniques in an incremental context must be considered when choosing an incremental instruction selection method. Retargetability as a design goal further restricts the possible choices of an instruction selection method. Chapter 5 discusses, in detail, a number of instruction selection techniques and the adaptation of one such method for incremental instruction selection.

Branch and call instructions require special attention in an incremental environment. By their very nature, such instructions are associated with some destination. However, while a statement that induces a branch instruction in the object code may be unchanged after an edit, the destination of the branch may become invalid. For example, in a Pascal **while** loop, a branch around an iterated statement is required when the while condition becomes false. However, editing the iterated statement will, very probably, alter the number of machine instructions generated for the statement. As a result, the branch destination becomes invalid and must be updated. Thus, the incremental updating of branch instructions is required in an incremental code generator. Furthermore, the destination of branches associated with unchanged statements may require updating.

Register allocation

Accessing variables from processor registers is much faster than accessing the same data from main memory. Thus, the quality of register allocation has a dramatic impact

on the performance of generated code. Good register allocation requires analysis of the register pressure exerted by contiguous fragments of code. Efficient, optimal solutions have proved elusive in non-incremental compilers, and heuristic-based approaches such as the colouring of interference graphs [Chow84, Chaitin82] have been developed. Good incremental register allocation, based on the incremental rebuilding and recolouring of interference graphs, is shown to be possible in [Bivens90] (see Section 2.4.6, p. 40).

Granularity

Choice of recompilation grain size is probably the key choice when designing an incremental compiler. To a first approximation, it would seem that the finer the granularity, the faster the recompilation speed. In fact, Fritzson reports in [Fritzson82] a factor of ten speed-up for statement-level granularity over procedure-level granularity. However, reducing the granularity impacts on most other aspects of the design; in particular, the space requirements for storing the intermediate data structures will increase as the granularity becomes finer.

When changes are made to an identifier declaration, each use site of that identifier must be recompiled. If the granularity of recompilation is coarse, for instance in the case of procedure-level granularity, then each procedure that contains a use of the affected identifier must be recompiled. Potentially, a great deal of redundant recompilation will occur as many procedures may be recompiled. Less redundant recompilation occurs in the case of fine-grained incremental recompilation, as only the affected statements, or parts of statements, are recompiled.

Dynamic interaction

The nature of dynamic interaction between the incremental code generator and the remainder of the environment must be chosen. A greedy approach minimises preparation time, but at the expense of response time. A demand-driven approach minimises response time, but at the expense of preparation time. Possible choices of interaction strategy may be constrained by the architecture of the environment in which the code generator operations. For example, a monolithic, single process architecture precludes the exploitation

of any potential concurrency between code generation and editing. A range of strategies is apparent in the systems surveyed in Section 2.4.

Implications and interactions

The above choices are not independent. Resolution of one choice will restrict the possible resolutions of some, if not all, of the other choices. Particular interactions may also have significant impact on the long-term storage requirements of an incremental compiler. In particular, the choice of granularity has a pervasive effect on each of the other choices. It is worth considering these interactions in more detail.

The canonical representation must be amenable to access and manipulation at the grain size. Consider the situation of when a textual canonical representation is associated with statement-level granularity. Statement boundaries are not clearly delimited in the text and non-trivial processing is required to incrementally locate them. This is apparent in Earley's method (see Section 2.4.1, p. 29), where the syntax of the supported language is heavily restricted to simplify the identification of recompilation grains. Higher-level structured representations, such as abstract syntax trees, are more appropriate for fine-grained incremental compilation. Abstract syntax tree nodes and the associated subtrees, which correspond to recompilation grains, can be easily determined from the label of each node. Locating the recompilation grains by tree traversal and label testing is much faster, and more easily implemented, than the incremental parsing and scanning required when a textual canonical representation is used.

Similarly, the intermediate representation is also somewhat constrained by the grain size. The intermediate representation functions as the interface between the canonical representation and the code generator. As such, it must be efficiently derivable from the canonical representation and directly usable for code generation. It also must be manipulated easily in fragments corresponding to the recompilation grain size.

Register allocation and instruction selection also operate on fragments at the grain size of recompilation. As such, appropriate algorithms are tightly constrained by the granularity. Consider the coarse-grained situation, for example, in a procedure-level incremental

compiler, where it can be guaranteed that basic blocks will not cross grain boundaries; in this situation, standard local register allocation techniques will suffice. However, if the grain size is chosen so that no such guarantees exist, then standard techniques are no longer applicable.

With a coarse grain size, standard instruction selection methods are applicable. The LOIPE prototype (Section 2.4.3, p. 35) demonstrates that the adaptation of an existing compiler can serve as a procedure-level granularity incremental code generator for an integrated environment [Feiler82]. The instruction selection and register allocation algorithms of the base compiler are essentially unchanged. A finer granularity complicates the adoption of existing technology.

Granularity will constrain the possible modes of dynamic interaction between code generation and the remainder of the environment. Procedure-level granularity when associated with greedy recompilation results in frequent recompilation of entire procedures as a programmer creates or modifies a few statements; greedy recompilation is thus not appropriate to coarse-grained systems. However, the Magpie system incorporates procedure-level granularity with a variant of the greedy recompilation strategy (see Section 2.4.4, p. 37). In the hybrid approach, recompilation is triggered by editor cursor movements out of some span of code. Choice of the span size is obviously constrained by recompilation granularity; the span size should be no smaller than the grain size. Statement, statement list or procedure spans would be suitable when a fine-grained incremental code generator is used, but anything less than procedure-size spans would be inappropriate with a coarse grain size. By its very nature, a hybrid approach requires tight coupling between the editor and the code generator, and is applicable primarily within a monolithic integrated environment.

The choice of canonical representation also constrains options available for the remaining choices. Incremental semantic analysis must access the canonical representation to infer the program's static semantics. Some representations are well suited to certain semantic analysis techniques. Incremental attribute evaluation is particularly appropriate when a tree-structured canonical representation is used. In fact, attribute evaluation will

often directly decorate the tree and the resulting stored semantic information could then be considered part of the canonical representation. Conversely, a pure relational representation, such as that considered in [Linton84], is not well suited to attribute-based methods. In this instance, the traversal associated with attribute propagation would necessitate a large number of relational queries.

The intermediate representation is derived from the canonical information and the data inferred by semantic analysis. It is important that the transformation from canonical form to intermediate representation is efficient; otherwise, the overall performance of the system is degraded. Design of the intermediate representation will be driven by the twin requirements for a fast transformation from the canonical representation and the suitability of this representation for code generation. This can be seen in the DICE system, where the decision to base the incremental code generator on the code generator from the Portable C Compiler [Johnson78] required expression trees as the intermediate representation. In DICE, subtrees of the abstract syntax tree corresponding to statements are mapped into expression trees and passed to the incremental code generator. DICE then discards these expression trees after each incremental recompilation.

Choice of a register allocation methodology and the desired target-independent optimisations also place particular requirements on the intermediate representation. Incremental register allocation by graph colouring, as described in [Bivens90], requires that basic blocks and virtual register usage are identified in the intermediate representation (see p. 40 in Section 2.4.6).

Persistence of data

Incremental algorithms transform a solution to an instance of some problem into a solution of a related instance of that problem. Suppose that the initial problem, P, is mapped into the final problem, P', by a transformation τ. The incremental algorithm exploits the solution, S, of P and the details of the transformation, τ, to infer a solution, S', of P'. Given some "size" measures for τ, the incrementalism condition is that the time taken to produce the solution, S', should be proportional to the size of τ.

Ideally only P, P', S and τ should be required to generate S'. However, in many instances, this is inadequate to satisfy the incrementalism condition; some information relating the initial problem, P, to the initial solution, S, may also be required. The amount and nature of such intermediate information is an important aspect of an incremental algorithm.

Focussing on the specific problem of incremental compilation, this additional information relates the object code to the canonical representation of its source. Each of the choices enumerated in the preceding section, and their interactions, will affect the nature and storage overhead of the necessary intermediate information. In turn, the speed of recompilation is affected. Firstly, it is affected by the overhead of the additional processing required to maintain the data structures encapsulating the required information. Secondly, the memory consumed by these data structures can affect the performance of the host system by causing excessive page faulting or cache misses.

The mapping from the canonical program representation into the object code is certainly required. If no such mapping exists, merging new object code fragments with existing object code could not occur and incremental recompilation would not be possible. A location in the object code must be associated with each possible recompilation grain, as any grain may potentially require recompilation. Thus, the number of entries in the mapping from the canonical form to the object code map is greater in a fine-grained environment than in a coarse-grained environment.

The DICE system (see Section 2.4.2, p. 30) demonstrates that it is possible to construct a fine-grained incremental compiler in which a mapping from recompilation grains in the canonical representation (in this case, the statement nodes in an abstract syntax tree) to the object code is the only intermediate information maintained. Not even the intermediate representation persists between successive recompilations.

It is useful to consider why the DICE incremental compiler requires only the source-to-object-code mapping and no other intermediate information. Statements are recompiled by generating their corresponding expression trees. The code generator maps the expression trees into object code; the new object code is then merged into the code file and

the expression tree discarded. Thus, the recompilation of a statement is essentially independent from that of any other statement. This inter-grain independence property of the DICE incremental compiler makes it possible to discard all but the source-to-object-code map between successive recompilations and still satisfy the incrementalism condition.

While inter-grain independence minimises the amount of intermediate information that must be stored, it eliminates the applicability of a number of optimisation and code generation techniques. Bivens' incremental register allocation algorithm [Bivens90], Pollock's incremental optimisation methods [Pollock85] and the incremental instruction algorithm presented in Chapter 5 all require some further intermediate information. It is clear that inter-grain independence levies a penalty on the quality of the generated code.

To summarise, the nature and amount of intermediate information that is required to persist between recompilations is determined by

- the granularity of recompilation, and
- the amount of information that needs to be propagated between recompilation grains.

In turn, the intermediate information requirements play a role in determining the performance of the environment constructed around the incremental compiler. The goal of minimal intermediate information requirements (and therefore small storage demands) must be weighed against the goal of generating high-quality object code.

The storage and processing requirements of the required intermediate information also impinge on the scalability of the environment. For any incremental code generation technique, the storage requirements of the intermediate information increases as the size of the program under development increases. As the amount of information increases, the processing overhead to manage this information increases. Eventually, the performance of the system will degrade beyond an acceptable level. Clearly, a balance must be struck between acceptable resource consumption and the requirements of an incremental code generator.

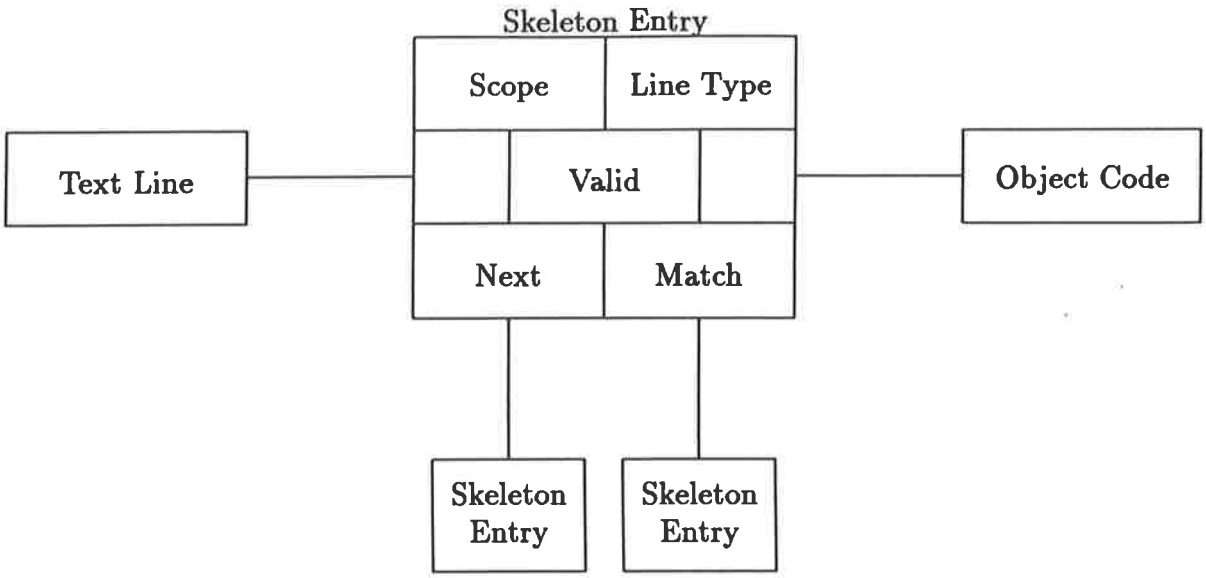


Figure 2.3 Skeletons from Earley's method [Earley72].

2.4 Previous systems

Most research into incremental compilation technology has been carried out in the context of integrated programming environments. This survey focusses on incremental compilation, and environments that include incremental compilers. Thus, significant systems such as the Cornell Program Synthesizer [Teitelbaum81], the Synthesizer Generator [Reps89a], PSG [Bahlke86] and Centaur [Borras88] are omitted, as the execution facilities in these environments are based on interpretation rather than compilation. Instead, attention is paid here to work that is directly concerned with the incremental generation of object code.

2.4.1 Earley's method

[Earley72] outlines an approach to constructing incremental compilers. Source text is used as the canonical representation and intermediate information is maintained in a data structure referred to as the "skeleton". This skeleton includes an entry for each line of source code. A skeleton entry, illustrated in Figure 2.3, identifies the type of the line and contains links to associated information; lines may be either statement lines, declaration lines or bracketing lines. Bracketing lines mirror the block structure of the program.

Granularity	*statement*
Canonical representation	*text*
Semantic analysis	*ad hoc*
Intermediate information	*line-based map of the source text*
Dynamic interaction	*demand-driven*

Figure 2.4 Summary of Earley's method.

After editing is completed, changed lines are analysed to update their skeleton entries and new entries are created for new lines. If changes are only found for statement lines, then those statements are recompiled. Changes to declarations cause recompilation of the entire scope of the declaration. Changes to bracketing lines require analysis to locate the lines at the start and finish of the bracket (using the Match field in Figure 2.3), and may trigger recompilation of the entire program.

A limitation of this approach is that the syntax of the supported language must be tightly constrained, so that the skeleton is both well-defined and amenable to rapid manipulation and updating. Incremental compilers for non-conforming languages cannot directly make use of this method.

Figure 2.4 summarises the key design features of Earley's approach to incremental compilation.[1] Note that this work predates any systematic studies of incremental semantic analysis or incremental parsing.

2.4.2 The DICE system

The DICE system [Fritzson82, Fritzson83, Fritzson84b, Fritzson84c, Santi89] is an integrated programming environment for the Pascal language.[2] DICE is distributed between a host machine, which runs the environment, and a target machine, which contains the executing program, as shown in Figure 2.5. A server process on the target machine

[1] Choices which do not appear to be documented in relevant literature, or are not relevant to the system, are silently omitted in this summary and in the ones that follow.

[2] DICE is an acronym for Distributed Incremental Compiling Environment.

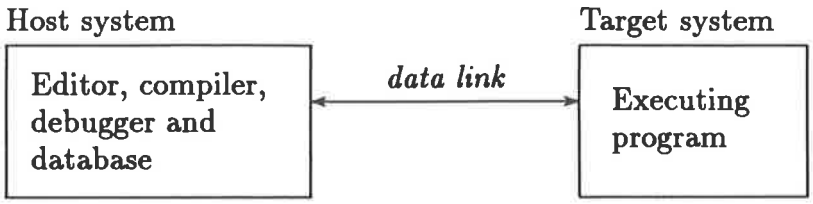


Figure 2.5 The structure of the DICE system.

responds to commands from the host to control the executing program. This system demonstrates the practicality of building integrated environments around fine-grained incremental compilers.

DICE includes an editor, parser, source-level debugger and a fine-grained incremental compiler. These tools are integrated in that they access a single abstract syntax tree representation of the program source and a common program database. DICE is intended to be usable for the construction of substantial programs; thus, the program database must be well ordered [Fritzson82].

The editor is a hybrid text and structure editor. During editing, an *editmark* attribute is maintained for statement nodes. *Editmark* is set for new and modified statement nodes, and left unchanged for existing and unmodified nodes.

The DICE compiler implements recompilation at statement-level granularity. Recompilation is invoked in response to explicit user requests. The structure of the DICE incremental compiler is shown in Figure 2.6. Marked nodes are identified during a pre-order traversal; *Incremental handling* breaks the abstract syntax tree into statements that are then passed separately to the *Code generator*. The resulting pieces of object code are buffered, and sent to the target computer and the host database. Changes to declarations are handled by *Name binding* and *Type expansion*. *Incremental handling* identifies the statements that contain uses of affected declarations and passes these statements to the incremental code generator.

For the purposes of incremental recompilation, statements do not always correspond directly to Pascal statements. For example, the Pascal *if* statement is treated as three

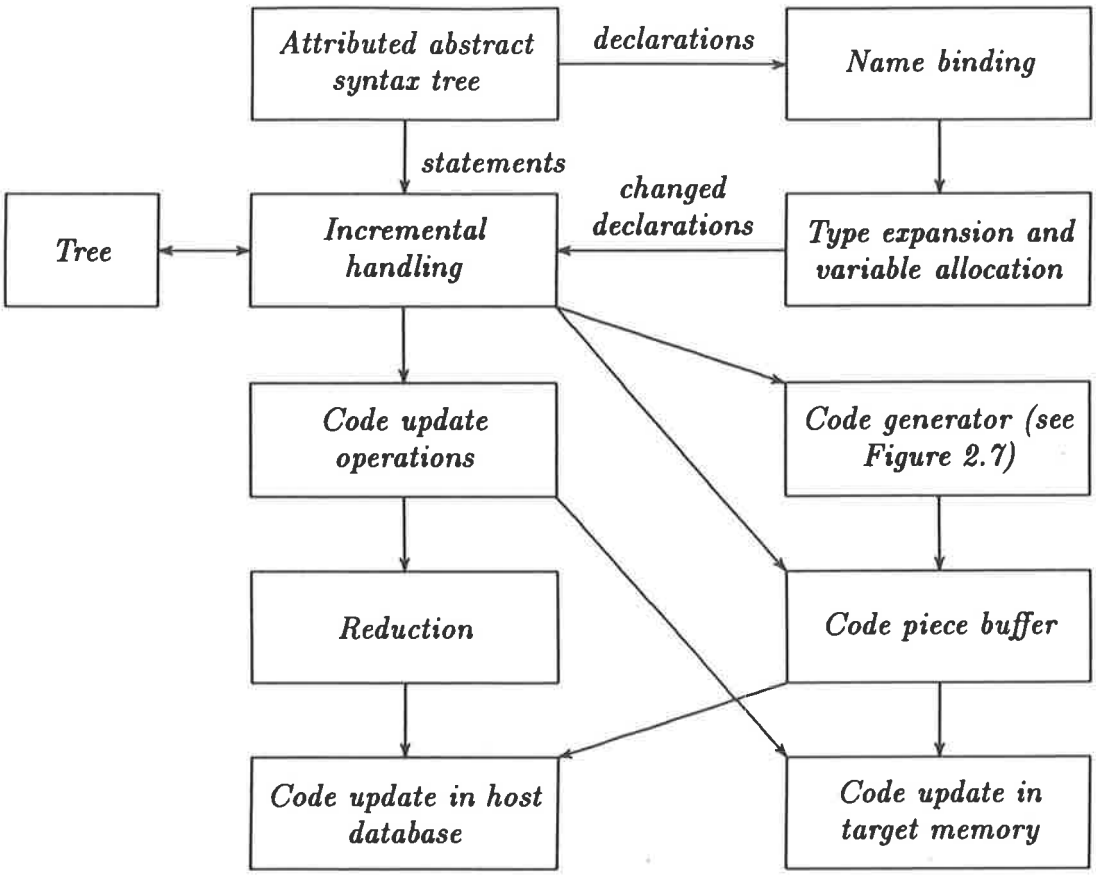


Figure 2.6 The DICE incremental compiler [Santi89].

separately compilable entities: the condition and each of the two branches. DICE handles this, and other similar constructs, with special hand-coded procedures.

The DICE code generator is based on the code generator from the Portable C Compiler (PCC) [Johnson78]. Input to the code generator is a set of abstract syntax tree fragments corresponding to statements. As shown in Figure 2.7, these fragments are converted into intermediate code trees. The intermediate code is a simplified copy of the abstract syntax tree, in which some nodes have been replaced by intermediate code trees. The intermediate nodes are decorated with information specific to the code generator, such as label keys for branch calculation or the required number of registers.

Tree-to-tree transformations may be applied to the intermediate code subtrees in the *Optimise* step, with the intent of improving the quality of the generated code. Constant folding is implemented by replacing subtrees that contain only constant values with single nodes. Simple expressions are recognised and replaced by more efficient equivalent

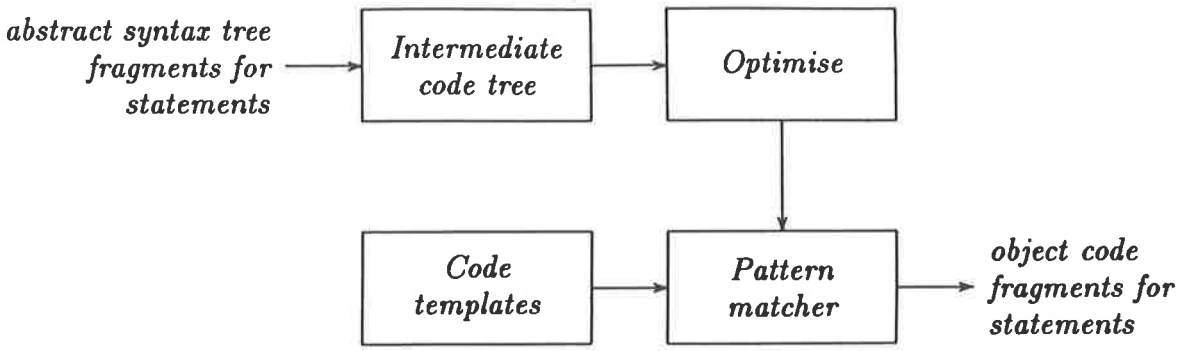


Figure 2.7 The DICE incremental code generator [Santi89].

expressions. Particular address expressions may be replaced to better fit the addressing modes of the target architecture.

Instruction selection is based on matching and rewriting the intermediate code tree. Pieces of the tree are matched against a table of templates and these templates associate an intermediate code pattern with a rewrite rule and a fragment of object code. When a particular template is selected, the object code is emitted and the matched expression's tree is destructively rewritten.

Before pattern matching, the register demands for each subtree of the expression tree are calculated. If the maximum number of required registers exceeds the available registers, the tree is subdivided and augmented with code to spill registers to temporary storage.

The source-to-object-code map is implemented by associating a *codesize* attribute with statement nodes in the abstract syntax tree. This attribute records the length of the object code emitted for the corresponding statement. For compound statement nodes, such as instances of the Pascal *while* or *if* statements, *codesize* is synthesised from the *codesize* attributes of the appropriate children.

No intermediate information, except for the *codesize* attribute, is maintained by the incremental compiler. Consequently, the storage overhead to support incrementalism is low, a stated design goal of DICE [Fritzson82], but no information can flow between the separately compiled statements. All optimisations and register allocations occur within

Granularity	*statement*
Canonical representation	*abstract syntax tree*
Semantic analysis	*ad hoc*
Intermediate information	***codesize and editmark***
Dynamic interaction	*demand-driven*

Figure 2.8 Summary of incremental compilation in DICE.

individual statements, necessarily restricting their nature and efficacy, and thus the quality of the emitted code.

Figure 2.8 relates the design of the DICE incremental compiler to the design issues enumerated in Section 2.3.1.

Architectural dependencies within DICE are localised in the code generator of the incremental compiler. The code generator originally generated PDP-11 code, but has since been retargeted to both the DEC-10 and MC68020. Retargeting to the MC68020 required both the creation of new code templates and the rewriting of many machine-dependent procedures within the code generator [Santi89]. In fact, considerable difficulties are reported in even discriminating between the architecture-dependent and architecture-independent parts of the code generator. Retargetability was a secondary design goal of the DICE system, as evinced by the use of PCC as the basis of the incremental code generator. The difficulties of the retargeting reported in [Santi89] reflect the problems inherent in porting PCC. For example, machine dependencies in the PCC intermediate representation have been noted by Henry in [Henry84].

The BEG-based incremental code generator used in the SPARC version of DICE [Fritzson95] is likely to prove simpler to retarget than the PCC-based version of DICE.

The DICE incremental compiler is hand-coded to support the Pascal language. In particular, incremental semantic analysis and the associated updating of the program database is hand-coded for Pascal, as are variable allocation and code generation. Creating a DICE environment to support some other language would require rewriting much of this code.

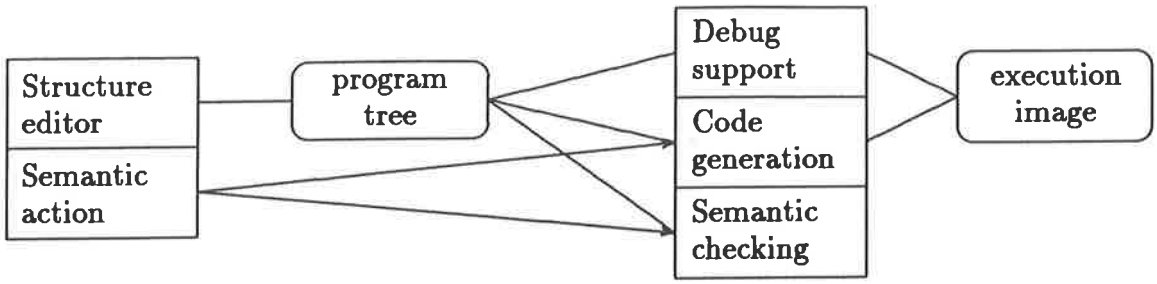


Figure 2.9 The LOIPE environment [Feiler82].

2.4.3 LOIPE

LOIPE [Feiler82, MedinaMora81a] is an integrated environment that includes a structure editor, a compiler and a debugger. All user interaction with LOIPE, whose structure is shown in Figure 2.9, is through the structure editor. A common program database is manipulated by the various tools.

The structure editor is an instance of the ALOE editor, arising from the GANDALF project [Notkin85, MedinaMora81b]. ALOE editors consist of a language-independent kernel supplemented by language-specific information, generated by the ALOEGEN compiler, and user-provided action routines. The action routines are automatically invoked by the ALOE kernel in response to editing events such as edits to the program or edit cursor movement.

The *program tree*, the canonical program representation of the LOIPE environment, is annotated with semantic and status information. The program tree is directly manipulated by the structure editor. When the program is modified, incremental semantic analysis, invoked by ALOE action routines, updates the semantic information. Further action routines, triggered by cursor movement, invoke code generation.

The primary focus of the LOIPE research was to demonstrate the feasibility of constructing a practical tightly integrated programming environment based on compilation technology. This discussion focusses on the incremental compilation aspects of the LOIPE system, rather than the overall environment issues that were the focus of both the LOIPE and GANDALF projects.

A type-checked and enhanced version of the C programming language, called GC, is supported by the LOIPE demonstration system. The LOIPE code generator is an adaptation of the code generator from the GC compiler, itself based on the Portable C Compiler [Johnson78]. Scanning and parsing is replaced by an interface to the program tree. When the user leaves a modified procedure, the program tree of the procedure is recompiled. The compiler performs both semantic analysis and code generation. Errors are reported in terms of program tree references.

Using an existing non-incremental compiler to perform code generation enabled the authors to focus on other aspects of integrated environments. However, this decision had a number of implications for recompilation grain size, retargeting of LOIPE and adapting it to support languages other than GC.

While Fritzson demonstrated that the Portable C Compiler's code generation algorithm could be adapted to perform fine-grained incremental compilation (see Section 2.4.2, p. 30), the LOIPE prototype directly includes all but the front-end of this compiler. The granularity of recompilation is therefore limited by the design of this compiler. The absence of procedure nesting in C implies that procedure-level granularity is essentially equivalent to the existing separate compilation facility. Thus, after the compiler has been suitably initialised with information about the global environment, procedure-level granularity is immediately realised.

Retargeting of LOIPE to a new architecture is essentially equivalent to retargeting the Portable C Compiler. In the PCC system, code templates must be recoded by hand, and architecture-specific support routines identified and recoded. Such an effort would be similar in complexity to Santi's retargeting of DICE (see Section 2.4.2, p. 34). Implicit machine dependencies in the PCC intermediate representation were noted in [Henry84]; these dependencies would complicate the retargeting of LOIPE.

Modification of the demonstration system to support a language other than GC would require either the adaptation and integration of some other base compiler, or the modification of the code generator for the new language. The first alternative requires the

Granularity	*procedure*
Code generator	*Portable C Compiler*
Register allocation	*Portable C Compiler*
Canonical representation	*annotated syntax tree*
Semantic analysis	*Portable C Compiler – triggered by action routines*
Dynamic interaction	*hybrid – triggered by action routines*

Figure 2.10 Summary of LOIPE.

availability of a suitable compiler, while the second requires that the new language be "close enough" to GC for the modification to be practical.

Figure 2.10 relates the design of the LOIPE incremental compiler to the design issues enumerated in Section 2.3.1.

2.4.4 Magpie

Magpie is an integrated programming environment for the Pascal programming language [Delisle84, Schwartz84]. The programmer interacts with the Magpie environment through *browsers*. *Code browsers* are used to construct and modify the program. *Workspaces* are used to specify execution and debugging actions. *Stack browsers* permit the examination of variables on the stack; *heap browsers* allow the examination of storage that is allocated on the Pascal heap. As much as possible, the Magpie environment appears as a single programming tool for the development of code, rather than as a collection of disparate tools. The range of Magpie browsers is described in greater detail in [Delisle84].

The Magpie environment is built on incremental tools. As the programmer modifies the source code using a code browser, an incremental parser maintains a decorated parse tree representation of the program. The parse tree is then incrementally compiled to native machine code and incrementally linked into the execution image.

Object code is generated during a single, syntax-directed walk of the decorated parse tree of a procedure. Thus, code generation occurs at procedure-level granularity. An incremental linker incorporates the new version of the generated code for each modified

Granularity	*procedure-level*
Canonical representation	*decorated parse tree*
Semantic analysis	*ad hoc*
Dynamic interaction	*greedy, but in the programmers idle time*
Granularity	*procedure-level*

Figure 2.11 Summary of the Magpie environment.

procedure into the program's object code. The combination of procedure-level granularity and an incremental linker eliminates the need for a complete mapping from the source code to the object code. Instead, the incremental linker must maintain the mapping from a procedure to its position in the program image.

Magpie attempts to provide a short preparation time by retranslating modified procedures as soon as all the errors in a procedure have been corrected. However, as also noted by Sawamiphakdi (see Section 2.4.5, p. 38), such greedy recompilation will adversely affect the response time of the environment. That is, the performance of the editor could well become unacceptably slow as the incremental code generator recompiles modified parts of the program. Procedure-level granularity, as in Magpie, exacerbates the situation. Sawamiphakdi's PSEP environment exploited parallelism to minimise the effect of code generation on the response time. In contrast, procedures that are ready for recompilation by the Magpie incremental code generator are placed on a queue. Magpie gives editing higher priority than translation, and so translation occurs only while no editing is performed, during the programmer's idle time.

[Schwartz84] reports that the prototype implementation is fast enough for efficient use. However, the memory requirements are considerable: on average, about 1,500 bytes are required for each Pascal source line. Several tradeoffs are suggested to reduce the storage requirements at the expense of processing time.

2.4.5 SEP

SEP is an experimental integrated programming environment for the Pascal-E language [Sawamiphakdi84, Ford85]. Pascal-E is a subset of Pascal [Jensen74] intended

(a) a fixed arity operator (b) a variable arity operator

Figure 2.12 Operator-operand trees in SEP [Sawamiphakdi84, Figures 2.24–25].

to provide a practical experimental programming language. SEP contains a structure-oriented editor for Pascal-E and a fine-grained incremental compiler that generates P-code from the program source.

The canonical program representation in SEP is a variant of abstract syntax trees called *operator-operand trees.* Operators in an operand-operand tree are either fixed arity or variable arity. Fixed arity operators, such as the *if* operator in Figure 2.12(a), require a fixed number of operands, where each operator is of a well-defined type. Variable arity operators, such as the *begin* operator in Figure 2.12(b), allow an arbitrary number of operands, but they must all be of the same type.

SEP uses an ad-hoc approach to semantic analysis. A hand-coded incremental semantic analyser maintains a symbol table for the Pascal-E program. Within the symbol table for a block, each visible identifier is associated with a list of the references occurring in that block. Thus, when an edit of a declaration occurs, SEP can identify the affected parts of the program that require recompilation.

[Sawamiphakdi84] is unclear about the details of the SEP instruction selection algorithm. However, as the focus of the research is on the parallel PSEP system and no discussion of instruction selection appears in this work, it seems that the translation to P-code is performed using simple template expansion during a top-down traversal of sections of the operator-operand tree.

Object code is kept in a code table. New tree fragments, resulting from program editing, are compiled into distinct segments of the code table. Code from the table is extracted and linearised prior to execution. For each node in the abstract syntax tree, a record is maintained in a *link buffer.* Each such record contains a *code address* and a *code*

Granularity	*expression*
Instruction selection	*presumably top-down template expansion*
Object code	*P-Code*
Register allocation	*none*
Canonical representation	*abstract syntax tree*
Semantic analysis	*ad hoc*
Dynamic interaction	*greedy*

Figure 2.13 Summary of SEP.

status field. If the code status is zero, then the code address field contains an absolute object code address. If the code status is one, then the code address field contains a link to a segment in the code table.

SEP was the precursor to the PSEP system. PSEP is a multiprocess variant of SEP in which the editor and code generator execute concurrently. The editor and code generator share the abstract syntax tree and symbol table. Edit notifications are written onto a worklist for processing by the concurrently executing code generator. The parallel incremental code generation algorithm of PSEP is discussed in more detail in Section 6.1.

2.4.6 Bivens' incremental register reallocation

The incremental generation of high-quality object code requires exploiting optimisations that cross recompilation grain boundaries. Good-quality register allocation, both locally (within a basic block of the source language) and globally (spanning basic blocks), contributes significantly to the quality of the object code. The work of Bivens, reported in [Bivens87, Bivens90], combines a small granularity of recompilation with the consideration of larger units for register allocation. Recompilation granularity is the intermediate code statement. Register allocation considers the entire enclosing procedure.

The intermediate representation used is a linear sequence of three-address intermediate instructions [Aho86] for each procedure.[3] The experimental design, described in

[3] [Bivens90] states that the intermediate representation is the control flow graph with

[Bivens87], includes only the code generator from a complete incremental compiler. Input to the prototype consists of three-address intermediate code statements and edits expressed in terms of these intermediate code statements. Thus, Bivens' research does not consider canonical representations, incremental semantic analysis, or the generation of the intermediate code.

In order to perform incremental register allocation that considers an entire procedure, Bivens incrementally maintains a control flow graph within which the vertices correspond to basic blocks containing the three-address intermediate instructions. Edits, expressed in terms of the intermediate instructions, may be partitioned into structural changes, affecting the topology of the control flow graph, and non-structural changes, whose effects are confined to single basic blocks. In addition to the control flow graph, the data flow information, encapsulating the extent and names used is required.[4] Both the control flow graph and the data flow information are part of the intermediate information maintained by Bivens' incremental code generator.

The set of available registers is partitioned into R_G, the registers that are available for global allocation, and R_L, the registers that are available for local allocation. Global allocation is performed first, followed by local allocation. Local allocation must also consider any global virtual register spans that could not be globally allocated.

Bivens describes two methods for global incremental register allocation. The first simply sorts the global spans according to decreasing priority and globally assigns registers to the first R_G spans. The second approach, based on graph colouring [Chaitin82,

---

vertices that are basic blocks consisting of three-address intermediate instructions. However, under the terminology established in Section 2.3.1, the control flow graph is considered intermediate information, while the three-address instructions form the intermediate representation.

[4] A *name* is the value of a variable in a region of the program. The name of a variable is changed by assignment. A name in a procedure has *global extent* if there is a use of it outside of the defining basic block; otherwise, it has *local extent.*

Granularity	*intermediate code statement*
Canonical representation	*none*
Semantic analysis	*none*
Intermediate representation	*three-address intermediate instructions*
Intermediate information	*control flow graph and register interference graph*
Register allocation	*incremental graph recolouring*

Figure 2.14 Summary of Bivens' incremental register reallocation.

Chow84], incrementally maintains and colours an interference graph in order to assign, and incrementally reassign, the R_G global registers to global spans.

Local register allocation proceeds by incrementally maintaining an interval graph. Vertices represent a contiguous span of intermediate instructions within a block corresponding to a virtual register span. Vertices are connected when the corresponding spans overlap.

Incremental algorithms for maintaining interference graphs permit incremental global register reallocation. Likewise, the incremental updating of the interval graphs enables incremental local register reallocation [Bivens87, Bivens90]. The experimental results reported in [Bivens87, Chapter 7] demonstrate considerable speedups in a comparison between the time to incrementally reallocate registers after a program change and the time to perform register allocation over the entire program.

2.4.7 Pollock's incremental optimisation

The generation of high-quality object code requires the application of optimisations such as common subexpression elimination or the moving of invariants out of loops. Such optimisations are effectively transformations that replace fragments of code with functionally equivalent fragments of code that are better with respect to some metric. Optimisations can be partitioned into local and global optimisations. A local optimisation occurs within a basic block. That is, no knowledge of the program from outside of a basic block is required in order to apply a local optimisation within the basic block. In contrast, global optimisations occur across the basic blocks of a program or require information to

flow across block boundaries. A methodology for the incremental implementation of both local and global optimisations is proposed in [Pollock85, Pollock86, Pollock92].

Not surprisingly, incrementally compiling code that has been transformed by optimisations adds to the complexity of incremental compilation. For instance, [Pollock86] identifies three ways in which the source-to-object-code mapping is complicated:

- optimisations such as *redundant store elimination* and *copy propagation* suspend the generation of code for some parts of the source code,
- optimisations such as *common subexpression elimination* rearrange the generated code, and
- an optimisation such as *constant folding* may replace a statement by another equivalent statement.

Furthermore, as noted in [Pollock86], when the source code is changed, the conditions for an existing optimisation may be invalidated. To ensure the correctness of the generated object code, such invalidated optimisations must be detected and reversed. Conversely, the same source code change may establish the necessary conditions for a new optimisation. Consistency of the generated code demands that the newly enabled opportunities be detected and the optimisations applied.

For example, local redundant store elimination is an optimisation in which a store to a variable, A, may sometimes be eliminated if there exists another store to A in the same basic block. If the necessary conditions for such a store elimination are satisfied, then the incremental optimising code generator eliminates the earlier store. However, if a use of A is subsequently inserted between the two stores, then the redundant store elimination is invalidated and the optimisation must be undone. Conversely, the deletion of a use of A between two stores to A may enable such a redundant store elimination.

Incremental program optimisation can thus be considered as a two part problem:

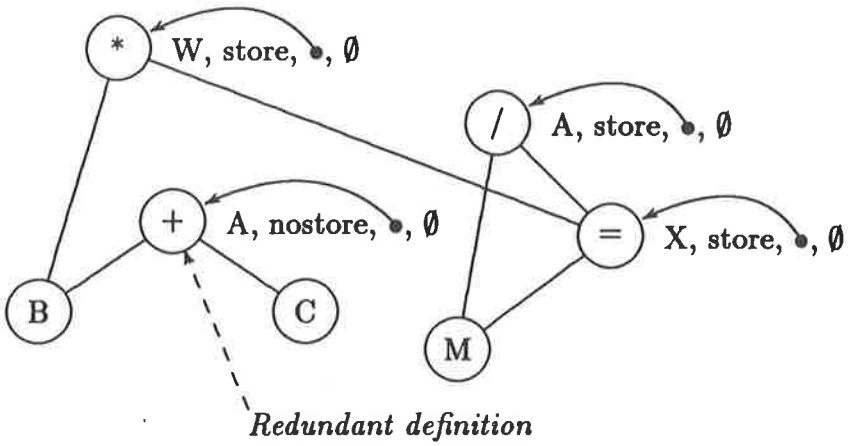
- the recognition of newly validated optimisations after a program edit, and
- the undoing of invalidated optimisations after a program edit.

X = M
A = B + C
W = B * X
A = M / X

(a) Unoptimised

X = M
W = B * X
A = M / X

(b) Optimised



(c) MFAD

Figure 2.15 MFAD representation of redundant store elimination [Pollock86, Figure 5.1].

Consequently, the intermediate information maintained during incremental optimisations must allow the efficient detection of valid optimisation and also maintain an optimisation history. Pollock uses a structure based on directed acyclic graphs (DAGS) and called the *Modified Flow Graph of Augmented DAGs* (MFAD). The MFAD representation of a program is a modified flow-graph in which each basic block is represented by a set of augmented DAGs [Pollock86, Section 4.3].

Algorithms that exploit the MFAD in order to incrementally implement a number of local and global optimisations are described in [Pollock86]. For example, consider the fragment of code shown in its unoptimised form in Figure 2.15(a). The MFAD representation of the code is shown in Figure 2.15(c). Nodes of the DAG are annotated with information that indicates the optimisation history of each statement. No uses are reachable by the indicated redundant definition. Thus, the elimination of the redundant store to A is a valid optimisation, as is apparent in the optimised variant of the code depicted in Figure 2.15(b). The use of this optimisation is recorded by setting the *storestatus* field of this node's annotation to *nostore*. In this instance, the MFAD representation both allows the detection of the potential optimisation and records its application.

Optimisations are not independent. The application, or the reversal, of a particular optimisation can enable the application of new optimisations or the invalidate existing

Granularity	*intermediate code statement*
Canonical representation	*unspecified*
Intermediate representation	*unspecified low-level code*
Intermediate information	*MFAD*

Figure 2.16 Summary of Pollock's incremental optimisation.

optimisations. Pollock's analysis establishes, for a well-defined collection of optimisations, the relationships between optimisations that require consideration during incremental optimisation of object code.

Pollock envisions the incorporation of the incremental optimiser into an integrated programming environment. Program modifications are initiated with a syntax-directed editor, and trigger incremental semantic analysis and the incremental generation of some low-level intermediate code for changed statements. Thus, the input to the incremental optimiser is a stream of intermediate code changes generated from the original source by the editor and semantic analyser [Pollock86, Chapter 2]. The optimiser can then either generate optimised intermediate code, requiring final code generation, or directly generate optimised machine code.

2.5 Conclusions

Incremental compilation addresses a fundamental aspect of the functionality desired for integrated software development environments: the provision of high-quality, efficient support for program execution. In particular, a fast turnaround after source code changes is desirable. Fine-grained incremental compilation is also an enabling technology for other sophisticated tools, such as Fritzson's debugging tools [Fritzson92, Fritzson94].

Feiler's LOIPE environment demonstrated the practicality of basing a programming environment on compilation technology. The DICE system shows that fine-grained incremental compilation is feasible as the underlying compilation paradigm.

Both LOIPE and Magpie exploited procedure-level incremental recompilation. The implementation of such coarse-grained incremental code generators is considerably simpler than the fine-grained incremental code generator embedded in the DICE environment. However, fine-grained incremental recompilation is an order of magnitude faster than coarse-grained incremental recompilation [Fritzson82]. The extent of recompilation required in a coarse-grained system after a change to an identifier declaration may include many complete procedures, whereas in a fine-grained system less redundant recompilation occurs.

Fine-grained incremental recompilation also enables the development of sophisticated debugging tools, such as the algorithmic debugging methods that have been added to DICE [Fritzson92, Fritzson94]. In such environments, the ability of a fine-grained incremental code generator to insert and modify very small fragments of object code is exploited to provide powerful capabilities in the debugging environment.

A goal of this work is to investigate very fine-grained incremental compilation. An incremental instruction selection algorithm capable of sub-expression level granularity is presented. An incremental code generator based on this algorithm is likely to prove a suitable basis for the investigation of applications of fine-grained incremental compilation to debugging and visualisation.

However, to have any chance of gaining widespread acceptance, the level of maturity of incremental compilation technology must be closer to that of traditional compilers. Both the quality of the emitted code and the performance of incremental compilers requires consideration, as does the question of their retargetability to other architectures. The work of Bivens and Pollock showed that the techniques used in traditional compilers to improve the quality of emitted code may be transported into an incremental setting. Sawamiphakdi suggested a parallel architecture that exploits coarse-grained parallelism to improve the performance of integrated environments based on incremental compilation.

Missing from the literature is a systematic approach to incremental instruction selection that allows the possibility of retargeting to new architectures. A retargetable incremental instruction selection algorithm is systematically derived from a non-incremental

algorithm in Chapter 5. This algorithm incrementally generates high-quality object code from an abstract syntax tree program representation. However, the reconciliation of this algorithm with a high-quality incremental register allocation technique or suitable incremental optimisation strategies is not investigated.

Two other matters are, however, addressed before the presentation of the derivation of the algorithm. The next chapter presents a distributed architecture for an integrated environment that allows the incorporation of an incremental code generator. Chapter 4 presents an algebraic model of abstract syntax and abstract syntax trees; this model serves as the theoretical basis of the derivations in Chapter 5 and Chapter 6.

The instruction selection algorithm derived in Chapter 5 is based on bottom-up tree rewriting. The adaptation of this algorithm that is derived in Chapter 6 is capable of operating in parallel with the other functions of an integrated programming environment (such as editing). The implementation of a prototype of this parallel incremental code generator is described in Chapter 7. Retargetability is an immediate consequence of the generation of the architecture-dependent components of the code generator from a precise specification of the target architecture.

Chapter 3

A distributed architecture for an integrated programming environment

Integrated programming environments assist a software developer engaged in the implementation and testing of software. They typically provide a mechanism for browsing and manipulating the program under development, and an execution facility to assist with program testing and debugging. Incremental compilation is one means by which a program execution facility can be provided in an integrated programming environment.

The MultiView distributed integrated programming environment is based on a client-server architecture. The server stores an abstract syntax tree representation of compilation units. Concurrently executing clients communicate with the server via message passing; they typically enable the browsing and manipulation of a single compilation unit. The prototype of the greedy parallel incremental code generation algorithm of Chapter 6 is implemented as a MultiView view. This implementation is described in Chapter 7.

This chapter commences with a discussion of two contrasting approaches to the construction of integrated programming environments, and a range of program execution facilities. Of particular interest is the manner in which an incremental code generator can be integrated into each architectural paradigm that is described. This integration is influenced by several aspects of the environment design:

- the data structure that represents the program under development,
- the manner in which the tools of the environment share data, and
- the mechanisms through which the tools of the environment communicate.

The remainder of the chapter presents the architecture of the MultiView environment; in particular, the message-based communication mechanism and the internal structure of a MultiView view are described. The client-server architecture of MultiView realises

several advantages of each of the two approaches to environment design discussed earlier in this chapter.

3.1 Integrated programming environments

Integrated programming environments are intended to support the coding phase of the software engineering life-cycle. Of particular interest in this thesis are those environments that provide at least a language-sensitive program editor for program creation and modification, along with support for program execution. Ideally, such an environment would be integrated into a larger environment, which would also contain tools to support other phases of the software life-cycle.

Several approaches to the provision of execution support have been exploited in existing integrated programming environments, including

- interpretation of pseudocode,
- an interface with an existing compiler, and
- incremental code generation.

Environments such as the Cornell Program Synthesizer [Teitelbaum81] and Pecan [Reiss84a, Reiss84b] support interpreted program execution. Each of these environments maintains some internal representation of the program under development; this representation is directly translated to pseudocode, which is then interpreted. The pseudocode may, in fact, be an integral part of the underlying program representation, as is the case in the Cornell Program Synthesizer [Teitelbaum81]. Alternatively, it may decorate the program representation, as is the case with Mughal's threaded interpreted code [Mughal88]. A further possibility is that the program representation may be annotated with references to the pseudocode that is stored in a separate structure, as in SEP [Sawamiphakdi84].

Interfacing the programming environment with an existing compiler is a rapid path to the provision of execution facilities. In order to facilitate the use of a traditional compiler, the environment must reconstruct the program text from its internal representation to serve as input to the compiler. The compiler listing is then analysed and error messages are related back to the internal program representation. For example, the second prototype

of the MultiView environment was interfaced to an Ada compiler [Altmann91]. In this case, the abstract syntax tree program representation is unparsed into a text file, the compiler invoked and then the resulting listing is analysed. In a compiler listing, each error message is associated with a text position in the source file. This position is mapped to a particular node of the abstract syntax tree, referred to as the *best fitting node.* The best fitting node for some location in the program source file is the root of the deepest subtree of the abstract syntax tree whose unparsing subsumes the location of the error in the compiler listing. Each error message is attached to its corresponding best fitting node and displayed appropriately to the MultiView user.

The approach to the provision of execution support that is explored in detail in this thesis is the integration of an incremental code generator into the programming environment. As described in Chapter 2, an incremental compiler maintains the object code of a program. After a change to the program, the object code is recompiled in time proportional to the "size" of the edit. The integration of the incremental compiler into the integrated environment is constrained by the architecture of the environment. For example, the DICE incremental compiler directly accesses the abstract syntax tree that is manipulated by the structure editor (see Section 2.4.2, p. 30). Thus, the editor is able to communicate with the code generator by storing information in the abstract syntax tree. Conversely, the incremental compiler proposed by Gafter assumes no such interaction with any other tool in the environment [Gafter90]; instead, it analyses the program source file and a log of edit operations.

The interaction between the compiler and the other tools of the environment can be characterised by two aspects of the environment architecture:

- the manner in which data is shared, and
- the mechanism through which the compiler is invoked.

The Unix programming environment serves to illustrate each of these aspects of the interaction between the compiler and the environment. Discrete Unix tools are explicitly invoked by the programmer and share data via the file system.

Unix editors manipulate an internal representation of the program source that is optimised for editing. Prior to compilation, this internal representation is translated into one-dimensional text and saved in a Unix file. The compiler parses the text in order to construct its internal program representation and generate the object code that is subsequently linked with system libraries into an executable file. In addition to object code, the compiler embeds a representation of its symbol table, and the mapping between the program source and locations in the object code, into the executable file. The symbolic debugger analyses this embedded data to synthesise its internal data structures.

The only data structures shared between tools are one-dimensional Unix files. In order to export information to other tools, a tool must translate its internal representation of the information into a one-dimension form and save it as a Unix file. To import information, other tools must analyse the one-dimension representation and construct a structured internal representation.

The invocation of the various tools is under the direct control of the programmer. When compilation is desired, the editor is explicitly told to save the file and the compiler explicitly invoked against the same file. When execution is desired, the debugger is directed to load the new version and run the program.

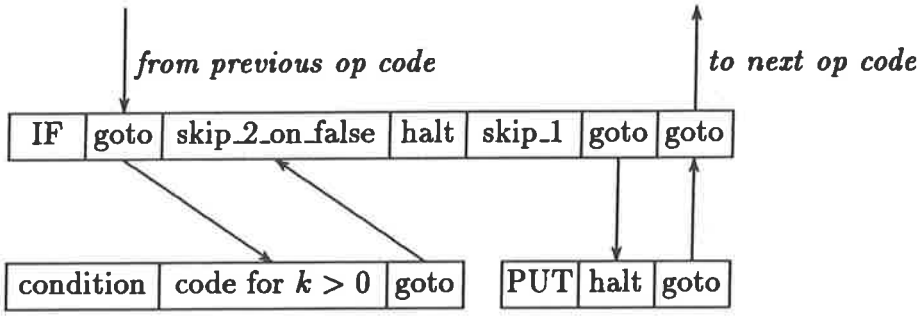
3.1.1 Monolithic integrated programming programming environments

When the editor and compiler share the program source as text, using the file system, the editor must translate its internal representation into one-dimensional text and the compiler parses this text in order to build its internal representations. However, if a single structured representation of the program source is shared by both the editor and the compiler, then this overhead is eliminated.

The distinguishing feature of *monolithic* integrated programming environments is that the program source is stored in a single data structure, referred to as the *canonical program representation*. For example, LOIPE is a monolithic integrated programming environment in which the canonical program representation is the program tree (see p. 35).

```
IF (k > 0)
  THEN statement
  ELSE PUT LIST (list-of-expressions)
```

(a) a source code fragment



(b) the corresponding derivation tree

Figure 3.1 Cornell Program Synthesizer derivation trees [Teitelbaum81].

Tighter coupling between the compiler and the environment permits a range of interaction paradigms:

- the LOIPE compiler is automatically invoked whenever the editing cursor moves out of a modified procedure,
- the SEP incremental compiler is automatically invoked immediately after a program update, and
- the DICE incremental compiler is invoked in response to an explicit request from the programmer.

The components that implement the diverse functionality of the environment, such as editing or interpretation, each access the shared canonical program representation. Consequently, the disparate information that is required in order to implement the functionality of each tool must be stored in this single representation. Execution facilities are no exception. Existing monolithic environments have used a range of paradigms for the provision of execution facilities. Several such examples will now be presented.

The Cornell Program Synthesizer is an integrated programming environment that supports coding, execution and debugging of programs written in PL/CS, an instructional

variant of PL/1 [Teitelbaum81]. Execution is facilitated by an interpreter that executes pseudocode.

Derivation trees are used as the internal representation of the program. Figure 3.1(b) illustrates the derivation tree for the code fragment shown in Figure 3.1(a). Nodes in the derivation tree include pseudocode for the interpreter. *Goto* instructions for the interpreter serve as the arcs between nodes. Unexpanded nodes[1] are indicated by a *halt* pseudo-instruction in place of the *goto* link. In particular, note that the tree structure is realised via the links between the code fields of the nodes. Thus, the interpreted pseudocode representation is an integral part of the program representation. The derivation trees, and the pseudocode, are constructed and modified as the program is edited.

A similar approach has also been used to provide execution facilities, based on the interpretation of threaded code, in a programming environment for Pascal generated by the Synthesizer Generator [Reps89a]. The Synthesizer Generator generates syntax-directed editors from a specification of the abstract syntax, the concrete syntax and unparsing schema. The resulting syntax-directed editor directly manipulates an abstract syntax tree. An incremental attribute evaluator is constructed from the specification of an attribute grammar over the abstract syntax. Attributes are stored in the nodes of the abstract syntax tree.

A scheme whereby fragments of pseudocode are created as attributes of nodes in the abstract syntax tree is described in [Mughal88]. Also, associated with each node is an *entry* and a *completion* attribute. The *entry* attribute of a node defines the location of the entry point of the code fragment associated with the node. When the execution of a fragment attached to a node is completed, the *completion* attribute of that node determines the next fragment to be executed.

---

[1] The Cornell Program Synthesizer editor enforces a top-down approach to program construction. Thus, incomplete nodes will occur in the derivation tree during program construction. Such incomplete nodes are represented by unexpanded placeholders.

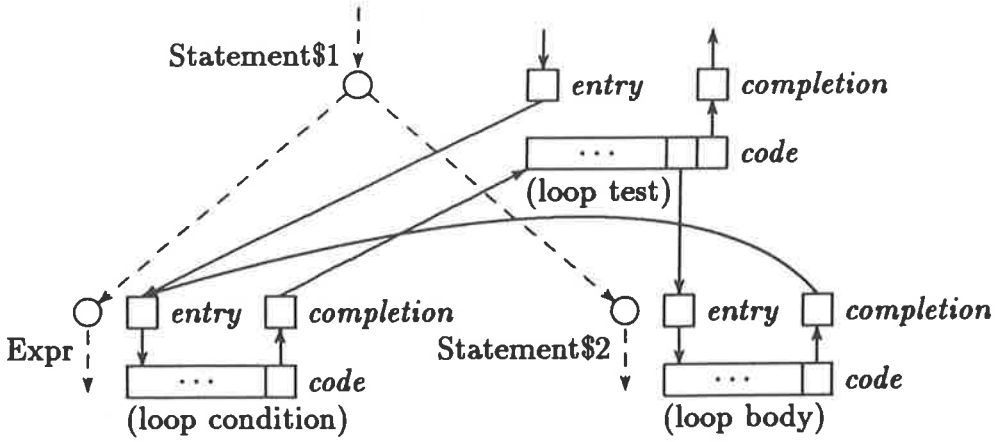


Figure 3.2 Attribution of a tree with threaded code [Mughal88, Figure 2].

The pseudocode attribution of a fragment of an abstract syntax tree corresponding to a while loop is illustrated in Figure 3.2. Abstract syntax tree nodes, denoted by circles and connected by dashed lines, are juxtaposed with representations of the *entry, completion* and *code* attributes. The *entry* attribute of a node points to either the start of the code fragment for that node, or the entry attribute of some other node. The *completion* attribute points to the *entry* attribute of some other node. The target of a branch instruction in a *code* attribute is either the *completion* attribute of the node, or the *entry* attribute of some other node. Thus, the discrete fragments of object code are linked, or threaded, by these attributes. In this case, the pseudocode executed by the interpreter decorates the canonical program representation but, in contrast to the Cornell Program Synthesizer, the pseudocode is not an integral part of the canonical representation.

The pseudocode attribution is specified, as part of an attribute grammar, in the Synthesizer Specification Language [Reps89a]. The resulting editor includes an incremental attribute evaluator. Attribute values are immediate evaluated after program edits. Thus, the pseudocode is incrementally regenerated immediately following any program update.

The SEP system (see Section 2.4.5, p. 38) is a monolithic environment that uses operator-operand trees as the canonical program representation, as depicted in Figure 2.12 (see p. 39). The SEP incremental code generator emits P-code that is stored in a data

structure, called the code file, separate from the shared operator-operand tree. The source-to-object-code mapping is realised by decorating nodes of the tree.

The SEP incremental compiler is invoked immediately after the operator-operand tree is updated by the editor. That is, the incremental compiler interacts with the editor in a greedy manner. In PSEP, which was developed from SEP, the single program representation is shared by concurrently executing code generator and editor [Sawamiphakdi84]. A mechanism is proposed in which the operator-operand tree is maintained in a file that is shared by the two processes.

In the DICE system (see Section 2.4.2, p. 30), the program editor and incremental compiler access the same shared data structure. The editor communicates with the code generator via the *editmark* attribute stored at statement nodes in the shared abstract syntax tree. The code generator stores information to implement the source-to-object-code mapping in the *codesize* attribute of statement nodes. The actual object code is not stored in the canonical program representation.

The DICE incremental code generator is demand driven (see Chapter 2, p. 9). That is, the compiler is not invoked until the programmer either requests program execution or explicitly commands recompilation.

The above examples of previous systems illustrate a spectrum of techniques that have been used to provide execution facilities in monolithic integrated programming environments:

- interpretation of pseudocode that is an integral part of the canonical program representation,
- interpretation of incrementally generated, threaded code that decorates the canonical program representation,
- interpretation of incrementally generated P-code that is stored in a separate data structure and linked into the canonical program representation, and
- direct execution of incrementally maintained object code.

The sharing of a single data structure between all the tools, the hallmark of monolithic integrated programming environments, has performance benefits for the complete system.

Furthermore, this sharing simplifies several aspects of the design and implementation of an integrated programming environment. For example, the DICE code generator exploits its ability to directly access the abstract syntax tree by

- communicating with the editor through the *editmark* attribute of statement nodes, which may be set by the editor, and
- storing information that must persist between successive recompilations in the abstract syntax tree, specifically in the form of the *codesize* attribute of statement nodes.

However, the use of a shared program representation also imposes limitations on monolithic integrated programming environments. For example,

- the addition of new functionality may require that the form of the shared data structure be modified in order to store additional information,
- the shared data structure is loaded with information specific to single tools, rather than containing just program information that is useful to all tools,
- integration with external tools is difficult, and
- the exploitation of any available parallelism is difficult because of the complexities inherent in the simultaneous manipulation of a single data structure by several concurrent processes.

A range of paradigms for invoking the incremental compiler is also apparent. For example, the demand-driven incremental compiler in DICE is invoked after a specific request from the programmer. Conversely, the greedy incremental compiler in SEP is invoked by the environment immediately after a program update. In each case, the compiler is tightly integrated into the environment, which is able to directly invoke the compiler.

3.1.2 Broadcast message servers

In contrast to the monolithic approach, an environment design that is focussed on a loosely coupled integration mechanism can more easily embrace disparate tools. The Field environment [Reiss90a, Reiss90b] integrates Unix tools via a broadcast message

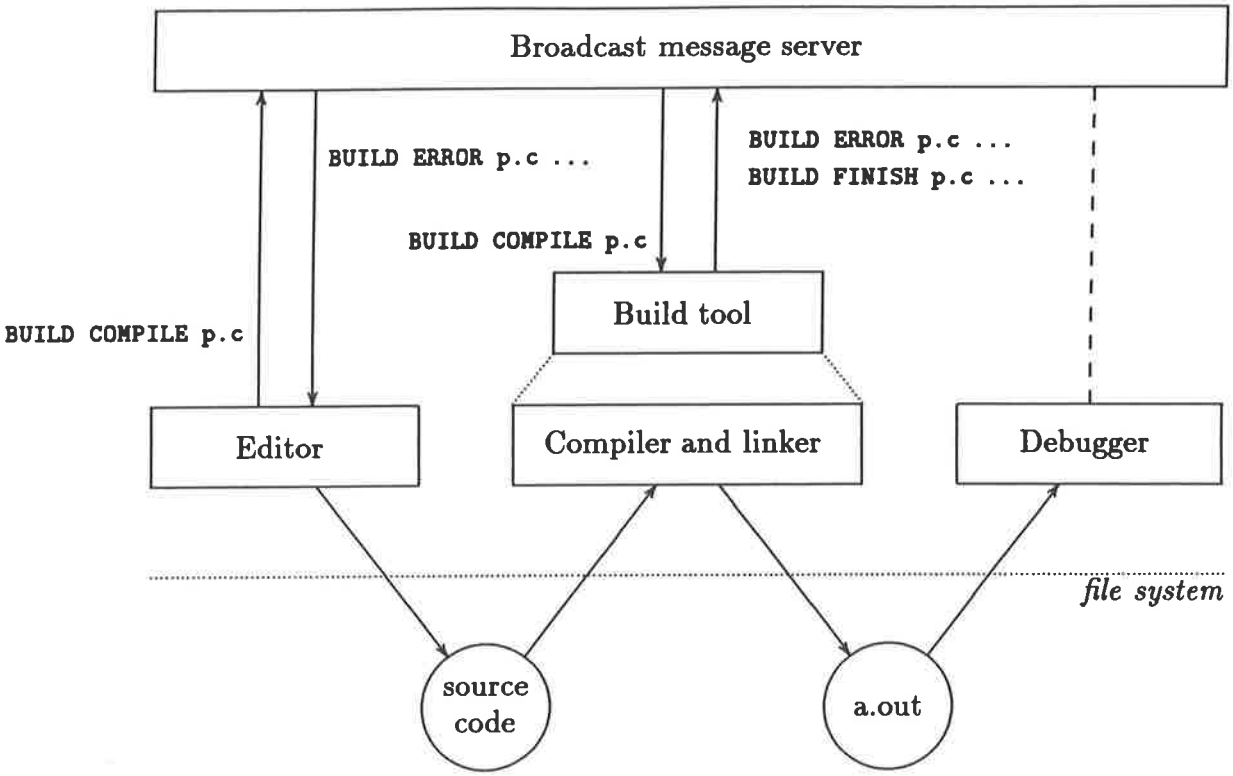


Figure 3.3 Integration of a compiler into the Field environment.

server. Textual messages delivered to the message server are selectively rebroadcast to a collection of other tools.

In contrast to the monolithic approach, in which each tool must manipulate the shared program representation, the message-based integration mechanism allows existing Unix tools to be integrated into the Field environment. The integrated tools still share data through the file system, but they are adapted so that actions are triggered by the delivery of messages from the message server.

For example, the Unix **make** utility [Feldman79] is integrated by via the *build tool* [Reiss90b]. The build tool provides a graphical interface to **make**. Makefiles can be interactively created, updated and saved to the file system. The **make** utility is invoked, from the graphical user interface, to compile and link the program.

The build tool is also connected to the broadcast message server. During initialisation, it registers a set of text patterns with the server. Whenever a message that matches one of these patterns is delivered to the message server, it is rebroadcast to the build tool. In

particular, a pattern that matches the string "BUILD COMPILE *filename*" is registered. Suppose that the editor sends the message "BUILD COMPILE p.c" to the message server, as illustrated in Figure 3.3. The message server selectively broadcasts this message, sending it to the build tool. When the build tool receives the "BUILD COMPILE p.c" message, it invokes the **make** utility to compile p.c. Error messages from the compiler are analysed and, for each such error message, the build tool sends a "BUILD ERROR p.c ..." message to the server. Figure 3.3 shows the broadcast of an error message to the editor. The editor can then indicate the location of the errors to the programmer. After synthesising a "BUILD ERROR p.c ..." message for each error, the build tool indicates that all errors have been found by sending a "BUILD FINISH p.c" message.

The interaction between the build tool and the editor illustrates the use of the message service both for issuing commands, exemplified by the sending of the "BUILD COMPILE p.c" message by the editor, and for signalling events, exemplified by the sending of the "BUILD ERROR p.c ..." and BUILD FINISH p.c ..." messages by the build tool. Any tool connected to the message server is able to signal events by sending an appropriate message. For example, during program execution, the Field *ddt* debugger [Reiss90b] exploits the message server to signal the status of the execution.

In general, tools in the Field environment interact via the message server and share data via the underlying file system. Suppose that an incremental compiler is to be integrated into the Field environment. Recompilation is triggered by the reception of messages. The program source is accessed, as text, via the Unix file system. During recompilation, the incremental compiler compares the new version of the program against the old version in order to incrementally update its internal representation of the program source and its semantics to reflect any changes. Then, any obsolete object code is regenerated.

The onus is placed on the incremental compiler to maintain some structured representation of the program and its semantics, in order to satisfy the incrementalism condition. Furthermore, the information gleaned by the compiler is not directly available to other tools that could usefully exploit it. For example, the debugger is unable to access the source to object code mapping kept by the incremental compiler, unless the compiler

specifically writes out a representation of this information in the executable file. Furthermore, any such representation that is generated by the code generator must relate the object code to positions in the source text, rather than to the internal representation kept by the code generator.

The message server can be used to implement a range of paradigms for the interaction between editing and incremental compilation. Demand-driven recompilation occurs if the incremental compiler recompiles updated source code in response to build messages. Greedy recompilation occurs if the editor is adapted to regularly save the updated source code and to send a notification whenever the source code is saved. In general, the form of the interaction is constrained by the frequency with which the program source is saved, and the existence of messages signalling that the shared program source file has been updated.

Gafter's design for an incremental compiler is well suited to this style of integration [Gafter90]. In Gafter's approach, the incremental compiler reads a log of text editing operations, along with fragments of the source program, in order to recompile the object code. However, the oft-repeated analysis of program source code would adversely affect the speed of such an incremental compiler. While not considered further in this thesis, this approach to incremental recompilation is worthy of further investigation because of its applicability to a wide range of environments.

3.2 A client-server architecture

The monolithic paradigm, in which a single data structure is shared by all the tools of the environment, simplifies the sharing of information between the tools. However, the integration of existing tools is particularly difficult and the construction of a new tool will often require changes to the form of the canonical representation. Conversely, the broadcast message server paradigm simplifies the integration of new and existing tools, but the sharing of information is more difficult.

An architecture in which a high-level canonical representation is understood by all tools, and in which the concurrently executing tools communicate via message passing,

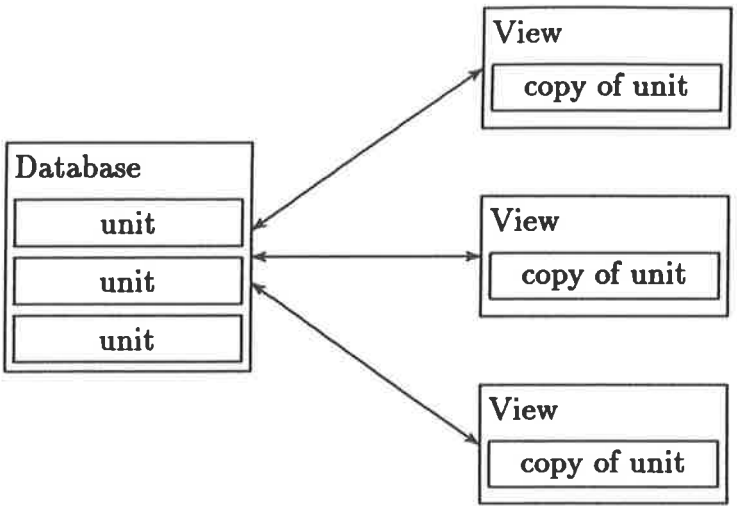


Figure 3.4 The MultiView architecture.

potentially combines the ease with which program-related information is shared in monolithic environments, with some of the flexibility of the message server approach. Furthermore, the provision of an appropriate infrastructure can simplify both the creation of new tools and the integration of existing tools via a simple adaptor.

The MultiView distributed integrated programming environment is based on a client-server architecture [McCarthy85, Marlin86, Altmann88, Marlin90, Altmann91, Marlin93, McCarthy94]. Multiple client processes connect to a single server process, as illustrated in Figure 3.4. The server, called the *MultiView database*, maintains a central representation of any program source that is being manipulated by the environment. Source code modules, referred to as *units*, are stored as decorated abstract syntax trees. Clients, called *MultiView views*, are connected to the database, and enable the browsing and editing of source code.

The database maintains the canonical representation of each unit that is currently loaded into the environment. A typical view facilitates the browsing or manipulation of a single unit. A copy of the unit is cached by the view and displayed, either graphically or as text. When a unit is modified, via an editing view, the cached copy is updated and the database notified of the edit. In turn, the database notifies other views that have cached copies of the same unit. Observe that the database, in addition to its other

functionality, does in fact provide functionality similar to that of a broadcast message server, with respect to the views.

The design of the MultiView environment is guided by several basic principles:

- the use of a tree-based canonical representation of the program source code,
- language independence,
- the exploitation of parallelism, and
- the provision of multiple, simultaneously updated, views that allow both browsing and editing of the source code.

Source code is stored and manipulated only as decorated abstract syntax trees. That is, units are realised in the database as abstract syntax trees, as are the copies that are cached by each view. The abstract syntax tree nodes may be decorated with arbitrary data. For instance, the database of the current prototype implementation is able to decorate the nodes with attributes resulting from incremental static semantic analysis of the program, and the textual editing view decorates each node of its cached copy of an abstract syntax tree with data that reflects the screen coordinates of the unparsed representations of the node.

The implementation of abstract syntax trees in the MultiView environment is based on a precise model of abstract syntax. This algebraic model facilitates the design of an abstract syntax specification language and also the implementation of the associated language compiler. Furthermore, the experience of implementing the abstract syntax tree and associated operations has reinforced the benefits of basing design and implementation decisions on a clear and unambiguous semantic model.

The provision of a formalism for the specification of the supported programming language makes the MultiView system essentially language independent. Constructing a MultiView environment for a new language requires the definition of the language in this formalism. The specification language is described further in Section 4.6, after the presentation of the algebraic model of abstract syntax in the earlier sections of Chapter 4.

The distributed architecture allows the exploitation of the coarse-grained parallelism available on modern multiprocessor workstations and networks of workstations. In the

prototype, the database and views are implemented as concurrently executing processes that communicate using the Unix interprocess communication primitives. In turn, these discrete processes are implemented as multitasking Ada programs, further increasing the potential exploitation of any parallelism provided by the underlying hardware. Communication between the database and views is facilitated by a message passing system, called the communication subsystem (CSS) [McCarthy94], that is implemented on top of the underlying interprocess communication mechanisms provided by the operating system.

Different kinds of view provide different visual representations of the program source. The views currently available will now be characterised briefly; earlier versions of some of these views were also constructed [Altmann86, Lee87, Brook90, Beck92]. TextView unparses the program representation to text and enables editing of the program using either structural editing operations or the more traditional text based editing operations [Read93]. FlowView unparses the canonical representation into flow charts [Jacobs91, Jacobs95]. A tree view has also been implemented that graphically displays the tree structure of the canonical representation. Simultaneously executing views of a particular unit are simultaneously updated in response to a single program edit. For example, if the user inserts a new statement using a textual view, a simultaneously executing flowchart view of the same unit is immediately updated to reflect the edit.

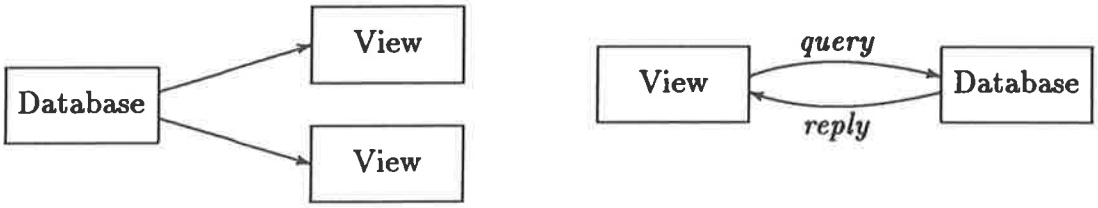
Of particular interest in this thesis are the model of the canonical program representation, namely abstract syntax trees, and the interaction between the database and the concurrently executing views. As already mentioned, the model of abstract syntax trees is described in Chapter 4. The interaction between the database and views is described in the following section.

3.2.1 Communication between the database and views

Interaction between the MultiView database and views is via the exchange of messages. A message consists of a tag field, indicating the type of the message, and a collection of data fields, fully determined by the tag. For instance, the query message from a view

```
message LOAD_UNIT is
  query
    FILE_NAME: STRING_TYPE;
  reply
    STATUS: STATUS_TYPE;
    UNIT: AST_IDENT_TYPE;
end LOAD_UNIT;
```

Figure 3.5 The high-level specification of the `LOAD_UNIT` query.



(a) asynchronous broadcast (b) synchronous query

Figure 3.6 Communication between the MultiView database and views.

that is tagged with the name `LOAD_UNIT` contains the file name corresponding to the program source that is to be loaded by the database.

Message types and their associated data fields are specified in a high-level protocol specification language [McCarthy94]. For example, an extract of the high-level protocol specification for MultiView is shown in Figure 3.5. This specification defines the `LOAD_UNIT` query. The query contains the string-valued field `FILE_NAME`. The reply, from the database, contains the two fields `STATUS` and `UNIT`. The complete set of messages that together define the MultiView protocol are specified in this protocol specification language. A protocol compiler generates the protocol-specific components of the MultiView implementation from the protocol specification (see p. 182).

Two forms of interaction between the database and views are distinguished. *Asynchronous broadcast* by the database, as illustrated in Figure 3.6(a), occurs when the database notifies a collection of views of some event, such as an edit to an abstract syntax tree or the creation of a new unit; the same message is broadcast to an arbitrary collection of views. From the perspective of the view, broadcasts from the database may arrive at

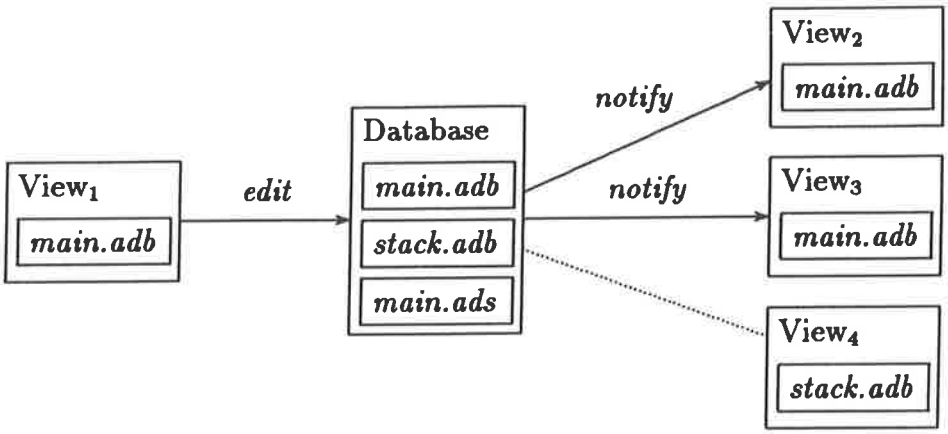


Figure 3.7 Broadcast of an edit notification.

any time, and so must be handled asynchronously. *Synchronous queries*, as illustrated in Figure 3.6(b), are initiated when a view sends a query to the database. The view then awaits the arrival of a reply from the database. Any information requested by the query is encapsulated in the reply message. This interaction is synchronous, from the perspective of the view, in that the view initiates the interaction with the database and then waits for the database to respond.

The fields of the synchronous query and reply messages are specified as part of the high-level protocol specification. For example, as already explained, the `LOAD_UNIT` query defined in Figure 3.5 contains a single string field, namely `FILE_NAME`, and the corresponding reply contains two fields, namely `STATUS` and `UNIT`. A similar mechanism serves for the specification of the fields in asynchronous broadcast messages.

Notifications of edit operations are delivered to views by the asynchronous broadcast mechanism. For example, consider the snapshot of a MultiView session shown in Figure 3.7. View_1 sends an edit message for the unit *main.adb*, as a synchronous query, to the database. Two other views, View_2 and View_3, have also cached *main.adb* and, presumably, registered an interest in this unit with the database. Thus, when an edit operation occurs on *main.adb*, notifications are broadcast to these views, as in Figure 3.7. View_4, having cached the unit *stack.adb*, has not registered an interest in *main.adb*; consequently, the notification is not broadcast to this view.

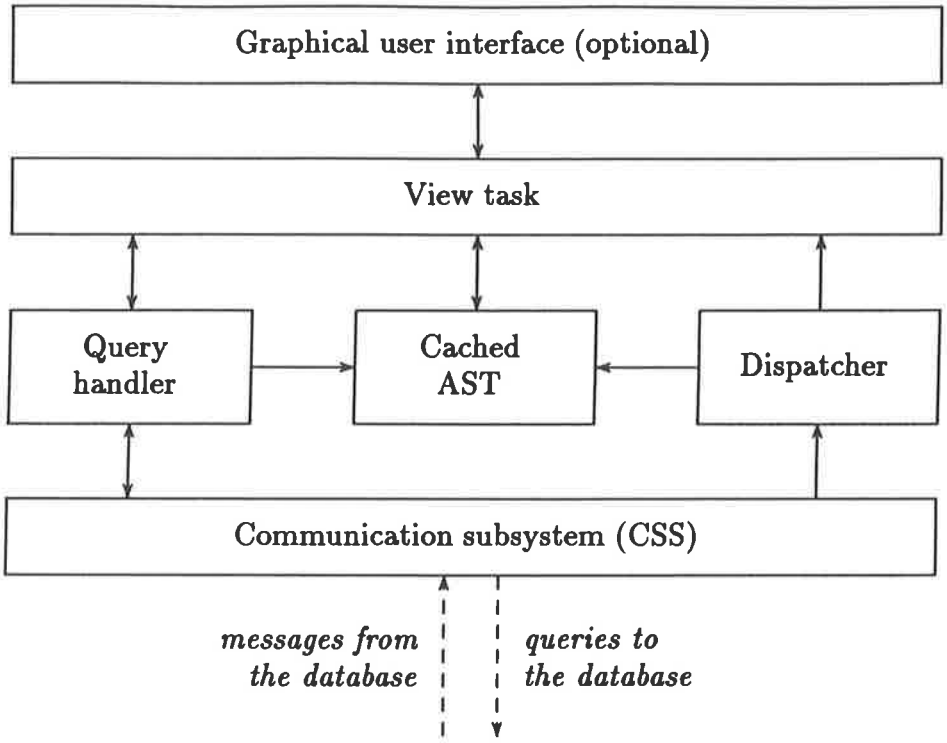


Figure 3.8 Structure of a MultiView view.

3.2.2 The structure of a view

The prototype incremental code generator, described in Chapter 7, is implemented as a MultiView view. This section gives a general overview of how a view is constructed.

In general, views are event driven. Input events are either asynchronous broadcast messages from the database or input from the user. For an incremental code generator view, the relevant input events come from the database and are notifications of subtree replacements.

The structure of a typical MultiView view is illustrated in Figure 3.8. A broadcast message arriving from the database is delivered, by the underlying interprocess communication mechanism, to the CSS. The CSS decodes the message and passes it to the *dispatcher*. Message processing by the dispatcher is view-dependent; in general, however, the dispatcher will either update the cached abstract syntax tree, in response to an edit notification, or invoke some view-specific action in the view task, or both. The *view task*, executing concurrently with the CSS, implements the view-specific operations and functionality. In the prototype code generator view, the view task is an incremental code

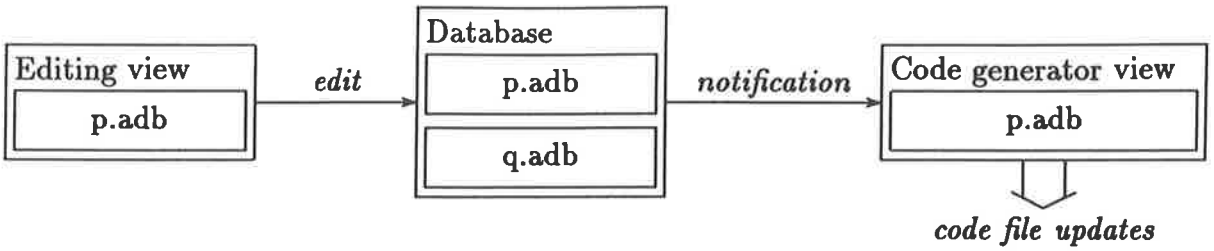


Figure 3.9 A MultiView code generator view.

generator. In the tree view, the view task implements a mapping from the cached abstract syntax tree to a tree-like visual representation. The graphical user interface is, of course, determined by the nature of the view. Indeed, it is by no means mandatory that a view has a graphical user interface at all.

The *query handler* provides an abstraction, corresponding to remote procedure call (RPC), over the synchronous query form of interaction with the database. For each query, there is a procedure call with an input parameter corresponding to each field of the query and an output parameter corresponding to each field of the reply. The query handler procedure constructs the necessary query, passes it to the CSS for transmission to the database, and then waits for the reply. The fields of the reply are extracted and returned in the output parameters of the query procedure.

The protocol compiler assists the view writer in the implementation of the dispatcher and the query handler (see Section 7.1.4). Furthermore, a mechanism is provided that allows the distillation of behaviour that is common to several views into a reusable component, such as that for caching an abstract syntax tree in the view.

The way in which an incremental code generator can be implemented as a MultiView view is shown in Figure 3.9. The code generator view caches a unit from the database. When an edit notification is delivered to the code generator, the incremental compiler is invoked to update the object code file. Such a code generator view is the basis for the approach to parallel incremental code generation discussed in Chapter 6.

3.2.3 A brief history of MultiView

To date, there have been four separate prototype implementations of the MultiView environment. Lessons have been learned from each successive prototype. In chronological order, the prototypes are

- the initial prototype implemented in Lisp,
- a reimplementation, in Modula2,
- a reimplementation, in Ada, and
- the current prototype, also implemented in Ada.

The initial implementation, described in [McCarthy85], was a preliminary exploration of the basic client-server approach to building an integrated programming environment. I programmed this prototype in Lisp to execute on a network of Sun workstations. The only view provided was a simple, structure-oriented, syntax-directed editor. This prototype demonstrated that the basic architecture, based on selective rebroadcast and querying, is a viable approach to the construction of integrated programming environments.

Following the initial prototype was a reimplementation in Modula2, by Altmann and Hawke. This version, described in [Altmann88, Marlin90, Altmann91], provided both a textual and a tree view for the Modula2 and Ada programming languages. Problems with the communications layer of this prototype indicated that a more sophisticated message passing protocol was required, and that the caching of abstract syntax trees in views is necessary to achieve acceptable performance.

The third MultiView prototype was implemented in Ada, to support editing of Ada source code. This prototype is reported in [Marlin90, Jacobs95]. An experiment in which this prototype was integrating with the Merlin process-centred software development environment is described in [Marlin93]. Experience with this prototype highlighted the need for:

- a well-defined and simple canonical representation,
- a systematic approach to the evolution of the communication protocol,
- a methodology for the development of views,

- the provision of specific tools to aid the evolution of the protocol, the implementation of views and the instantiation of MultiView for new languages, and
- the provision of facilities to aid the integration of MultiView with other systems.

The abstract syntax trees used in the first three prototypes allowed both fixed and variable arity operators. This program representation is analagous to the operator-operand trees used in the Mentor syntax-directed editor [DonzeauGouge80, Kahn83]. This dichotomy of operator types, and the absence of an unambiguous model of abstract syntax, complicates the creation and manipulation of abstract syntax trees. Furthermore, no preprocessor existed to compile formal specifications of an abstract syntax into the tables used in the MultiView implementation. Thus, the abstract syntax was encoded in tables that were essentially maintained by hand.

The lexicon of message types was also hand coded into the implementation of the communication subsystem. The addition of new messages types and the modification of fields of existing messages required the editing of quite intricate code. This led to numerous errors in the implementation of the message passing subsystem, and a consequent reluctance to experiment with the protocol. The intricate and repetitive nature of much of the message handling code makes it a prime candidate for automatic generation from a higher level specification.

The implementation of views in the third, and earlier, prototypes followed a largely ad hoc methodology. At best, a crude skeleton based on the text editing view was used as the basis for new views. However, the use of this skeleton precluded any experimentation with the lexicon of message types without also altering the implementation of each view based on the skeleton. Furthermore, the use of the view skeleton placed onerous constraints on someone implementing a view.

I addressed the difficulties manifested in the earlier prototypes in the fourth, and most recent, prototype. Firstly, I developed the precise model of abstract syntax that is described in Chapter 4. From this model, a formalism for the specification of abstract syntax was constructed and the corresponding compiler implemented. Next, the message structure and communication subsystem were completely redesigned, along with the

implementation of the protocol compiler. Finally, a new MultiView database was constructed, based on the new notion of abstract syntax and the restructured communication subsystem. The first view written for this prototype was the TextView implementation by Read, described in [Read93]. Subsequently, a tree view has been written and the flowchart view described in [Jacobs95] has been ported to the new prototype. Furthermore, based on the experience of integrating MultiView with Merlin [Marlin93], explicit support for the construction of adaptors is included in the CSS; such adaptors are the basis for the integration of MultiView into other environments.

The protocol compiler underpins the evolution of the communication protocol. New messages are added to the lexicon by editing the high-level protocol specification and regenerating the message-specific components of MultiView with the protocol compiler. Furthermore, view implementations are considerably less sensitive to variations in the protocol than in the previous prototypes; this flexibility is a direct consequence of the explicit support for view construction inherent in the protocol-specification compiler described in [McCarthy94].

3.3 Conclusions

The design of MultiView integrated programming environment is guided by four principles:

- the use of a tree-based canonical program representation,
- language independence,
- the exploitation of parallelism, and
- the provision of multiple, simultaneously updated views of the source code.

Source code is stored in MultiView as decorated abstract syntax trees. A typical view caches a copy of a single abstract syntax tree. This canonical representation is annotated with semantic and view-specific data. The implementation of abstract syntax trees that is described in Chapter 7 is based on the algebraic model of abstract syntax tree presented in the next chapter.

The language-specific components of the environment are generated from a formal specification of the supported programming language. The MultiView language specification language (LSL) is described in the next chapter, after the presentation of the algebraic model of abstract syntax on which LSL is founded.

MultiView is based on a distributed client-server architecture. The database and views communicate by exchanging messages. The lexicon of messages is defined in a high-level specification that is compiled by the MultiView protocol compiler. The database and views are implemented as separate Unix processes in the MultiView prototype described in Chapter 7.

Each type of view in MultiView enables the browsing or manipulation of a single compilation unit. Both textual and graphical views of the program are provided. The creation of new view types is facilitated by explicit assistance for the view implementor provided by the protocol compiler.

An incremental code generator can be realised as a MultiView view. Such a view receives notifications of edits from the database, updates a cached copy of the abstract syntax tree and recompiles the affected parts. Chapter 7 describes the implementation of a prototype MultiView incremental code generator view that is based on the greedy parallel incremental code generation algorithm of Chapter 6.

Chapter 4

Abstract syntax trees

The incremental instruction selection algorithm, presented in Chapter 5, is based on the incremental rewriting of abstract syntax trees. The derivation of this algorithm is founded on a precise model of abstract syntax trees, editing of abstract syntax trees, and the static semantics of programs represented as abstract syntax trees.

This chapter presents the algebraic model of abstract syntax used in this thesis; in this model, abstract syntax trees are considered to be words which are freely generated by a heterogeneous algebra [Higgins62, Birkhoff70]. The algebraic notation used is taken from [Higgins62]. The algebraic characterisation of abstract syntax trees is reconciled with the more familiar tree notation by relating abstract syntax trees, considered as words over an algebra, to tree domains [Kron75]. The model is then equipped with a notion of program semantics that is based on attribute grammars, and a notion of editing based on the replacement of subtrees in abstract syntax trees.

Finally, the MultiView language specification language (LSL) is introduced. The specification of a programming language in LSL consists of

- the abstract syntax of the programming language,
- the concrete syntax and a mapping to the abstract syntax, and
- an attribute grammar.

Appendix A contains a glossary of the symbols used in the algebraic model of abstract syntax presented in this chapter, and in the analyses in subsequent chapters.

The approach to the modelling and specification of abstract syntax described in this chapter is similar in spirit to the ASF+SDF formalism described in [Bergstra89] and [Klint93].

4.1 Abstract syntax

Let the *set of sorts,* $\mathcal{S}$, be any set of symbols; in this thesis, $\mathcal{S}$ is always a finite set of symbols. For the abstract syntax example that follows, let $\mathcal{S} = \{Int, Real, Expr\}$.

An n-term relation over $\mathcal{S}$ is a subset of $\mathcal{S}^n$. An $(n+1)$-ary relation $\mu \subseteq \mathcal{S}^{n+1}$ is an operation with arity n over $\mathcal{S}$ if $(s_1, \ldots, s_n, s) \in \mu$, and $(s_1, \ldots, s_n, t) \in \mu$ implies $s = t$. These operations with arity n, or n-ary operators, over $\mathcal{S}$ are used to specify the signatures of operators in the abstract syntax; 0-ary, or nullary, operations are considered to select a single member of $\mathcal{S}$. Let $\mathcal{R}(\mathcal{S})$ denote the set of relations over $\mathcal{S}$ and $\mathcal{R}^n(\mathcal{S})$ denote the set of n-ary relations over $\mathcal{S}$. For conciseness, $\mathcal{R}^n(\mathcal{S})$ will be written as $\mathcal{R}^n$ if $\mathcal{S}$ can be uniquely inferred from the context.

Let the *set of operators,* Ω, be the union of the disjoint sets of symbols $\Omega_n, n = 0, 1, 2, \ldots$ such that each Ω_n is disjoint from $\mathcal{S}$. An Ω-structure, α, on $\mathcal{S}$ is a mapping $\alpha : \Omega \to \mathcal{R}(\mathcal{S})$ such that Ω_n is mapped to $\mathcal{R}^{n+1}$. $\Sigma = (\mathcal{S}, \alpha)$ is called an *operator scheme* over $\mathcal{S}$.

Ω_n will contain the symbols for operators with arity n of the abstract syntax, and the image of an operator symbol $\omega \in \Omega_n$ under α will be the signature of that operator. Finite Ω-structures are adequate to express the abstract syntax of programming languages; thus, only finite Ω-structures are considered in this thesis. That is,

- $\mathcal{S}$ is finite,
- Ω_n is finite for all n, and
- $\Omega_n = \emptyset$ for all but a finite number of values of n.

Furthermore, because $\mathcal{S}$ is finite, then $R^n(\mathcal{S})$ is also finite and so $\omega\alpha \subseteq R^{n+1}(\mathcal{S})$ is finite for all $\omega \in \Omega_n$.

Continuing the example, define $\Omega_1 = \{const\}$, $\Omega_2 = \{plus\}$ and $\Omega_n = \emptyset$ for all other values of n. Define the Ω-structure, α, over Ω by

$$const\ \alpha = \left\{ \begin{array}{ll} (Int, & Expr) \\ (Real, & Expr) \end{array} \right\}, \qquad plus\ \alpha = \{(Expr, Expr, Expr)\}$$

The relations that form the image of an operator under α will be written using a function like notation:

$$const\ \alpha = \left\{ \begin{array}{lcl} Int & \rightarrow & Expr \\ Real & \rightarrow & Expr \end{array} \right\}, \qquad plus\ \alpha = \{Expr \times Expr \rightarrow Expr\}$$

An *abstract syntax,* $\mathcal{L} = (\Sigma, \mathcal{G}, \xi)$, is an operator scheme, Σ, over $\mathcal{S}$, together with an $\mathcal{S}$-indexed family of sets $\mathcal{G} = \{\mathcal{G}_s\}, s \in \mathcal{S}$, called the *generators* of the abstract syntax $\mathcal{L}$, and a symbol $\xi \in \mathcal{S}$ called the *start sort.* For notational simplicity $\{\mathcal{G}_s\}, s \in \mathcal{S}$, is usually written as just $\{\mathcal{G}_s\}$ and the range of indices is inferred from the context.

Using the operator scheme Σ, from above, and letting $\mathcal{G}_{Int}$ be the integers, $\mathcal{G}_{Real}$ be the real numbers and $\mathcal{G}_{Expr}$ be the empty set, then $(\Sigma, \{\mathcal{G}_s\}, Expr)$ is an abstract syntax.

Abstract syntax trees are defined as words generated by the abstract syntax. More precisely, $\Gamma_s(\mathcal{L})$, the set of *abstract syntax trees* of sort $s \in \mathcal{S}$ generated by an abstract syntax $\mathcal{L} = (\Sigma, \mathcal{G}, \xi)$ with $\Sigma = (\Omega, \alpha)$, an Ω-structure over the sorts $\mathcal{S}$, is recursively defined by:

(i) if $g \in \mathcal{G}_s$, then g is in $\Gamma_s(\mathcal{L})$, and

(ii) if $\omega \in \Omega_n$, $(s_1, \ldots, s_n, s) \in \omega\alpha$ and $t_1, \ldots, t_n$ are abstract syntax trees of sort $s_1, \ldots, s_n$, respectively, then $\omega(t_1, \ldots, t_n)$ is in $\Gamma_s(\mathcal{L})$.

If $T = \omega(t_1, \ldots, t_n) \in \Gamma_s(\mathcal{L})$, $\omega \in \Omega_n$, then T is said to have arity n, written as $arity\text{-}of(T)$.

Also, for an operator, $\omega \in \Omega_n$, such that $\omega\alpha$ contains the single tuple $(s_1, \ldots s_n, s_\omega)$, define the set, $\Gamma_\omega(\mathcal{L}) \in \Gamma_{s_\omega}(\mathcal{L})$, of abstract syntax trees over $\mathcal{L}$, rooted by ω, by

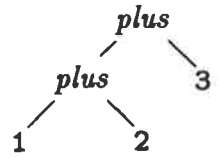
$$\Gamma_\omega(\mathcal{L}) = \{\omega(t_1, \ldots t_n); t_i \in \Gamma_{s_i}(\mathcal{L}), i = 1, \ldots, n\}$$

The set, $\Gamma[\mathcal{L}]$, of *proper abstract syntax trees* generator by $\mathcal{L}$ is the set $\Gamma_\xi(\mathcal{L})$, where ξ is the start symbol of the grammar. The set, $\Gamma(\mathcal{L})$, of all abstract syntax trees generated by $\mathcal{L}$ is given by:

$$\Gamma(\mathcal{L}) = \bigcup_{s \in \mathcal{S}} \Gamma_s(\mathcal{L})$$

Continuing with the example abstract syntax from before, $plus(plus(1,2),3)$ is a well-defined abstract syntax tree of type *Expr.* However, $plus(1)$ is not well-defined because

plus is a binary operator. Abstract syntax trees may be depicted using a graphical notation; specifically, $plus(plus(1,2),3)$ is drawn as



This notion of abstract syntax trees is quite general. In practice, the model will restricted in some fashion. For example, if the image of an operator symbol, ω, under α, is constrained to be a singleton set, then the sorts of the "subtrees" of abstract syntax trees labelled with ω are uniquely defined, as is the sort of trees rooted with ω. Syntax-directed editors generated by the Synthesizer Generator [Reps89a] manipulate trees generated from an abstract syntax of this kind. In this thesis, the restriction of $\omega\alpha$ to a singleton set is assumed in the interest of notational simplicity.

Restricting $\omega\alpha$ to a singleton set for all ω implies that the Ω-structure, α, is a function from operators to tuples. That is, $\alpha : \Omega_n \rightarrow \mathcal{S}^{n+1}$ where $\mathcal{S}^{n+1}$ is the set of $(n+1)$-tuples over $\mathcal{S}$. For a given programming language, extra operators may be required in an abstract syntax to compensate for the loss of expressive power resulting from this restriction on $\omega\alpha$. For example, in the simple abstract syntax modelling expressions, two operators would be required for constants. For example, *int-constant* and *real-constant* could be introduced, with signatures $Int \rightarrow Expr$ and $Real \rightarrow Expr$, respectively.

4.2 Tree domains and tree addresses

The preceding definition considers abstract syntax trees as words generated by an algebra. Often, a more tree-like notation is required, allowing particular "nodes" of abstract syntax trees to be selected and permitting the definition of various tree traversal operations. The above model can be equipped with these operations by constructing a tree domain from a word and defining the tree operations in terms of the associated tree domain and the inverse mapping back to the word. This section presents the basic definitions of tree domains and then establishes the relationship between abstract syntax trees and tree domains. The notation and definitions are taken from [Kron75].

Let $\mathcal{A}$ be a set. $\mathcal{A}^*$ is the set of all finite *sequences* of elements of $\mathcal{A}$. The empty sequence is denoted by ε. The concatenation of two strings α and β from $\mathcal{A}^*$ is denoted by $\alpha.\beta$. If α is a sequence over $\mathcal{A}$ and $Q = \{\alpha_1, \ldots, \alpha_n\}$ is a finite set of sequences over $\mathcal{A}$, then $\alpha.Q = \{\alpha.\alpha_1, \ldots, \alpha.\alpha_n\}$. The length of a string is denoted by $|\alpha|$.

Suppose that $|\alpha| = n$ and that we can write $\alpha = a_1.a_2 \ldots a_n$, where $a_i \in \mathcal{A}$. Define the following functions accordingly:

$$head(k, \alpha) = \begin{cases} \varepsilon, & \text{if } k = 0 \text{ or } n = 0 \\ a_1.a_2 \ldots a_k, & \text{if } 1 \leq k \leq n \\ \alpha, & \text{if } k \geq n \end{cases}$$

$$parent(\alpha) = head(|\alpha| - 1, \alpha), \alpha \neq \varepsilon$$

$$last(\alpha) = a_n, \alpha \neq \varepsilon$$

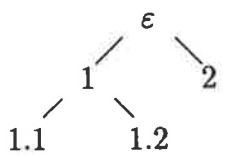
$$first(\alpha) = a_1, \alpha \neq \varepsilon$$

If $\mathbb{P}$ is the set of positive integers, a *position* is a sequence from $\mathbb{P}^*$; that is, a position is a sequence of non-negative integers.[1] For $p \in \mathbb{P}^*, p \neq \varepsilon$, and $last(p) > 1$, define

$$left\text{-}sibling(p) = parent(p).(last(p) - 1)$$

A *tree domain* is any non-empty subset of $\mathbb{P}^*$ closed under the parent and left-sibling functions. As will be seen later, a tree domain, $\mathcal{D}$, when associated with an appropriate labelling function, with domain $\mathcal{D}$, uniquely defines a tree.

For example, the set $\{\varepsilon, 1, 1.1, 1.2\}$ is a tree domain. It may be depicted pictorially as



[1] This definition deviates from the vocabulary used in [Kron75], which uses *node* rather than *position*. The term *position* is used in this thesis to distinguish a location in a tree from the contents of the node.

However, the set $\{\varepsilon, 1, 1.2\}$ is not a tree domain, because *left-sibling*(1.2) is not in the set. Similarly, $\varepsilon \in \mathcal{D}$, for any tree domain $\mathcal{D}$; otherwise, $\mathcal{D}$ would not be closed under the parent function.

If $\mathbf{D}$ is the set of tree domains, then a mapping, *dom*, from $\Gamma(\mathcal{L})$ to $\mathbf{D}$ can be defined, for $T \in \Gamma(\mathcal{L})$, as follows

$$dom(T) = \begin{cases} \{\varepsilon\} & \text{if } T \in \mathcal{G}_s, \text{ for some } s \in \mathcal{S} \\ \{\varepsilon\} & \text{if } T = \omega, \text{ for some } \omega \in \Omega_0 \\ \bigcup_{i=1}^{n} i.\, dom(T_i) & \text{if } T = \omega(T_1, \ldots, T_n), \text{ for some } \omega \in \Omega_n, n > 0 \end{cases}$$

The construction of $dom(T)$ for inner nodes, defined by the third alternative of the above definition, ensures that the function is closed under the *left-sibling* and *parent* relations; thus, *dom* is well-defined.

For example, the abstract syntax tree $plus(plus(1,2),3)$, depicted graphically on p. 74, is mapped by *dom* into the tree domain $\{\varepsilon, 1, 2, 1.1, 1.2\}$.

For a tree domain $\mathcal{D}$, and positions p and q, p is an ancestor of q if and only if p is a prefix of q. More precisely, for positions $p, q \in \mathcal{D}$, the ancestor relation is defined by

$$p \ anc \ q \iff \exists n \in \mathcal{D}, q = p.n$$

Positions p and q in a tree domain $\mathcal{D}$ are *independent*, denoted by $p \perp q$, if p is not an ancestor of q and q is not an ancestor of p. That is,

$$p \perp q \iff \overline{p \ anc \ q} \ and \ \overline{q \ anc \ p}$$

For an abstract syntax tree $T \in \Gamma(\mathcal{L})$, over an abstract syntax $\mathcal{L}$, and any position $p \in dom(T)$, the subtree of T, rooted at p, denoted by $T@p$, is defined by

$$T@p = \begin{cases} T & \text{if } p = \varepsilon \\ T_i@p' & \text{if } p = i.p',\ T = \omega(T_1, \ldots, T_k),\ \omega \in \Omega_k,\ i \leq k \end{cases}$$

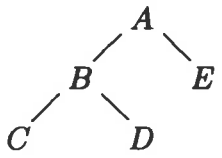
For example, use T to denote the abstract syntax tree $plus(plus(1,2),3)$, illustrated on p. 74. Then, $T@1$ is $plus(1,2)$, $T@1.1$ is 1 and $T@2$ is 3.

A label function for an abstract syntax tree $T \in \Gamma(\mathcal{L})$, is defined by

$$label\text{-}of(T) = \begin{cases} \omega, & \text{if } T = \omega(T_1, \ldots, T_k),\ \omega \in \Omega_k \\ g, & \text{if } T = g,\ g \in \mathcal{G}_s \text{ for some } s \in \mathcal{S} \end{cases}$$

Conversely, a tree domain, and labelling function over that tree domain that maps tree positions to operators, uniquely determines a tree. Appropriate restrictions on the labelling function are required to ensure that the resulting tree is a well-formed abstract syntax tree.

For example, the tree domain $\{\varepsilon, 1, 1.1, 1.2, 2\}$ and the labelling function $\varepsilon \mapsto A, 1 \mapsto B, 1.1 \mapsto C, 1.2 \mapsto D, 2 \mapsto E$ together determine the tree $T = A(B(C, D), E)$ which can be depicted as



Thus, *label-of*$(T@1.2) = D$, *label-of*$(T@1.1) = C$, *label-of*$(T@1) = B(C, D)$. If the labels are identified with the tree positions, then B *anc* C, A *anc* C, $C \perp D$ and $C \perp E$.

It is convenient to be able to talk about the sort of a node in an abstract syntax tree. For example, in the abstract syntax defined earlier, nodes labelled with *plus* could be said to have sort *Expr*. However, for a general abstract syntax, it is not possible to uniquely determine the sort of a node. For example, in the abstract syntax below, the sort of nodes labelled with *plus* is ambiguous.

$$\mathcal{S} = \{Int, Real\}, \qquad plus\ \alpha = \left\{ \begin{array}{lcl} Int \times Int & \rightarrow & Int \\ Real \times Int & \rightarrow & Real \\ Int \times Real & \rightarrow & Real \\ Real \times Real & \rightarrow & Real \end{array} \right\}$$

In the general case, the best that can be done is to test for membership in Γ_s, for $s \in \mathcal{S}$. However, if the restriction on p. 74 is valid and the signature of the operator $\omega \in \Omega_k$ is $\omega : s_1 \times \ldots \times s_k \rightarrow s_\omega$, then the sort of the operator ω is s_ω.

4.3 Tree patterns

Abstract syntax trees with some of their leaves replaced by variables are required for the specification of patterns. Let $\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathcal{S}}$ be an $\mathcal{S}$-indexed family of variable symbols. The set $\Gamma_s(\mathcal{L}, \mathcal{X})$ of tree patterns of sort s with variables from $\mathcal{X}$ is defined by:

(i) if $g \in \mathcal{G}_s$ then g is in $\Gamma_s(\mathcal{L}, \mathcal{X})$,

(ii) if $x \in \mathcal{X}_s$ then x is in $\Gamma_s(\mathcal{L}, \mathcal{X})$, and

(iii) if $\omega \in \Omega_n$, $(s_1, \ldots, s_n, s) \in \omega\alpha$ and $t_1, \ldots, t_n$ are abstract syntax trees of sorts $s_1, \ldots, s_n$, respectively, then $\omega(t_1, \ldots, t_n)$ is in $\Gamma_s(\mathcal{L}, \mathcal{X})$.

The set, $\Gamma(\mathcal{L}, \mathcal{X})$, of tree patterns generated by $\mathcal{L}$, with variables from $\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathcal{S}}$, is given by:

$$\Gamma(\mathcal{L}, \mathcal{X}) = \bigcup_{s \in \mathcal{S}} \Gamma_s(\mathcal{L}, \mathcal{X})$$

Only linear tree patterns are used in this thesis; a tree pattern P is linear if, for all variables x, x occurs at most once in P.

The function $dom : \Gamma(\mathcal{L}) \rightarrow \mathbb{D}$ can be extended to a mapping $dom : \Gamma(\mathcal{L}, \mathcal{X}) \rightarrow \mathbb{D}$ in the obvious fashion. It is useful to distinguish which of the leaves of a tree pattern are labelled with variable symbols. Thus, for a tree pattern P, define $var(P)$, the set of variable positions in P by

$$var(P) = \{p \in dom(P); \mathit{label\text{-}of}(P@p) = x, x \in \mathcal{X}_s\}$$

The tree pattern P is said to *match* the subject tree T at $n \in dom(T)$ if

- for all $p \in var(P), T@n.p \in \Gamma_s(\mathcal{L})$, where $P@p \in \mathcal{X}_s$, and
- for all $p \in dom(P) \setminus var(P), \mathit{label\text{-}of}(P, p) = \mathit{label\text{-}of}(T, n.p)$.[2]

4.4 Attribution of abstract syntax trees

An abstract syntax tree is able to capture the syntactic structure of a program. However, this is clearly insufficient for many purposes, including the incremental generation of object code. Some mechanism for expressing the semantics is required. It has proved appropriate in this work to use attribute grammars for representing program semantics.

Let $\mathcal{L} = (\Sigma, \mathcal{G}, \xi)$ be an abstract syntax, where $\Sigma = (\Omega, \alpha)$ is an Ω-structure.

First, sets of symbols that will serve as attribute names are defined. Let $\mathcal{A} = \{\mathcal{A}_s\}, s \in \mathcal{S}$, be a disjoint $\mathcal{S}$-indexed family of sets of symbols such that each $\mathcal{A}_s$ is the disjoint union of the sets Syn_s and Inh_s; furthermore, $Inh_\xi = \emptyset$, where ξ is the start

[2] If A and B are sets, then $A \setminus B$ denotes the set $\{a \in A;\ a \notin B\}$.

symbol of the abstract syntax. Let $Local = \{Local_\omega\}_{\omega \in \Omega}$ be a disjoint Ω-indexed family of sets of symbols. Syn_s and Inh_s contain the symbols that name the synthesised and inherited attributes for nodes in Γ_s. $Local_\omega$ is the set of symbols that name the local attributes of nodes labelled with ω. $\mathcal{A}'$, the complete set of attribute name symbols, is given by

$$\mathcal{A}' = \bigcup_{s \in \mathcal{S}} \mathcal{A}_s \cup \bigcup_{\omega \in \Omega} Local_\omega$$

If the abstract syntax satisfies the restriction of p. 74, meaning that, for all operators $\omega \in \Omega_n$, $\omega\alpha = \{(s_1, \ldots, s_n, s_\omega)\}$, then the sort of the operator ω is well-defined. Thus, the sets $Inh_\omega = Inh_{s_\omega}$ and Syn_{s_ω} are well-defined.

Next, the set of sorts and the associated families of generators are expanded. Let $\mathcal{S}'' = \mathcal{S} \cup \mathcal{S}'$ and let $\mathcal{G}'' = \mathcal{G} \cup \mathcal{G}'$ where $\mathcal{G}' = \{\mathcal{G}_s\}, s \in \mathcal{S}'$, is an $\mathcal{S}'$-indexed family of sets. $\mathcal{S}'$ extends the sorts available for attribute types, while $\mathcal{G}'$ provides generator sets for the new sorts. Attributes are typed with the *type-of* function that maps attribute symbols into symbols from $\mathcal{S}''$:

$$type\text{-}of\colon\ \mathcal{A}' \to \mathcal{S}''$$

Now, the attribute dependencies and evaluation functions for tree nodes labelled with some operator are defined. Let $\omega \in \Omega_n$ be an operator symbol of arity n such that $\omega\alpha = (s_1, \ldots, s_n, s_\omega)$, and then let

$$\begin{aligned}
occurrences(\omega) &= \{< 0, a >;\ a \in \mathcal{A}_{s_\omega} \text{ or } a \in Local_\omega\} \cup \\
&\quad \{< i, a >;\ i = 1, \ldots, n \text{ and } a \in \mathcal{A}_{s_i}\} \\
inputs(\omega) &= \{< 0, a >;\ a \in Inh_{s_\omega}\} \cup \\
&\quad \{< i, a >;\ i = 1, \ldots, n \text{ and } a \in Syn_{s_i}\} \\
outputs(\omega) &= \{< 0, a >;\ a \in Syn_{s_\omega} \text{ or } a \in Local_\omega\} \cup \\
&\quad \{< i, a >;\ i = 1, \ldots, n \text{ and } a \in Inh_{s_i}\}
\end{aligned}$$

The set of attribute occurrences at nodes labelled by ω are enumerated by the set $occurrences(\omega)$. These occurrences are partitioned into input and output occurrences by $inputs(\omega)$ and $outputs(\omega)$. For each output occurrence, a defining function, in terms of the

input occurrences, is required. This defining function establishes a graph of dependencies between the attribute occurrences of a node. Usually, the dependencies will be implied by the attribute definitions; however, in this model, the dependencies must be explicitly defined in order to specify the domain and range of the definition functions. Thus, if $\mathcal{P}(occurrences(\omega))$ denotes the set of permutations over $occurrences(\omega)$, then for each operator, ω, the function $dependencies_\omega$, of the form

$$dependencies_\omega\colon\ outputs(\omega) \rightarrow \mathcal{P}(occurrences(\omega))$$

must be specified. Then, for each operator symbol ω, and for each occurrence $< k, a >$ from $outputs(\omega)$ such that $dependencies_\omega(< k, a >) = a_1.a_2 \ldots a_n,\ a_i \in occurrences(\omega)$, an *attribute definition function*, $value_{\omega,a,k}$, must be specified, where

$$value_{\omega,a,k}\colon\ \mathcal{G}''_{type\text{-}of(a_1)} \times \ldots \mathcal{G}''_{type\text{-}of(a_n)} \rightarrow \mathcal{G}''_{type\text{-}of(a)}$$

Furthermore, for each sort $s \in \mathcal{S}''$, such that $\mathcal{G}'' \neq \emptyset$, and each attribute $a \in Syn_s$, a function

$$eval_a : \mathcal{G}'' \rightarrow \mathcal{G}''$$

must be specified. Usually, this will be the identity function.

An attribute grammar is therefore associated with an abstract syntax by specifying the attribute symbols, the new sorts for attribute values and the attribute definition functions as defined above. The evaluation of an attribute at a node in an abstract syntax tree is defined by the *eval* function; for $T \in \Gamma(\mathcal{L})_s$, $p \in dom(T)$, and $a \in A_s \cup Local_\omega$, this function is defined by

$$eval(T,p,a) = \begin{cases} eval_a(g), & \text{if } T@p = g,\ g \in \mathcal{G}_s,\ a \in Syn_s \\ apply_{\omega,a}(T,p,0), & \text{if } a \in Syn_s \cup Local_\omega,\ \omega = label\text{-}of(\mathrm{T@p}) \\ apply_{\mu,a}(T,p',j), & \text{if } a \in Inh_s,\ p = p'.j,\ label\text{-}of(T@p') = \mu \end{cases}$$

in which

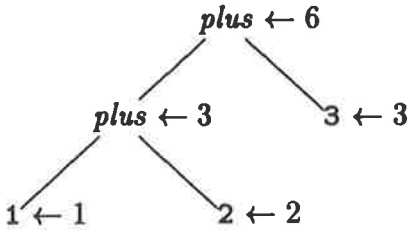
$$apply_{\omega,a}(T,p,k) = value_{\omega,a,k}(\phi_1, \ldots, \phi_m)$$

$$dependencies(< k, a >) =< p_1, a_1 > \ldots < p_m, a_m >$$

$$\phi_i = \begin{cases} eval(T,p,a_i), & \text{if } p_i = 0 \\ eval(T,p.p_i,a_i) & \text{if } p_i > 0 \end{cases}$$

$$eval(T, \varepsilon, sum) = 6$$
$$eval(T, 1, sum) = 3$$
$$eval(T, 2, sum) = 3$$
$$eval(T, 1.1, sum) = 1$$
$$eval(T, 1.2, sum) = 2$$

(a) instances of the *sum* attribute



(b) decoration with attribute instances

Figure 4.1 Evaluating instances of attributes.

For example, define an abstract syntax by $\mathcal{S} = \{Int\}$, $\Omega_1 = \{const\}$, and $\Omega_n = \emptyset$ for all other values of n. Define the Ω-structure, α, over Ω by

$$plus\ \alpha = \{Int \times Int \rightarrow Int\}$$

Let the generator set of *Int* be the set of natural numbers. A simple attribute grammar can then be defined over this abstract syntax by the following specifications.

$$Syn_{Int} = \{sum\}$$
$$Inh_{Int} = \emptyset$$
$$Local_{plus} = \emptyset$$
$$outputs(plus) = \{< 0, sum >\}$$
$$dependencies_{plus}(< 0, sum >) =< 1, sum >, < 2, sum >$$
$$value_{plus,sum,0}(x, y) = x + y$$
$$eval_{sum}(x) = x, \text{ for all natural numbers, } x.$$

Then, for each position of the abstract syntax tree $T = plus(plus(1, 2), 3)$, the *sum* attribute can be evaluated. Figure 4.1(a) enumerates the values of each instance of the *sum* attribute in T. The same instance values are illustrated graphically in Figure 4.1(b).

An attribute grammar over an abstract syntax $\mathcal{L}$ is *non-circular* if the *eval* function is well-defined for all abstract syntax trees $\Gamma[\mathcal{L}]$. In particular, for ordered attribute grammars, the set TDP(ω) can be constructed for each operator ω from $dependencies_\omega$, as shown in [Kastens80]. Then, because TDP(ω) is acyclic for each ω, *eval* can be shown to be well-defined. However, this construction is beyond the scope of this thesis.

4.5 A simple model of editing

The input to an incremental instruction selection algorithm will consist of

- the initial source code,
- the initial object code, and
- the edit operation.

In this work, source code is represented as an abstract syntax tree and object code may be considered as a one-dimensional array of instructions. Intermediate information requirements, including the source to object code mapping, will be inferred from the algorithm. Edit operations, however require precise definition.

The simplest edit operation on an abstract syntax tree is the replacement of a complete subtree. More complicated editing paradigms, based on a model of tree transformations that preserve unchanged components of mutated program fragment, have the potential to reduce the extent of recompilation. However, this work considers only complete subtree replacement.

The substitution of t for $T@p$, where $p \in dom(T)$, is denoted $T' = T_{p \leftarrow t}$, and is defined by constructing a tree domain and a labelling function $\gamma : dom(T') \rightarrow \Omega$, as follows

$$dom(T') = \{q \in dom(T);\ \overline{p\ anc\ q}\} \cup p.dom(t)$$

$$\gamma(n) = \begin{cases} T@n & \text{if } \overline{p\ anc\ n} \\ t@q & \text{if } p\ anc\ n \text{ and } n = p.q \end{cases}$$

The replacement of the subtree at p of the proper abstract syntax tree T by the abstract syntax tree t is defined to be $T_{p \leftarrow t}$ if this results in a well-formed, proper abstract syntax tree.

The *size* of a replacement is defined to be the number of nodes in the new subtree.

```
1: sort EXPRESSION;
2: operator plus: EXPRESSION EXPRESSION → EXPRESSION;
3: operator minus: EXPRESSION EXPRESSION → EXPRESSION;
4:
5: type IDENTIFIER is
6:   regexp "[a-zA-Z](_?[a-zA-Z0-9])*";
7: end;
```

Figure 4.2 Declaration of a sort, an operator and a generator set.

4.6 A language specification formalism

A language for the specification of programming languages has been based on the algebraic model of abstract syntax. In this language specification language (LSL), an abstract syntax specification consists of

- declarations of the generator sets,
- declarations of the sorts, and
- declarations of the operators and their signatures.[3]

An extract from a language definition is shown in Figure 4.2. The declaration on line 1 asserts the existence of a sort called `EXPRESSION`. The generator set of the sort `EXPRESSION` is the empty set. Line 2 asserts the existence of a binary operator called `plus` such that

$$\texttt{plus}\ \alpha = \{\texttt{EXPRESSION} \times \texttt{EXPRESSION} \rightarrow \texttt{EXPRESSION}\}$$

A generator set is declared on lines 5–7 of Figure 4.2. This declaration asserts the existence of a sort called `IDENTIFIER` such that the generator set of this sort is the set of strings that may be derived from the given regular expression.

The context-free grammar of a programming language is called the concrete syntax. Concrete syntax is specified as a set of context-free productions. A mapping from strings over the context-free grammar into abstract syntax trees over the abstract syntax is specified by the association of an abstract syntax tree-valued expression, called a tree constructor, with each production of the concrete syntax.

[3] The signature of an operator ω, is the image of ω under an Ω-structure.

Expression ::= *Expression* "+" *Expression* { **plus** (*Expression*$_1$, *Expression*$_2$) }
Expression ::= *Expression* "-" *Expression* { **minus** (*Expression*$_1$, *Expression*$_2$) }

Figure 4.3 Productions from the concrete syntax.

Two productions from a concrete syntax specification are shown in Figure 4.3. The tree constructor associated with the first production, indicates that binary expressions involving the infix "+" operator, are translated into binary abstract syntax tree nodes that are labelled with the `plus` operator.

The tree constructors implicitly associate a sort from the abstract syntax with each production of the concrete syntax. For the first production of Figure 4.3, the abstract syntax tree fragment is labelled with the `plus` operator. Thus, the sort of the fragment is `EXPRESSION`. The first appearance of a non-terminal N, on the left hand side of a context-free production defines the sort of the non-terminal to be the sort of this production. The sort of all subsequent productions, with left-hand side N, must be the same as the sort of N.

Optionally, an attribute grammar over the abstract syntax can be specified. Typed attributes of sorts, and local attributes of operators, may be declared. For each output attribute occurrence, the defining function is specified as a simple expression.

The declarations of two synthesised attributes of the sort `EXPRESSION` are shown in Figure 4.4. Thus, for each operator of type `EXPRESSION`, defining functions are required for both of these attributes. The defining functions for the output occurrences of the `is_constant` and `value` attributes for the `PLUS` operator are illustrated.

To date, specifications for the C, Ada, Modula2 and VHDL programming languages have been have been written in LSL, as has the specification of the small language used to prototype the incremental code generator that is described in Chapter 7. Of these, a full attribute grammar exists only for the latter; the descriptions of C, Ada, Modula2 and VHDL contain only specifications of the abstract syntax and the context free syntax of these languages. An extract from the specification of this small language appears in Appendix B.

```
sort EXPRESSION is
  syn is_constant: BOOLEAN;
  syn value: INTEGER;
end;
operator PLUS is
  $$is_constant = $1.is_constant and $2.is_constant;
  $$value =
    if $$.is_constant then $1.value + $2.value else 0 fi;
end;
```

Figure 4.4 Attribute declaration and definition.

4.7 Conclusions

An algebraic model of abstract syntax has been constructed in which abstract syntax trees are considered as words freely generated from the abstract syntax. Attribute grammars are used to capture the static semantics of programs represented as abstract syntax trees. Program editing is modelled by the replacement of subtrees in abstract syntax trees. The derivation of the BURS-based incremental instruction selection algorithm in Chapter 5 and the greedy parallel code generation algorithm in Chapter 6 are based on this model.

Abstract syntax tree are the canonical program representation used by the MultiView distributed integrated programming environment, described in Chapter 3. The implementation of abstract syntax trees described in Section 7.1.1 is firmly grounded in the algebraic model of abstract syntax trees presented in this chapter.

The MultiView language specification language (LSL) is also based on this algebraic model of abstract syntax. The LSL compiler described in Section 7.1.2 then generates the language-dependent components of the MultiView implementation from a formal LSL specification of the programming language.

Chapter 5

Incremental instruction selection

Chapter 2 described several techniques that have been used for fine-grained incremental compilation. However, none of these techniques exhibit a systematic approach to incremental instruction selection. DICE, for instance, made use of the Portable C Compiler code generator (see Section 2.4.2), while SEP used a naive template-based approach to emit P-code (see Section 2.4.5). A systematic approach to incremental instruction selection should improve of the quality of the emitted code, reduce the difficulty of retargeting the code generator and, ideally, allow the provision of tools to aid retargeting.

The construction of fast, retargetable code generators for traditional compilers has been a goal of compiler research for at least two decades. Since the work of Graham and Glanville [Glanville77, Glanville78], a number of approaches to generating code generators from descriptions of target architectures have been proposed. To date, such an approach has not yet been applied to incremental code generation.

Section 5.1 surveys several retargetable instruction selection techniques in which the architecture-dependent components of the code generator are generated from a description of the target architecture. Of these methods, bottom-up rewrite system (BURS) based instruction selection is argued to be the most suitable foundation of an incremental instruction selection algorithm.

The first pass of the BURS instruction selection algorithm labels nodes of the subject tree with pattern matcher states. Section 5.2 considers the issue of incrementally restoring a consistent labelling after a subtree of the subject tree has been replaced. The formal model of abstract syntax from Chapter 4 is the foundation for this analysis.

Abstract syntax trees are a suitable canonical program representation in an integrated programming environment. However, they contain insufficient information for BURS pattern matching. An intermediate representation must be constructed from the abstract

syntax tree, so that all necessary information is captured in the labelling of the tree. Section 5.3 presents a suitable intermediate representation that may be inferred in a single top-down traversal of a subtree and realised with a simple relabelling of nodes in the abstract syntax tree. The remainder of this chapter presents the BURS-based incremental instruction selection algorithm, drawing on the analysis of Section 5.2 and the relabelling transformation of Section 5.3.

5.1 Retargetable code generation

The contemporary compiler writer has a well stocked chest of tools available to automatically generate many of the components that make up a compiler. Typically, these tools generate language-dependent and target-dependent compiler components from formal specifications. Front-ends have proved particularly amenable to such treatment; however, somewhat ironically, these are probably the least difficult parts of a compiler to implement by hand. For example, scanner and parser generators are able to generate the lexical and syntactic analysis phases from definitions based on regular expressions and context free grammars, respectively. A variety of techniques have proved suitable for the specification of static semantic analysis and the subsequent generation of semantic analysers. For example, the Eli system [Gray92] uses an attribute grammar specification of the static semantics of a language from which an attribute evaluator is derived.

Code generation is the final phase of compilation. The code generator emits sequences of machine instructions, either raw machine code or some low level code, such as P-code, which is then interpreted. Input to the code generator is typically some form of intermediate program representation, such as quadruples or expression trees. This intermediate representation is derived from both the program source and information inferred during semantic analysis.

A code generator is considered *retargetable* is there is a well–defined procedure by which it can be altered to generate code for a new architecture [Glanville77]. In practice, retargetability implies that a tool is available to generate most, if not all, of the machine dependent parts of the code generator from some formal description. Commonly

used formalisms include register transfer language (RTL) [Stallman94], machine grammars [Glanville77] and rewrite system specifications [Hatcher85, PelegríLlopart88a, Aho89].

In this section, a technique amenable to the derivation of an incremental recompilation algorithm is chosen from the existing approaches to constructing retargetable code generators in non-incremental compilers. The ensuing survey examines several such techniques and discusses their suitability as the basis for an incremental algorithm.

Henry's taxonomy of retargetable code generation [Henry84] methods partitions work into that predating Graham and Glanville's contribution and work since this contribution. While considerable work was done in the earlier period, such as the Production Quality Compiler Compiler project [Cattell79, Cattell80], only several of the more mature techniques from the later period are considered here. The survey below first considers the seminal Graham-Glanville approach, examining its suitability for incremental instruction selection. Subsequently, several techniques based on tree pattern matching are considered, from which a technology is chosen on which to base an incremental instruction selection algorithm.

The assessment of the suitability of code generation methods for incremental code generation requires consideration of the goals enumerated in Section 2.2. Of these goals, speed of compilation, code quality and storage overhead are of particular interest in an assessment. Consistency and separability are more traits of the derived incremental algorithm, rather than of the base code generation technology. Each of the technologies considered here generates a machine-specific code generator from some machine description, and so the retargetability goals (see p. 13) are implicitly satisfied.

5.1.1 Graham-Glanville code generators

Graham-Glanville code generators [Glanville77, Glanville78, Henry84] exploit LR parser technology [Aho86] in order to generate object code. A grammar that represents the instruction set of the target architecture is used to construct a parser that analyses a

linear intermediate program representation. Object code is emitted according to the resulting derivation sequence. Retargeting the code generator to a new architecture involves the creation of a machine description grammar for the new target.

The analysis phase of compilation produces a set of tables and an *Abstract Program Tree* (APT) to represent the executable portion of the source program. This *intermediate representation* (IR) must have the *locality* property [Glanville77, Section 3.1]; that is, the meaning of an intermediate expression must not be affected by the context in which it appears. An immediate consequence of this is that the representation must be low level; implementation decisions, such as storage allocation, must have been made prior to the generation of the APT.

As already mentioned, the Graham-Glanville approach to code generation is based on traditional LR parsing techniques. Such parsers are unable to directly analyse trees; in particular the input to a parser of this kind must be linear. A Polish prefix representation of the APT serves as the linear IR that is the input to Graham-Glanville style code generators.

Target architectures are described using the *Target Machine Description Language* (TMDL). A TMDL specification consists of:

- a list of target machine registers,
- declarations of terminals and non-terminals, and
- a description of the instructions of the target computer.

An extract from a TMDL description is shown in Figure 5.1, in which λ represents the null non-terminal symbol. Nonterminal symbols in TMDL descriptions, in general, represent registers. The terminal symbols must include all the operators of the IR. Target machine instructions are related to the IR by the productions of the target computer description. For example, instruction description (4) in Figure 5.1 contains a production that enables the LR parser to reduce an occurrence of `k` to the non-terminal "`r`". This reduction models the loading of a constant value into a register and so is associated with the `load` machine instruction. Because `r.1` does not appear on the right-hand side of the production, reductions by rule (4) cause the allocation of a value from the

```
$registers
  $allocatable [r0,r1,r2,r3,r4]
  $dedicated [r5,r6,r7]

$symbols
  $nonterminals
    r = r0,r1,r2,r3,r4,r5,r6,r7
  $terminals
    k   0...32767
    +   binary
    ↑   unary
    :=  binary

$instructions
    r.2   ::=  (+ ↑ + k.1 r.1 r.2)        add     r.2,k.1,r.1   (1)
    r.1   ::=  (+ r.1 ↑ + k.1 r.2)        add     r.1,k.1,r.2   (2)
  ...
    λ     ::=  (:= ↑ + k.1 r.1 r.2)       store   r2,*k.1,r.1   (3)
  ...
    r.1   ::=  ( k.1 )                    load    r.1,=k.1      (4)
  ...
```

Figure 5.1 Extract from a TMDL description [Glanville77, Figure 3.1].

set {r0,r1,r2,r3,r4,r5,r6,r7}. This value is then pushed onto the parser stack as a semantic value.

From the TMDL description, a code generator based on a deterministic shift-reduce parser is constructed. As mentioned above, input to the parser is the Polish prefix representation of the APT. As reductions are signalled, semantic actions are invoked and code is emitted. Because the context free grammar in the TMDL description is likely to be ambiguous, a simple heuristic is used to resolve ambiguities while parsing. Rules are ordered by the preprocessor in a "best instruction first" sequence and are subsequently tested, at code generation time, in that order. Semantic information is recorded on the parser stack along with the parser state. For example, when a reduction to a nonterminal that denotes a register is signalled, either a free register is allocated, or an operand register is reused and the register identifier is pushed onto the stack.

For example, Figure 5.2 shows a fragment of an APT whose equivalent Polish prefix form is "+ ↑ + k k k". A derivation sequence that reduces this fragment to the

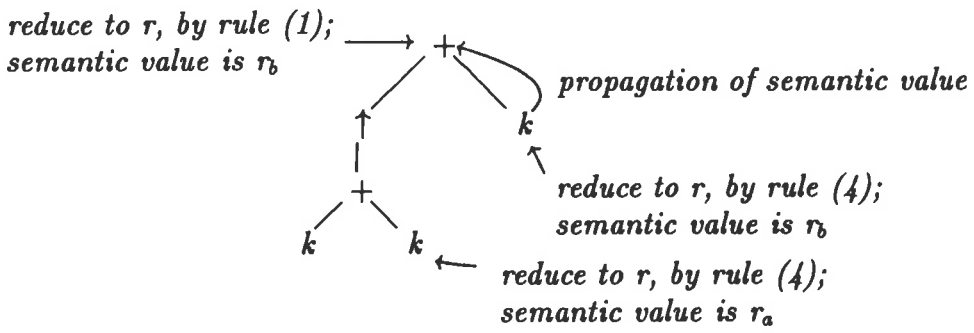


Figure 5.2 Reduction by a Graham-Glanville code generator.

non-terminal **r** can be constructed from the instruction descriptions of Figure 5.1. The appropriate reductions are indicated graphically in Figure 5.2.

Register allocation occurs during reduce operations. After a reduction to a non-λ nonterminal, if the result is not semantically linked to another register in the pattern by sharing a common suffix, a free register is allocated. If the result is linked to a register in the pattern, then the same register is used. In the example of Figure 5.2, using the machine description of Figure 5.1, the result register of rule (4) is not semantically linked to a right-hand side register (since there is no register on the right-hand side), and so a free register is allocated and pushed onto the stack. However, rule (1) semantically links the result register with a right-hand side register (the rightmost one, in fact), as indicated by the common subscript. Thus, the register value is propagated up the derivation tree as shown in Figure 5.2.

The propagation of semantic information associated with a non-terminal symbol is equivalent to the evaluation of a single synthesised attribute over the strings derivable from the machine description grammar. Ganapathi and Fischer extended the use of attributes in code generators based on LR parsing [Ganapathi85]. In their *affix grammar driven* approach to code generation, the target machine is described by an affix grammar, allowing symbols to be associated with a finite set of synthesised and inherited attributes. Attribute evaluation can occur as the parser takes a reduction step and is able to influence later reductions. Consequently, information flows about the intermediate representation, albeit in a manner constrained by the left to right nature of the parsing algorithm.

The LR parsing based approaches to code generation exhibit a left bias. That is, in the case of a binary operator in the IR stream, reduction of the left operand will occur before the right operand is examined. Thus, code selected for the left operand cannot be influenced by the right operand, leading to suboptimal code in many instances.

An approach to both eliminating the left bias of LR parsing and also the systematic resolution of ambiguities is described in [Christopher84]. Earley's parsing algorithm [Earley70] is used to effectively generate all derivation sequences of the linear intermediate code string. Dynamic programming is used to select a least cost sequence and object code is emitted according to this least cost derivation sequence.

An incremental derivative of the Graham-Glanville code generation algorithm would demand, at the very least, replacing the LR parser with an incremental LR parser. An intermediate representation, functionally equivalent to the linear Polish prefix form, would thus be required to be maintained incrementally. In an environment using abstract syntax trees as the canonical representation, as is assumed in this work (see p. 17), the cost of maintaining this representation is likely to be high, both in time and storage. This approach to incremental code generation has not been further investigated due to the apparent greater suitability of tree rewriting methods. In addition, the tree rewriting based techniques do not suffer from the left bias inherent in LR parsing based methods. For similar reasons, neither the affix grammar approach, nor the approach based on Earley's parser and dynamic programming, are considered further as the basis for an incremental method for instruction selection.

5.1.2 Instruction selection by tree pattern matching

In the previous section, the Graham-Glanville algorithm was abandoned as the basis for an incremental code generator. The primary reason for this was the requirement for a linear intermediate representation, or an equivalent structured form amenable to LR parsing. Algorithms that directly operate on a tree-based intermediate representation appear to be more appropriate for incremental code generation. In particular, a technique which operated directly on abstract syntax trees would be well suited to implementing an

1.	*lvalue*	$\rightarrow$	$\textit{local-var}_n$	add %sp,n,%r$_o$
2.	*rvalue*	$\rightarrow$	*lvalue*	ld [%r$_i$],%r$_o$
3.	*rvalue*	$\rightarrow$	$\textit{constant}_n$	ld n,%r$_o$
4.	*rvalue*	$\rightarrow$	*plus (rvalue, rvalue)*	add %r$_{i1}$,%r$_{i2}$,%r$_o$
5.	*rvalue*	$\rightarrow$	$\textit{plus (rvalue, constant}_n\textit{)}$	add %r$_i$,n,%r$_o$

Figure 5.3 Extract from a tree-pattern based architecture description.

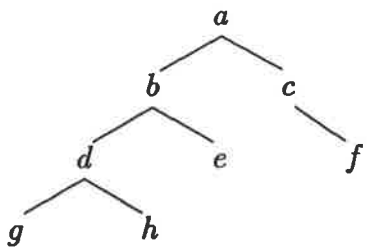
incremental code generator in a programming environment which used abstract syntax trees as the canonical program representation.

Tree pattern matching has been used as the basis of a number of code generator generators. In this approach, code generators are derived from an architecture description that relates patterns in the intermediate code tree to the desired object code. Figure 5.3 shows an extract from such a description. The architecture is defined by a set of productions and the right-hand side of each production is a tree pattern. The left-hand side of a production is a symbol which is distinct from the operators of the intermediate code trees. Associated with each production is a fragment of object code. Generation of object code from an intermediate code tree commences by finding a cover of the tree with patterns from the productions of the architecture description.

A cover of an intermediate code tree, T, from a set of tree productions is a set of tuples of the form $(g \rightarrow P, n)$ where g is a non-terminal symbol, P is a tree pattern as defined in Section 4.3 (see p. 77), $g \rightarrow P$ is a tree production and n is a position in T such that

(i) if $p \in \textit{dom}(T)$ and $\omega = \textit{label-of}(T@p)$, then there is exactly one tuple, $(g \rightarrow P, n)$, in the cover, such that $n \ \textit{anc} \ p$ and $\textit{label-of}(P@q) = \omega$, when $p = n.q$, and

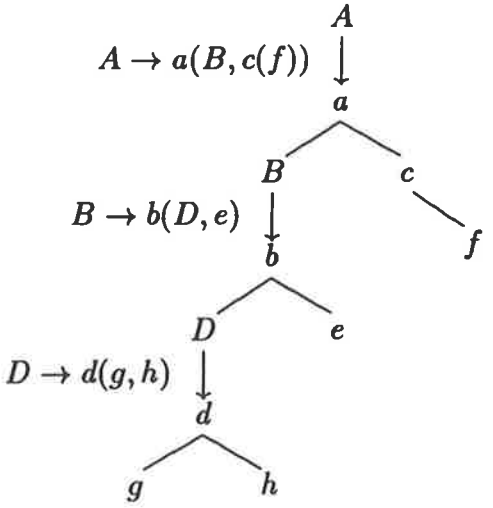
(ii) if $(g \rightarrow P, n)$ is in the cover and $q \in \textit{dom}(P)$, such that $X_q = \textit{label-of}(P@q)$ is a non-terminal symbol, then there is exactly one tuple of the form $(X_q \rightarrow Q, n.q)$ in the cover.



(a) the subject tree

$(A \rightarrow a(B, c(f)), \quad \varepsilon)$
$(B \rightarrow b(D, e), \quad 1)$
$(D \rightarrow d(g, h), \quad 1.1)$

(b) a cover of the tree



(c) covering the subject tree

Figure 5.4 Covering a subject tree with a set of tree patterns.

A cover of the subject tree T in Figure 5.4(a), with the following tree productions:

$$A \rightarrow a(B, c(f))$$

$$B \rightarrow b(D, E)$$

$$D \rightarrow d(g, h)$$

is shown in Figure 5.4(b); the same cover is illustrated graphically in Figure 5.4(c). Note that capital letters are used for non-terminal symbols and lower case letters for terminal symbols in the above productions. Consider the position 1.1 of the subject tree. If $\gamma = d(g, h)$, then $(D \rightarrow \gamma,\ 1.1)$ and $\textit{label-of}(\gamma@\varepsilon) = d$. Furthermore, no other pattern in the cover contains the operator d. Thus, the first condition for a cover is satisfied at the position 1.1 of the subject tree. A similar construction applies for every other position in the subject tree. If $\delta = b(D, e)$, then the tuple $(B \rightarrow \delta,\ 1)$ is in the cover and $\textit{label-of}(\delta@1)$ is the non-terminal D. There is exactly one tuple in the cover at 1.1, that is $(D \rightarrow d(g, h),\ 1.1)$. Thus, the second condition for a cover is satisfied for the tuple $(B \rightarrow b(D, e),\ 1)$. A similar construction applies for every other non-terminal occurrence in the patterns of the cover.

Figure 5.5 shows two alternative covers of the same intermediate code tree, using the patterns in Figure 5.3. Nodes at which a pattern has been matched are annotated

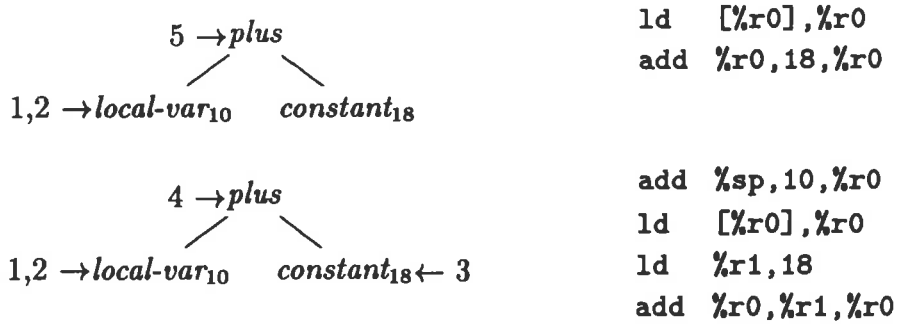


Figure 5.5 Two covers for an intermediate code tree.

with the number of the matching rule. Consider the uppermost cover in Figure 5.5. The leftmost leaf has been matched by the right-hand side of rule 1 in Figure 5.3; the subtree may thus be replaced by a leaf labelled with *lvalue* which is the left-hand side of rule 1. This new node is then matched by rule 2, causing it to be replaced with *rvalue*. Finally, the root node is matched by rule 5 and consequently it is replaced by *rvalue*.

Given a cover that enables the reduction of the intermediate code tree into a symbol, often denoting a register of the target machine, object code is then emitted during a top-down traversal of the covered intermediate code tree. Again consider the first cover of Figure 5.5. The reduction of the leftmost leaf to *lvalue*, as a result of applying rule 1, results in the emission of "`add %sp,10,%r0`". The subsequent application of rule 2 causes "`ld [%r0],%r0`" to be emitted. Finally, the application of rule 5 at the root emits "`add %r0,18,%r0`".[1]

Two features distinguish code generators based on tree pattern matching:

- the nature of the pattern matcher used to generate covers, and
- the methods used to select the "best" cover from all the possible covers of the intermediate code tree.

The *tree matching problem* consists of a finite set of tree patterns $p_1, \ldots, p_k$ over some alphabet, and a subject tree t over that alphabet. A solution to the matching problem is a list of all the pairs (n, i), where n is a node in t and p_i matches at n [Hoffmann82].

[1] Register allocation is conveniently ignored in this example, and for much of this thesis.

Tree pattern matchers operate either in a *top-down* or a *bottom-up* fashion. A bottom-up matcher attempts to find, at each point in the subject tree, all patterns and parts of patterns that match at this point. In such a pattern matcher, the match set associated with a node depends only on the label of the node and the match sets assigned to its subtrees.

[Hoffmann82] describes a top-down tree pattern matcher, subsequently used in the Twig system (see p. 97), that reduces tree matching to string matching by considering the subject tree in terms of the paths from the root to the leaves. The state assigned to a node by a top-down matcher depends only on the label of the node and the states assigned to its ancestors.

Top-down pattern matchers are inherently less powerful than bottom-up matchers [PelegríLlopart88a, Section 2.2]; that is, the class of patterns recognisable by a top-down tree automaton is smaller than the class recognisable by a bottom-up automaton. Furthermore, bottom-up matching is faster than top down matching; a sufficiently powerful bottom-up matcher can assign states, representing match sets, to nodes of the subject tree in time proportional to the size of the subject tree.

Finding a cover of a subject tree by a set of tree reducing productions is a different, but related, problem to that of finding all the matches of a subject tree by a tree pattern set. The pattern matchers of the code generators described here are adaptations of tree pattern matching algorithms inferring the list of possible matches of a subject tree, which have been adapted to find covers of the subject tree.

For a given intermediate code tree, there will usually be several covers by patterns from the architecture description. In particular, this will arise where the description includes patterns for a general case and several patterns to recognise specific restrictions of the general case. Consider, for example, the extract from an architecture description in Figure 5.3. The pattern of rule 4 is a general pattern for the addition operator. Rule 5, on the other hand, is a more specific rule for this operator when the right operand is a constant. When the pattern of rule 5 matches, better code will be emitted. This is reflected by the two covers shown in Figure 5.5. The first cover uses the more specific

rule, resulting in object code that uses three instructions and one register to implement the addition. The second cover uses the more general rule, resulting in four machine instructions and using two registers. To produce high quality code, a code generator using such patterns must employ some method to select the most appropriate cover.

In each of the three tree pattern matcher based code generators described below, ambiguities are resolved by associating a cost metric with each of the patterns of the architecture description. The total cost of a cover is the sum of costs of the individual patterns that make up the cover. The least cost cover is then taken to be the best cover. Locally optimal code is defined to be the code resulting from a least cost cover of the intermediate code tree. Thus, object code is considered optimal if it is locally optimal with respect to a particular pattern set and associated cost metric. If several least cost covers are found, an arbitrary choice may be made from among them.

The derivation of an incremental version of a tree pattern based code generation algorithm will depend on inferring incremental versions of both the tree pattern matching algorithm and the ambiguity resolution strategy. Ideally, the incremental pattern matcher should be able to operate on abstract syntax trees, so that the expense of incrementally maintaining an intermediate representation may be avoided.

The Twig system combines top-down pattern matching with dynamic programming to select a least cost cover of intermediate code trees [Aho85, Aho89]. Rewrite rules, in which the right-hand side is a tree pattern and the left-hand side a symbol, are associated with a cost and an action. The cost may be an arbitrary expression and, in the application of Twig to code generation, the action is a fragment of object code. Figure 5.6 shows a collection of rules, costs and actions of the kind recognised by Twig.

The Twig preprocessor generates a top-down pattern matcher from the patterns of the architecture description. The matching algorithm employed is adapted from Hoffmann and O'Donnell's top-down tree pattern matcher [Hoffmann82]. A top-down tree automaton assigns states to nodes of the subject tree. The states of a node, n, denote the sets of path strings and partial path strings that match at n. Bit strings are generated at each node in order to track partial matches and determine the nodes that start

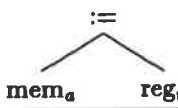
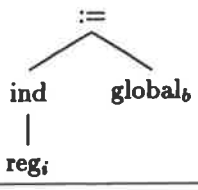
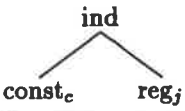
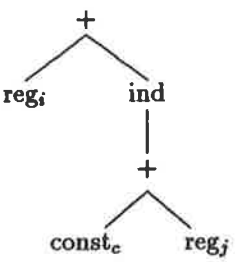
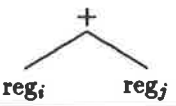
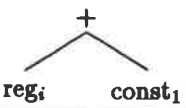
	Target	Pattern	Cost	Instruction
(1)	$reg_i \rightarrow$	$const_c$	2	`mov #c,ri`
(2)	$reg_i \rightarrow$	mem_a	2	`mov a,ri`
(3)	$\lambda \rightarrow$		2 + cost.reg_i	`mov ri,a`
(4)	$\lambda \rightarrow$		2 + cost.reg_i	`mov b,*ri`
(5)	$reg_i \rightarrow$		2 + cost.reg_j	`mov c(rj),ri`
(6)	$reg_i \rightarrow$		2 + cost.reg_i + cost.reg_j	`add c(rj),ri`
(7)	$reg_i \rightarrow$		1 + cost.reg_i + cost.reg_j	`add rj,ri`
(8)	$reg_i \rightarrow$		1 + cost.reg_i	`inc ri`

Figure 5.6 Twig tree-rewriting rules [Aho89, Table I].

complete pattern matches. For each node, n, and tree template, t_i, of the pattern set, a bit string, $n.b_i$, of length equal to one more than the length of the longest path in t_i is required.[2] Nodes that start complete pattern matches must be reduced to the appropriate goal symbol for consideration in pattern matches that start at an ancestor.

[2] Details of the creation and usage of the bit strings are not important to the present discussion; of interest here is the amount of information that must be associated with each node.

Dynamic programming is used to determine if a reduction is potentially part of a least cost cover. At each node of the subject tree, a vector of costs is calculated; $n.cost(l)$ is the cost of the cheapest match of some rule $l \leftarrow t_i$ that reduces to l. The index of the rule that results in the cheapest reduction to l is stored in $n.rule(l)$. If $l \leftarrow t_k$ matches at n and if $cost(t_k, n) < n.cost(l)$, where $cost(t_k, n)$ is the cost of rule t_k matching at n, then $cost(t_k, n)$ is assigned to $cost(l)$ and $n.rule(l)$ becomes k.

The use of dynamic programming ensures that Twig is successful at generating good quality object code. Furthermore, [Aho89] reports good compilation time when compared to the Portable C Compiler. However, a number of aspects of the Twig code generation algorithm suggest that it is not a good choice as the basis for an incremental code generator.

Firstly, Twig associates a large amount of state information with each intermediate tree node. [Aho89] indicates that Twig requires 200 bytes for each node in the intermediate tree. A large amount of information would have to be stored in order to satisfy the incrementalism condition; at the very least, the cost vectors associated with nodes would have to persist across recompilations. Consider the case where the left operand of a binary node has been replaced. The top-down nature of the pattern matching automaton ensures that the state assignments and cost vectors of nodes in the right operand are unchanged. Furthermore, the cost vectors from the right operand will be required to evaluate possible new reductions of the binary node. If cost vectors of the right operands are not kept as intermediate information, then they must be re-evaluated, violating the incrementalism condition.

Secondly, evaluating the cost expressions at code generation time is potentially expensive because Twig allows arbitrary expressions for the cost metrics of productions. Furthermore, dynamic programming requires that the cost of each potential reduction of a node be evaluated. Thus, several cost metric evaluations will be necessary at each node.

Finally, top-down tree automata are inherently less powerful than bottom-up tree automata. That is, the set of tree languages recognisable by top-down automata is a subset of those recognisable with bottom-up tree automata [PelegríLlopart88a, Section

2.2]. Patterns that enable potentially useful optimisations may not be recognisable by a top-down tree automaton.

For the above reasons, Twig is not a suitable base on which to develop an incremental recompilation facility. The top-down matcher not only requires considerable intermediate information, but is also less powerful than a bottom-up automaton. A code generator based on bottom-up pattern matching would address these deficiencies. Additionally, if more cost analysis is performed at preprocessing time and less cost analysis is carried out during recompilation, then faster recompilation, without sacrificing code quality, becomes possible.

In contrast to Twig, the code generator described by Hatcher in [Hatcher85] is based on bottom-up tree pattern matching. A Graham-Glanville style description, in which rules are annotated with a cost datum, specifies the target architecture. The resulting instruction selector uses three passes over the input trees to emit optimal object code.[3]

The first pass uses an adaptation of Hoffmann and O'Donnell's bottom-up pattern matcher [Hoffmann82] to assign matcher states to each node of the input tree. States encode the set of patterns and subpatterns matching at the node. The set of all patterns and subpatterns to be matched is finite, because there is a finite number of rules in the target architecture description; thus, the powerset of the set of the patterns and all subpatterns is finite. As a result, an integer state assigned to each node is able to uniquely denote the set of matching subpatterns. A node's state is determined by its label and the states of its children. The state transitions are encoded in tables generated at code generator generation time.

The second pass is a top-down traversal that selects the best sequence of patterns that were matched during the first pass. A goal symbol, taken from the set of non-terminals of the target architecture description, is inherited by each node. The goal of the root node is determined by the target architecture description. The goal of a non-root node is determined by the pattern selected at its parent. A table, called *matchsettab* in

[3] Code resulting from a least cost cover is assumed to be optimal in Hatcher's approach.

[Hatcher85], guides this choice; *matchsettab* is generated from cost information during code generator generation.

The third and final pass emits code based on the pattern sequences selected by the second pass. Hatcher observes that while code could be emitted during the second pass, the use of an extra pass allows the evaluation order of operands to be chosen in a way which minimises register usage.

In contrast to Twig, and other methods that use dynamic programming to select optimal object code, Hatcher shows that cost analysis can occur during code generator generation. Optimal code can then inferred, during code generation, via table lookup. However, Hatcher is unable to guarantee that the tables produced by his preprocessor will always choose a least cost sequence of patterns [Hatcher85, Section 3.5]. Furthermore, Henry observes in [Henry89] that the theoretical underpinnings of Hatcher's work are "shaky".

Cost analysis performed at code generator generation time, referred to as *static cost analysis* in [Henry89], requires that the cost metric of a pattern is constant. In contrast, performing cost analysis at code generation time (that is, when the code generator is executing) allows greater freedom in the choice of the cost metric. For example, an arbitrary expression may be specified as the cost metric in Twig. The costs shown earlier in Figure 5.6 indicate that, during code generation, the total cost of each potential reduction is formed by adding some constant, determined by the rule, to the sum of the costs of each reduction at positions corresponding to non-terminals of the pattern.

Hoffman and O'Donnell indicate that their bottom-up pattern matcher can be easily adapted to quickly rematch a subject tree after a perturbation [Hoffmann82]. Hatcher's variant of the Hoffman and O'Donnell matcher inherits this property and could thus form the basis of an incremental instruction selector. Particularly attractive is the static cost analysis that ensures the selection of high quality object code without requiring expensive analysis to be performed during recompilation. Furthermore, coalescing the second and third passes into a single pass would contribute to the speed of the algorithm, albeit at the expense of relinquishing the ability to reorder operand evaluation.

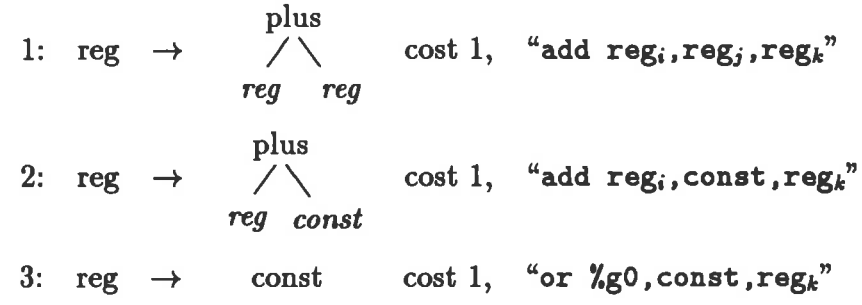


Figure 5.7 Rewrite rules from a BURS architecture description.

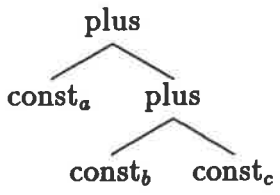
Assuming that the algorithm could be modified to operate on an abstract syntax tree, the state and goal symbols would need be stored at each node in order to satisfy the incrementalism condition. If the pattern set contains patterns with depth greater than one, then an indication of the ancestor that determined the goal is also required. Further decoration of nodes is needed to implement the source to object code mapping.

5.1.3 Code generation based on bottom-up rewrite systems.

For Twig and for Hatcher's code generators, the target architecture descriptions contain tree patterns that are associated with reduction symbols and are annotated with cost data. Reduction of matched subtrees drives code generation. A theory of *bottom-up rewrite systems* (BURS) is developed in [PelegríLlopart88a] and then applied to the construction of code generators. This approach characterises instruction selection as solving the problem of finding a least cost rewrite sequence that rewrites intermediate code trees into a predetermined goal symbol [PelegríLlopart88a, PelegríLlopart88b].

A target architecture is described as a list of rewrite rules. Each rewrite rule consists of a linear tree pattern and a reduction symbol. Associated with each rule is a constant cost datum and an action, usually consisting of fragments of object code to be emitted when the rule appears in a chosen rewrite sequence. Figure 5.7 illustrates rewrite rules annotated with instruction fragments and cost data.

The object code for an intermediate code tree can be obtained from any rewrite sequence that rewrites the tree into a predetermined goal symbol. For a given subject tree, there may be many suitable rewrite sequences. Figure 5.8 shows, with respect to



(a) a subject tree

1: rule 3 at 2.1
2: rule 3 at 2.2
3: rule 1 at 2
4: rule 3 at 1
5: rule 1 at ϵ

(b) sequence of cost 5

1: rule 3 at 2.1
2: rule 2 at 2
3: rule 3 at 1
4: rule 1 at ϵ

(c) sequence of cost 4

Figure 5.8 Two rewrite sequences of a subject tree.

the rewrite rules in Figure 5.7, (a) a subject tree, (b) a rewrite sequence of cost 5 that rewrites the tree into the symbol *reg*, and (c) a sequence of cost 4. From the first rewrite sequence, the following code is generated:

```
or   %g0,const_b,%10
or   %g0,const_c,%11
add  %10,%11,%12
or   %g0,const_a,%13
add  %12,%13,%14
```

The second rewrite sequence results in the following, shorter, piece of object code:

```
or   %g0,const_b,%10
add  %10,const_c,%12
or   %g0,const_a,%13
add  %12,%13,%14
```

In this example, the cost metric corresponds to the number of target machine instructions resulting from the application of a rule. Thus, the total cost of a rewrite sequence corresponds to the number of instructions resulting from that sequence. With such a cost metric, a least cost rewrite sequence of an intermediate code tree clearly results in the shortest object code sequence that implements the program fragment represented by the tree.

A code generator is able to emit locally optimal code, with respect to a given cost metric, by first finding a least cost rewrite sequence to the goal symbol and then emitting the code associated with the corresponding rewrite rules. Such a code generator may be characterised as solving the fixed goal C-REACHABILITY problem [PelegríLlopart88a, Chapter 6].

The fixed goal C-REACHABILITY problem is, given a rewrite system R, a tree T and a goal symbol γ, to determine whether there is a rewrite sequence in R that rewrites

T into γ and, if so, to find the one with minimum cost. The existence of a bottom-up tree automaton capable of solving the fixed goal C-REACHABILITY problem for finite bottom-up rewrite systems in time proportional to the size of the subject tree is established in [PelegríLlopart88a, Chapter 6]. From a solution to C-REACHABILITY, in the case that T is an intermediate code tree and R corresponds to a target architecture description, locally optimal code for T can be generated.

The automaton that solves C-REACHABILITY for finite BURS requires no cost analysis at code generation time. Instead, cost analysis, based on dynamic programming and occurring during automaton generation, encodes cost information into the states of the bottom-up tree automaton. Eliminating cost analysis during code generation contributes significantly to a faster compilation speed.

The BURS-based approach to instruction selection described in [PelegríLlopart88a, Chapter 6] provides a suitable theoretical foundation upon which an incremental instruction selector can be built. This approach is well suited because:

- the resulting incremental instruction selector is fast,
- the code generator state that must persist between recompilation is of a manageable size, and
- locally optimal object code is emitted.

The remainder of this chapter describes, in detail, an incremental instruction selection algorithm based on BURS. However, a number of the fundamentals of Pelegrí-Llopart's BURS theory require further exposition before proceeding further. Firstly, the extent of the information encoded into the states of the automaton is used to show that the algorithm can satisfy both the incrementalism condition and the goal of consistent recompilation. Secondly, the concept of the local rewrite sequence assigned to a node is used to determine the particular fragments of object code associated with the node. Note that the ensuing definitions are extracted from [PelegríLlopart88a].

Two requirements are placed on the states associated with nodes of the subject tree by the tree automaton:

(i) the collection of states contain enough information to characterise the minimum cost sequence that rewrites the subject tree to the goal symbol, and

(ii) the states can be computed in a bottom-up pass over the input tree.

Pelegrí-Llopart's δ-LR graphs satisfy both of these requirements. Furthermore, if the rewrite systems belongs to the finite BURS class of rewrite systems, then the number of possible δ-LR graphs is finite. Consequently, a single integer may be used to uniquely denote the δ-LR graph assigned to a node. Readers interested in further details of both the properties and construction of the δ-LR graphs, and of finite bottom-up rewrite systems, are referred to [PelegríLlopart88a].

The rewrite sequence can be inferred from the states assigned to the nodes of the subject tree and the fixed goal symbol. Each state may be mapped into a vector, ν, of rewrite rules indexed by the goal symbols of the rewrite system. If node n is assigned state σ by the automaton, then $\nu_\sigma(\gamma)$ denotes the rewrite rule that must be applied at n in a least cost rewrite sequence that rewrites the subtree rooted at n into γ. The pattern of this rule determines the goal symbols for the rewriting of the operands of n as part of this rewrite sequence. Thus, by using the fixed goal symbol as the goal of the root, a least cost rewrite sequence for the subject tree may be inferred from the states assigned by the tree automaton.

During incremental compilation, the object code associated with a node must be identifiable for each node of the subject tree, in order to maintain the source to object code mapping. In a tree rewrite based code generator, this is tantamount to associating the appropriate subsequence of the least cost rewrite sequence with each node of the subject tree. However, for an arbitrary rewrite system, there is no guarantee of a one-to-one mapping from nodes of the subject tree to subsequences of the rewrite sequence. For instance, if a rewrite sequence contains loops, there may be several subsequences associated with a particular node. Restricting the least cost rewrite sequences that may be

chosen by the automaton to *normal form* rewrite sequences [PelegríLlopart88a, Definition 5.1] overcomes this problem.

If τ is a rewrite sequence without loops, then τ is in *normal form* at ε if it is of the form $\tau_1 \ldots \tau_n \tau_0$, where τ_i is a subsequence of τ, and

(i) for all $i, 1 \leq i \leq n$, all rewrite rule applications in τ_i are at positions that are descendants of i,

(ii) there is no rewrite sequence τ', equivalent to τ, and of the form $\tau_1 \ldots \tau_n \tau_0'$, where τ_0' is a permutation of τ_0 starting with a rewrite rule application at a position $k.p$, for some k, $1 \leq k \leq n$, and position p, and

(iii) for all $i, 1 \leq i \leq n$, $(\tau_i)_{@i}$ is in normal form at i.[4]

For any non-looping rewrite sequence, τ, for a rewrite system R, there is a permutation of τ that is in normal form [PelegríLlopart88a, Proposition 5.1]. Thus, a least cost normal form rewrite sequence from R for a subject tree, T, is also a least cost rewrite sequence from R for T. A rewrite sequence for a tree in normal form assigns a *local rewrite sequence* [PelegríLlopart88a, Definition 5.1] to each node of the tree.

Let τ be a normal form rewrite sequence of the form $\tau_1 \ldots \tau_n \tau_0$ that is applicable to a tree T. The local rewrite sequence assigned by τ to a position p in T is defined by $F(T, \tau, p)$, where

(i) $F(T, \tau, \varepsilon)$ is τ_0, and

(ii) if p is of the form $i.q$ for some i, $1 \leq i \leq n$, and T is of the form $\omega(T_1, \ldots, T_n), \omega \in \Omega_n$, then $F(T, \tau, p)$ is $F(T, \tau_i, q)$.

Furthermore, the local rewrite assignment of τ and T is the function assigning the corresponding local rewrite sequence to each position in T.

Consider the rewrite sequence in Figure 5.8(c). The local rewrite assignment of the root node is (rule 1 at ε). The local rewrite assignment of the left operand of the root is (rule 3 at 1). However, the rightmost leaf has an empty local rewrite assignment. Code associated with a node by the incremental instruction selection process will depend

[4] If $\tau_i = (r_1 \text{ at } i.p_1).(r_2 \text{ at } i.p_2) \ldots (r_n \text{ at } i.p_n)$, where p_i is a path (see p. 74), then $(\tau_i)_{@i}$ is defined as $(\text{rule}_1 \text{ at } p_1).(\text{rule}_2 \text{ at } p_2) \ldots (\text{rule}_n \text{ at } p_n)$

on the local rewrite assignment. For instance, recalling the object code associated with the rewrite sequence on p. 103, the object code from the local rewrite assignment of the root is "`add %12,%13,%14`". The code associated with the left operand of the root is "`or %g0,const$_a$,%13`". No object code is associated with the rightmost leaf.

5.2 BURS-based incremental instruction selection

The code generated for an intermediate code tree by a BURS-based instruction selector is determined by a least cost normal form rewrite sequence of the tree. In the BURS-based incremental instruction selection algorithm proposed in this thesis, the code associated with a node of the abstract syntax tree is determined by the local rewrite sequence assigned by a least cost normal form rewrite sequence. Thus, BURS-based incremental instruction selection is equivalent to incrementally maintaining, for each node of the subject tree, the local rewrite sequence assigned by a least cost normal form rewrite sequence.

The least cost normal form rewrite sequence deduced for the subject tree, and hence the local rewrite sequence assigned to each node, is determined by:

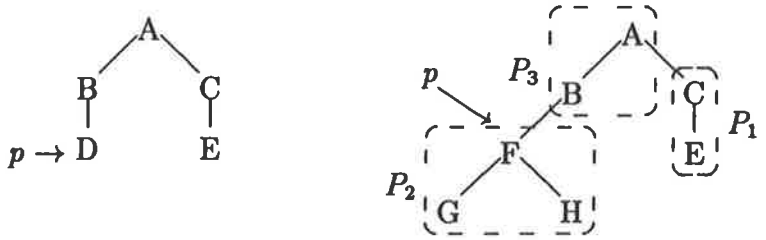
(i) the state assigned to each node by the matching automaton, and

(ii) the goal symbols inherited by nodes that appear in the rewrite sequence.

The incremental evaluation of each of these requires consideration. Furthermore, such consideration will lead to an understanding of the intermediate information that must persist between recompilations.

Consider a subject tree, T, whose nodes are fully annotated with the states assigned by a BURS matching automaton. Next, suppose that the subtree of T rooted at position p is replaced, resulting in the new subtree T'.[5] Relating the state assignment of T' to the state assignment of T will establish the subset of the nodes of T' for which recalculation of states is needed. This will thus establish the extent of incremental reassignment of states after subtree replacement. Analysis of the relationship between the state assignment of T

[5] Recall that subtree replacement is the only editing operation considered in the thesis (see Section 4.5, p. 82).



(a) initial tree (b) edited tree

Figure 5.9 Subtree replacement.

and that of T' is facilitated by a partitioning of the nodes of T', based on the replacement point, p.

Partition the set $dom(T')$ into the disjoint union of

$$P_1 = \{n \in dom(T'); p \perp n\}$$

$$P_2 = \{n \in dom(T'); p \ anc \ n\}$$

$$P_3 = \{n \in dom(T'); n \ anc \ p \text{ and } n \neq p\}$$

P_1 contains all the positions of T' that are independent of p. P_2 is that subtree of T' corresponding to the replacement subtree. Finally, P_3 is not a subtree of T'; instead, it is the collection of all the nodes of T' on the path from p to the root of T', excluding p but including ε. Observe that P_1 can be further partitioned into a set of subtrees of T' with the root of each subtree an immediate descendant of some member of P_3.

For example, Figure 5.9(a) shows an initial subject tree. Figure 5.9(b) shows the same tree after the subtree rooted at p (which is position 1.1) in the initial tree has been replaced by a new tree fragment. The partitioning of this new tree is indicated by dashed lines in Figure 5.9(b).

Denote by $\sigma(T, p)$ the state assigned by the pattern matching automaton at position p in T. Recall that one of the stated goals for these states is that they can be computed in a bottom-up pass of the subject tree (see p. 105). Thus, $\sigma(T, p)$ is fully determined by the label of $T@p$ and the assignment of the nodes belonging to the set $\{n \in dom(T);\ n \neq p \text{ and } p \ anc \ n\}$, that is the descendants of p. An immediate consequence of this is that if T and T' are isomorphic, then $\sigma(T, p) = \sigma(T', p), \forall p \in dom(T)$.

Consider a node $n \in P_1$. The subtree $T'@n$ of T' is isomorphic with the subtree $T@n$ of T. Thus, $\sigma(T,n) = \sigma(T',n)$ for any $n \in P_1$.

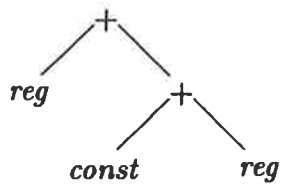
Next, consider the partition P_2. This is the subtree of T' with no corresponding subtree in T. For $n \in P_2, \sigma(T',n)$ is independent of $\sigma(T,m), \forall m \in dom(T)$.

Finally, suppose that node $n \in P_3$ and $n \neq \varepsilon$. Furthermore, suppose that $\sigma(T',n) = \sigma(T,n)$. Let n' be the parent of n, $k = \textit{arity-of}(T@n')$, and let $j \in \{1,\ldots,k\}$ be the integer such that $n = n'.j$. Now $\sigma(T',n')$ is fully determined by $\textit{label-of}(T@n')$ and $\sigma(T'@n'.i)$, for $i \in \{1,\ldots,k\}$. For $i = 1..k$, $i \neq j$, we have $p.i \in P_1$; thus, $\sigma(T',p.i) = \sigma(T,p.i)$. Furthermore, $\sigma(T',n'.j) = \sigma(T,n'.j)$, because of the supposition that $\sigma(T',n) = \sigma(T,n)$. Thus, because the label of $T'@n'$ is certainly the same as the label of $T@n'$, it follows that $\sigma(T',n') = \sigma(T,n')$. An immediate consequence is that if $n \in P_3$ and $\sigma(T',n) = \sigma(T,n)$, then $\sigma(T',q) = \sigma(T,q)$, for all $q \in P_3$ such that $q \ anc \ n$.

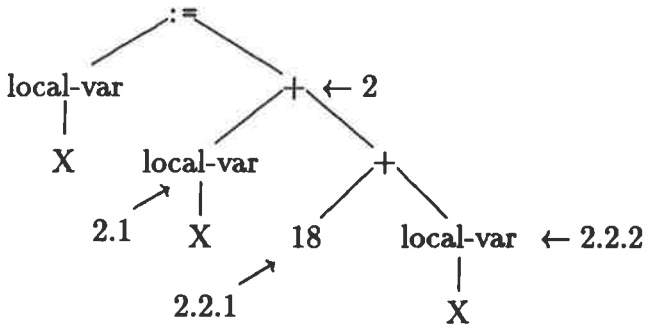
Thus, when a subtree of the subject tree is replaced, the extent of recalculation and reassignment of BURS pattern matcher states is determined by the partitioning of the nodes of the subject tree into the sets P_1, P_2 and P_3. States of the new nodes, of course, require calculation. States of nodes independent of the replacement subtree are unchanged. Nodes between the replacement point and the root of the tree need to be considered sequentially from the replacement point towards the root, calculating and reassigning states until the newly calculated state of a node is the same as the previous value.

In the example of Figure 5.9, the nodes labelled with C and E belong to P_1 and so the states are unaffected by the subtree replacement. The newly created nodes F, G and H belong to P_2; thus, they require state calculation and assignment. This leaves nodes A and B, both of which belong to P_3. The new state of node B requires calculation and assignment. However, the state at A requires calculation only if the new state of B differs from the old state.

The extent of incremental state reassignment therefore consists of the replacement subtree and a subset of the nodes between the point of replacement and the root of the subject tree. This is reflected, later in this chapter, by the algorithm of Figure 5.19. To



(a) a pattern



(b) a matched subject tree

Position in subject tree	Position in pattern tree	Goal at this position in the subject tree
2	ε	*undefined*
2.1	1	*reg*
2.2.1	2.1	*const*
2.2.2	2.2	*reg*

(c) match details

Figure 5.10 Inferring goal symbols for subnodes.

satisfy the incrementalism condition, states assigned by the automaton to each node must persist between successive incremental recompilations.

The goal symbol of a node, other than the root, that appears in the least cost normal form rewrite sequence of a subject tree is determined by a rule application at one of its ancestors. Suppose the pattern of a rule contains the non-terminal symbol N at position n and that the rule appears in the least cost normal form rewrite sequence for the subject tree T at tree position p. Then, the goal symbol of the node at position $p.n$ in T is N.

For example, suppose that a rule, r, contains the pattern shown in Figure 5.10(a) and that the rewrite application (r at 2) occurs in a normal form rewrite sequence for the subject tree shown in Figure 5.10(b); that is, the pattern has matched at position 2 of the subject tree. The table in Figure 5.10(c) summarises this match. For instance, the second entry indicates that position 2.1 in the subject tree corresponds to position 1 of the matched pattern. Thus, the goal symbol *reg* is assigned to node 2.1 of the subject tree.

The normal rewrite sequence for the subject tree must then assign local rewrite sequences to the nodes at 2.1, 2.2.1 and 2.2.2 of the subject tree that rewrite these subtrees to the indicated goals.

In general, suppose that $(g \rightarrow P \text{ at } n)$ occurs in a normal form rewrite sequence for a subject tree T. Define $var(P)$, where P is a pattern, by

$$var(P) = \{m \in dom(P); P@m \text{ is a non-terminal}\}$$

Then, for each $q \in var(P)$, $P@q$ is the goal symbol of $n.q$ in the subject tree, T, and n is the *goal determining node* for $T@n.q$. For nodes $q \in dom(P) \setminus var(P)$, the node $T@n.q$ has neither a goal symbol nor a goal determining node. If the maximum height of a pattern in the rewrite system is k, then the goal determining node of $T@n$, where $n = n_1.n_2 \ldots n_m$, n_i an integer, is one of $\{T@n_1 \ldots n_{m-1}, T@n_1 \ldots n_{m-2}, \ldots T@n_1 \ldots n_{m-k}\}$.

To incrementally determine the local rewrite assignment to a node n, the state and the goal symbol of the node must be incrementally inferred. However, the goal symbol of n is determined by the local rewrite assignment to the goal determining node, d. Thus, if either the goal or the state assigned to d changes, as the result of an edit to the subject tree, the goal of n must be reassigned. This incremental processing is complicated by both the range of possible goal determining nodes for n and the possibility that n may not have a goal at all. Restricting the rewrite system so that rules are in *normal form* [Balachandran90, Section 3.1] simplifies this incremental processing.

A rule is in *normal form* if it is either a chain rule of the form $X \rightarrow Y$, or a pattern rule of the form $X \rightarrow \omega(X_1, \ldots, X_n)$, where $X, Y, X_1, \ldots X_n$ are non-terminals and ω an operator of arity n. Note that if ω is nullary, the pattern rule becomes $X \rightarrow \omega$.

Observe that, by the addition of extra non-terminal symbols, any rewrite rule can be replaced by an equivalent set of normal form rewrite rules. For example, the non normal form rewrite rule $goal \rightarrow P$, where P is the pattern of Figure 5.10(a), can be replaced by the pair of normal form rewrite rules in Figure 5.11, introducing the new non-terminal *plus*.

Figure 5.11 Normalisation of a rewrite rule.

If each rule of the rewrite system is in normal form, then the local rewrite assignment to node n by a normal form rewrite sequence of a subject tree T is of the form $(o \text{ at } n),(c_1 \text{ at } n),\ldots,(c_k \text{ at } n)$, where, for $k \geq 0$, c_i is the chain rule $X_i \rightarrow Y_i$ and o is the pattern rule $X_o \rightarrow \omega(Z_1,\ldots,Z_m)$, where $T@n$ is labelled by ω, and $Y_k = X_0$. That is, a pattern rule application followed by a possibly empty sequence of chain rules. X_k must be the goal symbol of the node n. In turn, the pattern rule determines the goal symbols for the m children of $T@n$, that is $Z_1,\ldots,Z_m$.

If each rule application of a normal form rewrite sequence of a subject tree T is in normal form, then each node is assigned a non-empty local rewrite sequence. For each node n of T, the goal determining node of n is the parent of n, with the exception of the root, for which the goal symbol is the predetermined fixed goal. In turn, the local rewrite sequence of n, determined by the state assignment, σ, and the inherited goal symbol, g, of n, is given by $\nu'_\sigma(g,n)$, where[6]

$$\nu'_\sigma(g,n) = \begin{cases} (\text{rule } \nu_\sigma(g) \text{ at } n) & \text{if } \nu_\sigma(g) \text{ is the pattern rule } g \rightarrow \omega(X_1,\ldots X_m) \\ \nu'_\sigma(X,n).(\text{rule } \nu_\sigma(g) \text{ at } n) & \text{if } \nu_\sigma(g) \text{ is the chain rule } g \rightarrow X \end{cases}$$

In turn, if $(g \rightarrow \omega(X_1,\ldots,X_m) \text{ at } n)$ is the first rule application of the local rewrite assignment of n then, for $j = 1,\ldots m$, the goal symbol assigned to $n.j$ is X_j.

The relationship between the goal symbols of nodes of a subject tree before an edit operation, T, must now be related to the goal symbols of the new subject tree T' formed by replacing the subtree of T at position p by γ.

Recall the partitioning of the tree domain of T' into P_1, P_2 and P_3, based on the position p, where $T' = T_{p \leftarrow \gamma}$ (see p. 108). Now define the subset P'_3 of P_3 by

$$P'_3 = \{n \in P_3; \sigma(T,n) \neq \sigma(T',n)\}$$

[6] Recall the definition of ν_σ on p. 105.

which is the subset of P_3 for which new states are calculated. If P_3' is empty, then the state assignments for nodes from P_1 and P_3 of T' are identical to the corresponding nodes in T. Consequently, the goal symbols and the local rewrite assignments of each of these nodes of T' are the same as for T. Define the node t by

$$t = \begin{cases} p & \text{if } P_3' = \emptyset \\ t' \in P_3' & \text{such that } q \in P_3' \text{ implies } t' \ anc \ q, \text{ if } P_3' \neq \emptyset \end{cases}$$

In the first case, the state assignment for each node of P_1 and P_3 of T' is unchanged from the state assignment of the corresponding node of T. Thus, for these nodes, the local rewrite assignment is unchanged from T. The goal symbol of $T'@p$ is inherited from its parent and thus is equal to the goal symbol of $T@p$. The local rewrite sequence of $T'@p$, and those for all its descendants, can then be evaluated.

In the second case, t is the node of T' such that for any node t' of T' such that the state assignment differs from the state of the corresponding node of T, then $t \ anc \ t'$; that is, t is the node closest to the root of T' such that the state assignment in T' is different to the state assignment in T. The goal symbol of $T'@t$ is inherited from its parent and so is equal to the goal symbol of $T@t$. However, $T'@t$ having a different state assignment than $T@t$ will have a different local rewrite sequence. The local rewrite sequence of $T@t$ and all its descendants can then be evaluated.

Thus, after an edit operation, the states of the nodes in the modified subtree are incrementally updated and the node t inferred during the update. Then local rewrite sequences and goal symbols for the subtree rooted at $T'@t$ are evaluated and assigned. This is reflected in the algorithm of Figure 5.20 on p. 129. The goal symbol for each node, along with the state assignment, is required intermediate information for the incremental BURS instruction selector.

In fact, it is possible that some nodes belonging to P_1, but in the subtree rooted by $T'@t$, may have a goal and local rewrite assignment equal to that of the corresponding nodes of T. All of the descendants of any such node also have a goal and state assignment equal to the corresponding nodes in T, because the goal assignment to the children of a node n depend only on the label, goal and state assignment of n. If, during evaluation of

Label	Description	Arity	Left child	Right child	Value	
ASSIGN	assignment	2	destination	source		
PLUS	addition	2	operand	operand		
MUL	multiplication	2	operand	operand		
CBRANCH	conditional branch	2	test	label		
INDIR	memory fetch	1	address			if not left child of assign
INDIR	memory store	1	address			if left child of assign
SCONV	scalar conversion	1	operand			
CONST	integer constant	0			constant	
NAME	global variable	0			name	
REG	dedicated register	0			register	
LABEL	label	0			number	

Figure 5.12 Some operators from PCC-IR [Henry84, figure 2.1].

the new goal symbol for a node from P_2 such that $T'@t$ is an ancestor, the goal symbol is equal to the corresponding node in T, then the local rewrite assignment and goal symbol of that node, and all its descendants, are equal to those of the corresponding nodes of T. This possible optimisation, while contributing to the speed of incremental recompilation, is not exploited in the algorithm of Figure 5.20.

5.3 Transforming abstract syntax trees

The BURS instruction selection algorithm, as originally presented by Pelegrí-Llopart in [PelegríLlopart88a] and Proebsting's variation in [Proebsting92], assumes that the input to the code generator is a sequence of intermediate code trees. Pelegrí-Llopart made use of the front end provided by the UW-CODEGEN system [Henry84] to provide this input to his prototype code generator [PelegríLlopart88a, Chapter 8]. Consequently, it uses the PCC-IR intermediate representation. The low level nature of the operators of PCC-IR is apparent from the examples of PCC-IR operators shown in Figure 5.12. The BURS-based code generator used in the `lcc` compiler uses a similarly low level intermediate representation as input [Fraser91a, Fraser91c, Fraser92, Proebsting92].

The *shape* of the intermediate code tree is the tree domain and the label assignment; it does not include any additional information that may annotate nodes. For example, if a node is labelled with REG, then the label and the position of the node in the tree are considered as part of the shape. However, the index of the register allocated to

that node is not part of the shape. In each of the implementations of BURS-based code generators, the assignment of states by the pattern matcher depends only on the shape of the intermediate code trees.

The *shape* of the emitted object code is the sequence of machine op-code that is selected to implement the intermediate code tree. Details of the operands are not considered to be part of the shape. In each of the BURS-based code generators, the shape of the emitted object code is uniquely determined by the state assignment of the intermediate code tree.

Semantic information is exploited during the generation of the intermediate code trees to distinguish between the alternative interpretations of fragments of source code. For example, an identifier occurrence could denote a global variable, a parameter allocated to a register, or some other entity, depending on the source language. In the first instance, the corresponding node of the PCC-IR expression tree would be labelled with the NAME operator in Figure 5.12, and in the second with the REG operator.

The pattern matcher cannot directly exploit semantic information when assigning states. Instead, all semantic information that can affect the shape of the emitted code is encoded in the shape of the intermediate code tree. This notion of *syntax as semantics* is elaborated on in [Henry84].

Incrementally maintaining a separate intermediate representation could adversely affect the speed of recompilation and thus the response time of the integrated programming environment. It would certainly affect the memory demands of the environment. However, the shape of an abstract syntax tree does not by itself contain sufficient information for a BURS pattern matcher to encode states that distinguish between the alternative object code shapes for a particular fragment of source code. This can be illustrated by two simple examples.

Firstly, consider the loading of an integer literal into a register. In the SPARC architecture [Sun87], the literal may be loaded into a register with either a single instruction, if the constant will fit into a 13 bit field of the instruction, or with two instructions if the constant is too large for a 13 bit field. Object code fragments for these two cases are

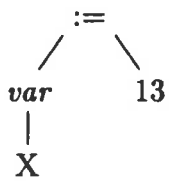
```
or %g0,value,%10
```

(a) value less than 2^{13}

```
sethi %hi(value),%11
or %11,%lo(value),%10
```

(b) value greater than or equal to 2^{13}

Figure 5.13 Loading an integer constant into the SPARC register %10.



(a) a tree fragment

```
add %g0,13,%10
st %10,[%fp-12]
```

(b) when X is a local variable

```
add %g0,13,%o0
```

(c) when X is an out parameter

Figure 5.14 Identifier resolution in an abstract syntax tree.

shown in Figure 5.13(a) and (b), respectively. In order to claim locally optimal instruction selection, the code generator must choose the two instruction form only if the literal value is greater than or equal to 2^{13}. Consequently, the bottom-up pattern matcher in the BURS instruction selector must assign a state to the node representing the literal, where this state reflects the range to which the value belongs. Because the BURS pattern matcher cannot exploit semantic information, the range must be coded into the intermediate representation. On the SPARC, two operators are needed to distinguish the ranges of interest for integer literals in the intermediate representation.

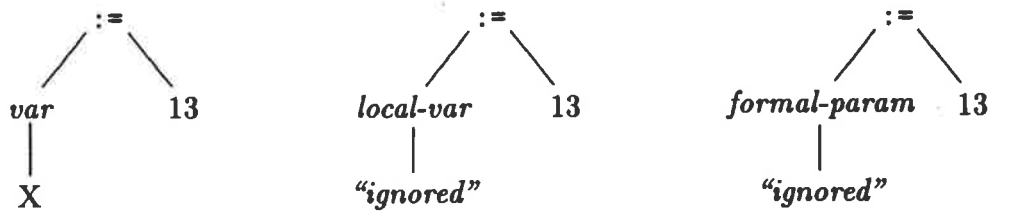
As a second illustration, consider the many meanings of an identifier occurrence in an abstract syntax tree. An identifier on the left hand side of an assignment statement can denote a local variable, a global variable, a parameter or some other entity, depending on the source language. Figure 5.14 shows an assignment statement and object code appropriate to two such alternatives for the statement. If X in Figure 5.14(a) denotes a local variable, then the literal value must first be loaded into a register and then the value stored into the local activation record, as shown in Figure 5.14(b). If X is an out parameter that is allocated to the SPARC register %o0, then the literal is loaded directly into %o0, as shown in Figure 5.14(c). The correct object code for the tree fragment of Figure 5.14(a) clearly depends on semantic information. In order to generate correct

object code, the state assigned by the BURS pattern matcher to the node labelled ":=" must distinguish between the possible meanings of X. Given that the pattern matcher can examine only the labels of nodes in the abstract syntax tree, it is clear that BURS-based instruction selection cannot directly operate from an abstract syntax tree, since it lacks the ability to access semantic information.

In an integrated environment that uses abstract syntax trees as the canonical program representation, the incremental code generator has access only to the abstract syntax tree and any semantic information that has been derived from this tree. Furthermore, the inputs to the code generator are notifications of edit operations on the abstract syntax tree. As illustrated by the preceding section, insufficient information is encoded into the shape of an abstract syntax tree for BURS-based instruction selection to operate. In particular, semantic information that can affect the shape of the emitted code must be reflected in the shape of the tree analysed by the BURS pattern matcher. In a non-incremental BURS-based code generator, the intermediate representation encapsulates the necessary semantic information. Likewise, the shape of the input to a BURS-based incremental code generator must also encapsulate any semantic information that can affect the shape of the emitted code.

A separate data structure to implement an intermediate representation would consume considerable extra memory, and its creation and incremental manipulation would adversely affect the speed of incremental recompilation. However, an intermediate representation can be defined that is both suitable for BURS pattern matching and sufficiently close to the abstract syntax tree that the expensive maintenance of a separate structure is not required.

The notion of sufficiently close to the abstract syntax tree, mentioned above, requires more careful examination. Suppose that T' is the tree structured intermediate representation of the abstract syntax tree T. If $dom(T) = dom(T')$, then T and T' are isomorphic up to a relabelling of nodes, and T' can be implemented by providing a field for a new label in each node of T.



(a) tree fragment (b) relabelling when local variable (c) relabelling when formal parameter

Figure 5.15 Semantic relabelling of an abstract syntax tree.

For example, Figure 5.15 shows how the tree fragment in Figure 5.14(a) can be relabelled so that the shape of the tree indicates the nature of the identifier occurrence. Observe that the unary node that is labelled by the *var* operator in the abstract syntax tree fragment of Figure 5.15(a) corresponds in each of Figure 5.15(b) and (c) to a node that is labelled with an operator that provides sufficient semantic information about the variable X to determine the shape of the generated code. A distinguished operator *"ignored"* indicates nodes in the intermediate representation that have no effect on the pattern matcher state assignment.

Constructing the intermediate representation is therefore reduced to a problem of incrementally maintaining the relabelling operator for each node in the abstract syntax tree. This relabelling can be precisely described as a function that maps an abstract syntax tree, T, over the canonical abstract syntax, $\mathcal{L}$, to an abstract syntax tree, T', over a new abstract syntax, $\mathcal{L}'$, that is an extension of $\mathcal{L}$. Appropriate constraints on both $\mathcal{L}'$ and the nature of the transformation ensure that T and T' are structurally isomorphic, and that the transformation has an efficient incremental implementation.

The relabelling transformation requires access to semantic information. Such information is inferred from the program text during incremental semantic analysis. The relabelling transformation can be precisely described only if the means of accessing the semantic information through the abstract syntax tree is determined. This thesis describes a class of transformations that assume that the canonical abstract syntax tree is decorated with semantic information derived from an attribute grammar. The transformation is then specified as a mapping of the abstract syntax tree into a new abstract

syntax tree over a modified abstract syntax that accesses attributes associated with nodes of the source tree.

Let $\mathcal{L} = (\Sigma, \mathcal{G}, \xi)$ be an abstract syntax. Suppose that the set S of sorts of $\mathcal{L}$ includes the symbol *Boolean* and that the generator set of the sort *Boolean* is $\{true, false\}$. Section 4.4 showed how abstract syntax trees can be attributed and the evaluation function, *eval*, can be defined (see p. 80). Suppose that an attribution has been defined for $\Gamma(\mathcal{L})$. Then, for each operator ω with type $s \in \mathcal{S}$,[7] define the set, $\mathcal{P}_\omega$, of predicates for ω by

$$\mathcal{P}_\omega = \{a \in (Syn_\omega \cup Inh_\omega \cup Local_\omega);\ type\text{-}of(a) = Boolean\}$$

Let Ω' be the disjoint union of the sets $\Omega'_n, n = 0, 1, 2, \ldots$, such that Ω' is disjoint from Ω, the set of operator symbols of the abstract syntax, $\mathcal{L}$. Furthermore, for each $\Omega_n \neq \emptyset$, the set Ω'_n is the disjoint union of the finite sets of symbols Cut_n, $Inner_n$ and $\{\varphi_n\}$. Cut_n contains the extending operators that label subtrees that may be effectively excised from the intermediate tree. $Inner_n$ contains the extending operators that label inner nodes of the intermediate tree, that is the non-leaf nodes of the intermediate tree. φ_n is a distinguished operator used to indicate nodes of the intermediate tree that have no effect on code generation (that is, the *"ignored"* nodes of Figure 5.15). For each operator $\omega \in \Omega_n$ of the abstract syntax, define a set of pairs

$$pairs_\omega = \{(a, \omega'); a \in \mathcal{P}_\omega, \omega' \in \Omega'_n\}.$$

That is, each operator ω in the abstract syntax is associated with a, possibly empty, set of tuples that associate a predicate with a relabelling operator. The attribute will be used to guard a mapping of nodes labelled with ω into nodes labelled with a relabelling operator. Furthermore, the relabelling operator must have the same arity as ω. Constraining the arity of the relabelling operator ensures that an abstract syntax tree will be isomorphic, up to relabelling, with its intermediate representation. That is, if the relabelling of a

[7] Recall the assumption on p. 74 that allows the type of an operator to be precisely defined.

canonical abstract syntax tree T results in the intermediate tree T', then $dom(T) = dom(T')$.

Let α be the Ω-structure of the abstract syntax $\mathcal{L}$ and define an Ω-structure α' over $\mathcal{S}$ mapping $\Omega_n \bigcup \Omega'_n$ into $\mathcal{R}^{(n+1)}(\mathcal{S})$ by

$$\omega\alpha' = \begin{cases} \omega\alpha & \text{if } \omega \in \Omega_n \\ \bigcup_{\nu \in \delta(\omega)} \omega'\alpha & \text{if } \omega \in \Omega'_n \\ \mathcal{R}^{(k+1)}(\mathcal{S}) & \text{if } \omega = \varphi_k \end{cases}$$

$$\text{where } \delta(\omega) = \{\nu \in \Omega_k; \exists a \in \mathcal{P}_\nu, (a, \nu) \in pairs_\nu\}$$

In the first case, the signatures of operators from Ω are unchanged in the Ω-structure α'. The second case constructs the signature for a relabelling operator $\omega \in \Omega'_n$ as the union of the signatures of each operator potentially relabelled by ω'. The final case specifies that the signature of φ_k includes all the $(k+1)$-tuples over $\mathcal{S}$.

Then, $\Sigma' = (\mathcal{S}, \alpha')$ is an Ω-structure of $\mathcal{S}$, and $\mathcal{L}' = (\Sigma', \mathcal{G}, \xi)$ is an abstract syntax called the *semantic extension* of the abstract syntax $\mathcal{L} = (\Sigma, \mathcal{G}, \xi)$. For an abstract syntax tree $T \in \Gamma(L)$, a semantic extension $\mathcal{L}'$ of $\mathcal{L}$, and $p \in dom(T)$ a position in T, define the transformation function from $\mathcal{L}$ into $\mathcal{L}'$ by

$$transform(T,p) = \begin{cases} T@p & \text{if } T@p \in \mathcal{G}_s \text{ for some generator set } \mathcal{G}_s \text{ of } \mathcal{L} \\ \omega'(transform(T,p.1), \ldots, transform(T,p.k)) & \\ \qquad & \text{if } active(T,p) = \{(a,\omega')\},\ \omega' \in Inner_k \\ \omega'(cut(T,p.1), \ldots, cut(T,p.k)) & \\ \qquad & \text{if } active(T,p) = \{(a,\omega')\},\ \omega' \in Cut_k \\ \omega(transform(T,p.1), \ldots, transform(T,p.k)) & \\ \qquad & \text{if } active(T,p) = \phi,\ \omega = label\text{-}of(T,p),\ \omega \in \Omega_k \\ undefined & \text{otherwise} \end{cases}$$

$$\text{where } active(T,p) = \{\omega' \in \Omega';\ (a,\omega') \in pairs_\omega, \omega = label\text{-}of(T,p), eval(T,p,a) = true\}$$

$$cut(T,p) = \begin{cases} \varphi_0 & \text{if } T@p \in \mathcal{G}_s \\ \varphi_0 & \text{if } T@p = \omega, \omega \in \Omega_0 \\ \varphi_k(cut(T,p.1), \ldots, cut(T,p.k)) & \text{if } T@p = \omega(t_1, \ldots, t_k), \omega \in \Omega_k \end{cases}$$

The *transform* function constructs an abstract syntax tree that is isomorphic up to relabelling with its parameter T. Generator instances are unaffected by *transform*, as is apparent from the first case in the definition of *transform*. The transformation of an operator application at a position p in T is determined from the evaluation of the guard attributes applicable at $T@p$. The set $active(T,p)$ contains the relabelling operators for which the guard attribute instances at p evaluate to *true*.[8] If $active(T,p)$ contains the single inner operator ω', then the transformation of the subtree $T@p$ is the application of ω' to the transformations of the subtrees of $T@p$. If $active(T,p)$ contains the single cutting operator ω', then the transformation of the subtree $T@p$ is the abstract syntax tree that is isomorphic up to relabelling with $T@p$, such that each position is labelled with φ_k, for the appropriate value of k. If $active(T,p)$ is the empty set and $T@p$ is labelled with ω, then the transformation of $T@p$ is the application of ω to the transformations of the subtrees of $T@p$. Finally, the result is undefined when $active(T,p)$ contains more than one member.

The function *transform-ast*:$\mathcal{L} \rightarrow \mathcal{L}'$ that maps an abstract syntax tree to its intermediate representation is then defined by

$$transform\text{-}ast(T) = transform(T, \varepsilon)$$

A suitable semantic extension and transformation function to partition integer constants in order to select the appropriate SPARC instruction sequence from Figure 5.13 can now be defined. Suppose that $\mathcal{L}$ contains the operator *constant* with signature

$$constant: \text{INT} \rightarrow \text{EXPR}$$

Define the boolean-valued attributes *is-small* and *is-large*, both of which belong to the set of local variables for the operator *constant*; these attributes indicate to which range

[8] Recall that guard attributes are defined to have type *Boolean*, whose generator set is {*true*, *false*}.

the integer belongs. Define the operators *small-constant* and *large-constant* as members of Cut_1. Then, the semantic transformation is determined by defining

$$pairs_{constant} = \{(is\text{-}small,\ small\text{-}constant), (is\text{-}large,\ large\text{-}constant)\}$$

A transformation function for the identifier resolution example in Figure 5.14 and Figure 5.15 is obtained by making the following assumptions:

$$\{local\text{-}variable,\ formal\text{-}parameter\} \subseteq Cut_0$$

$$\{is\text{-}local,\ is\text{-}formal\text{-}parameter\} \subseteq \mathcal{P}_{var}$$

$$\{(is\text{-}formal\text{-}parameter, formal\text{-}parameter),$$

$$(is\text{-}local, local\text{-}variable)\} \subseteq pairs_{var}$$

These new operators from Ω' encode the semantic information in a manner that may be utilised by a BURS pattern matcher. A number of benefits of such a transformation can be identified:

- subtrees of the abstract syntax tree not associated with any object code are effectively excised by mapping to φ_k, for various k,
- the transformed tree shares storage with the canonical abstract syntax tree, and
- the transformation may be implemented incrementally.

For a given abstract syntax tree, there will be subtrees that are not mapped to object code. For example, an uninitialised declaration will not usually require object code to be emitted. In a traditional compiler, no intermediate code trees are produced for such declarations. Relabelling the nodes of subtrees corresponding to such declarations with φ_k indicates to the incremental instruction selector that the subtree can be skipped.

If T is an abstract syntax tree and $T' = semantic\text{-}transform(T)$, then the semantic transformation function is defined to ensure that T' is structurally isomorphic to T. That is, $dom(T') = dom(T)$. Thus, the representation of T' can be implemented by providing storage for the relabelling operator in each node of T.

After a subtree replacement operation, the new transformed subtree can be inferred in a single top-down traversal of the new tree fragment. Figure 5.16 shows the incremental

```
1: Prune-subtree(T, p) :
2:   let k = arity-of(T@p)
3:   T@p.relabel ← φ_k
4:   for i = 1, ..., k do
5:     Prune-subtree(T, p.k)
6:   od
```

(a) Prune-subtree.

```
1: Transform-subtree(T, p) :
2:   let k = arity-of(T@p)
3:   if active(T, p) = ∅ then
4:     T@p.relabel ← label-of(T, p)
5:     for i = 1, ..., k do
6:       Transform-subtree(T, p.i)
7:     od
8:   elsif active(T, p) = {ω}, ω ∈ Ω_k then
9:     T@p.relabel ← ω
10:    if ω ∈ Cut_k then
11:      for i = 1, ..., k do
12:        Prune-subtree(T, p.i)
13:      od
14:    elsif ω ∈ Inner_k then
15:      for i = 1, ..., k do
16:        Transform-subtree(T, p.i)
17:      od
18:    elsif ω = φ_k then
19:      Prune-subtree(T, p)
20:    fi
21:  fi
```

(b) Transform-subtree.

Figure 5.16 Semantic transformation after an edit.

semantic transformation algorithm. If, before the edit operation, $T@p.relabel = \varphi_k$, for some k, then the edited abstract syntax tree is consistently relabelled by calling *Prune-subtree*(T, p); otherwise, *Transform-subtree*(T, p) is called.

Prune-subtree(T, p) in Figure 5.16(a) sets the relabelling field of $T@p$ and all its descendants to φ_k, for various k.

Transform-subtree(T, p) in Figure 5.16(b) traverses the subtree of T rooted at p in order to determine the relabelling operators of each node. If the active set of the root of the subtree (Figure 5.16(b), line 3) is empty, then the root's new label is the same as

the original label (line 4) and the remainder of the subtree is then visited (lines 5 and 6). If the active set contains a single operator ω (line 8), then the node is relabelled to this operator (line 9). If ω is a cutting operator (line 10), then the remainder of the subtree is pruned from the semantically transformed abstract syntax tree (lines 11 and 12). If ω is an inner operator (line 14), then the remainder of the subtree is recursively transformed (lines 15 and 16). Finally, if the ω is the cutting operator φ_k, then the entire subtree is pruned (line 19).

This transformation method accesses semantic information as attribute values decorating the nodes of an abstract syntax tree. The prototype implementation described in Chapter 7 uses an incremental visit sequence analyser [Reps89a, Chapter 12] to perform incremental semantic analysis. Chapter 7 describes the implementation of the abstract syntax tree transformation (see p. 187).

The transformation depends on semantic information to infer the *active* set for each node that is not excised from the tree. As presented, *active* is defined in terms of pure attribute evaluation. Alternative definitions of *active* would allow the above method to be used with different kinds of incremental semantic analyser. However, it is necessary that *active* can be efficiently realized. Furthermore, evaluation of *active* should be dependent only on the node under consideration. For example, Hedin's door attribute grammars [Hedin92], or Horwitz's symbiosis of relations and attributes [Horwitz85, Horwitz86], would readily lend themselves to this technique.

Transforming the abstract syntax tree in this fashion avoids the necessity of maintaining a separate intermediate representation. However, in order to generate code from transformed abstract syntax trees, the rewrite sequence becomes larger and more complex. In particular, there are now non-terminals and associated rewrite rules that effectively model translation to an intermediate representation.

The class of transformations described in this section has proved adequate to implement a BURS-based incremental instruction selector for a small expression-oriented language (see Appendix C). However, further experimentation is required to gauge the applicability of this method to more realistic languages.

5.4 The incremental instruction selection algorithm

The analysis in Section 5.2 demonstrates that it is possible to base an incremental instruction selector on bottom-up rewrite systems. Section 5.3 shows the need for the generation of some form of intermediate code, derived from the canonical abstract syntax tree and semantic information, prior to bottom-up pattern matching. An intermediate representation, suitable for use in an integrated programming environment, along with an efficient transformation algorithm (Figure 5.16) is described in Section 5.3.

The BURS-based incremental instruction selection algorithm can now be described. The algorithm is first presented without the complications of managing the object code file or branch updating. Next, the source to object code mapping is considered, and the algorithm is extended to incrementally manage the object code file and the mapping from source code to object code. Finally, the problem of branch updating is considered.

5.4.1 Updating after statement edits

The basic incremental instruction selection algorithm updates the code file in response to an edit. Recall that an edit operation is considered to be the replacement of a subtree of the abstract syntax tree. Subtree replacement is modelled as the substitution of a subtree at a given position of the initial tree (see Section 4.5). It is assumed that the replacement results in a well formed abstract syntax tree (see p. 73) and that incremental semantic analysis has been performed.

A part of the abstract syntax tree will require recompilation after an edit. The part requiring recompilation is the extent of recompilation (see Section 2.1, p. 9). In this approach to incremental recompilation, this extent is always a subtree of the abstract syntax tree. Thus, the extent is uniquely determined by the position of the root of the subtree. In the ensuing discussion, this subtree is referred to as the *extent*, and the root of the extent is referred to as the *extent determining position.*

The size of an edit operation is defined to be the number of nodes in the subtree rooted by the extent determining position of the modified abstract syntax tree.

```
1: Recompile (T, p, goal) :
2:   if is-pruned (T@p) then
3:     Prune-subtree (T, p)
4:   else
5:     Transform-subtree (T, p)
6:   fi
7:   let k = arity-of(T@p)
8:   if T@p.relabel ≠ φ_k then
9:     Match-subtree (T@p)
10:    top ← Expand (T, p)
11:    if top = p then
12:      Reduce(T, p, goal)
13:    else
14:      Reduce(T, top, T@top.goal)
15:    fi
16:  fi
```

Figure 5.17 Recompilation after subtree replacement.

After an edit operation, the *Recompile* procedure, shown in Figure 5.17, is invoked. It requires three parameters:

- T, the abstract syntax tree containing the program representation,
- p, the position in the abstract syntax tree at which the replacement occurred, and
- *goal*, the goal symbol inherited from the parent of $T@p$.

It is possible that the subtree replacement occurred in a part of the tree previously pruned during subtree transformation. In this instance, the new subtree is also pruned on line 3 of Figure 5.17 by calling *Prune-subtree*, defined in Figure 5.16(a). The predicate *is-pruned*, called on line 2 of Figure 5.17, examines the relabelling operator of the parent node: if this is either a cutting operator (that is, a member of the set Cut_k defined on p. 119) or φ_k, where k is the arity of the parent, then the replacement has occurred in a pruned subtree of the abstract syntax tree and should thus be pruned.

If the new subtree is not contained in a pruned subtree, then it must be semantically transformed prior to BURS state assignment. *Transform-subtree*, defined in Figure 5.16(b) and called on line 5 of Figure 5.17, relabels the abstract syntax tree as described in Section 5.3.

```
1: Match-subtree (T) :
2:   for i = 1, ..., arity-of(T) do
3:     Match-subtree (T_i)
4:   od
5:   T.state ← transition(T)
```

Figure 5.18 The BURS pattern matching automaton.

If the new subtree is not pruned, either by containment in a pruned subtree or as a result of relabelling, then object code must be incrementally regenerated for the abstract syntax tree. Recall from Section 5.2 that BURS-based incremental instruction selection is equivalent to incrementally inferring the local rewrite sequence of each node in the abstract syntax tree. In turn, the local rewrite assignment is dependent on the state assigned by the BURS pattern matcher and the inherited goal symbol. Thus, incremental recompilation requires two passes over the extent. The first pass updates the BURS pattern matcher state assignments. The second pass is a top-down traversal of the extent that infers the local rewrite assignment from the state and the inherited goal symbol of each node.

Consistency of recompilation requires that, after incremental pattern matching, the assignment of states to each node of the abstract syntax tree is the same as that resulting from matching the entire abstract syntax tree. The analysis of Section 5.2 determined the set of nodes of the abstract syntax tree that will require consideration by the incremental pattern matcher. The partitioning of the abstract syntax tree into three disjoint sets of nodes guides the determination of the extent (see p. 108). Partition P_1 contains nodes that are independent of the replacement position. No rematching of nodes from P_1 is required. However, the extent may contain subtrees of P_1, rooted by nodes from P_3. Partition P_2 is the new subtree of the abstract syntax tree; it is always contained in the extent. Nodes from partition P_3 need to be sequentially examined until the updated states converge with the original states; some nodes of P_3 are thus potentially part of the extent. In summary, the extent will be either just P_2, or a subtree of the abstract syntax tree rooted by a well defined node of P_3.

```
1: Expand (T, p) :
2:   top' ← p
3:   parent ← parent-of(p)
4:   while top' ≠ ε and T@parent.state ≠ trans(T@parent) do
5:     T@parent.state ← trans(T@parent)
6:     top' ← parent
7:     parent ← parent-of(top')
8:   od
9:   return top'
```

Figure 5.19 Expansion of the extent of recompilation.

State assignment is required for all nodes of P_1. The *Match-subtree* procedure, shown in Figure 5.18, performs a bottom-up traversal of a subtree, assigning the pattern matcher state at each node to the state field of the node, denoted by $T.state$ in Figure 5.18 and in all subsequent figures. *Match-subtree* is an implementation of the BURS pattern matching automaton. The *transition* function, called on line 5 of Figure 5.18, is the state transition function of this automaton. The label of a node n, and the state assignment of its immediate descendants, are examined by *transition* to determine the state assignment of n. In practice, the transition function will be generated from a formal specification of the rewrite system. Section 5.4.2 briefly elaborates on the generation of the transition function.

Let $\delta(P_3)$ be the subset of P_3 containing those nodes at which the state assignment changes as a result of the edit operation and which must be included in the extent of recompilation. If $\sigma(p)$ denotes the state assignment of the node at position p prior to the edit and $\sigma'(p)$ denotes the correct state assignment after the edit, then $\delta(P_3)$ is defined by

$$\delta(P_3) = \{p \in P_3; \sigma(p) \neq \sigma'(p)\}$$

If $\delta(P_3) = \emptyset$, then the extent of recompilation is the new subtree of the abstract syntax tree; that is, the extent of recompilation is simply P_2. Otherwise, if $\delta(P_3) \neq \emptyset$ and *top* in Figure 5.19 is the member of P_3 such that $q \in \delta(P_3) \Rightarrow top\ anc\ q$, then *top* is the root of the subtree that is the extent of recompilation. The function *Expand*, whose algorithm is shown in Figure 5.19, searches for the deepest node of P_3, denoted by *parent*

```
1: Reduce(T, p, goal) :
2:   rule ← ν_σ(goal) where σ = T@p.state
3:   templates ← templates-of(rule)
4:   for i = 1, ..., length-of(templates) do
5:     if t_i = (fragment, Δ) then
6:       fragment ← Expand(Δ, T, p)
7:       Output-code-line(fragment)
8:     elsif t_i = (child_j, goal') then
9:       T@p.j.goal ← goal'
10:      Reduce(T, p.j, goal')
11:    elsif t_i = (chain, goal') then
12:      Reduce(T, p, goal')
13:    fi
14:  od
```

Figure 5.20 Top-down reduction and generation of object code.

in the algorithm, such that the state assignment is unchanged. This node must then be the immediate ancestor of *top* (eventually denoted by *top'* in Figure 5.19) because BURS state assignment depends only on the label of a node and the states assigned to its immediate descendants. That is, if a position $q \in P_3$, is located such that $q \notin \delta(P_3)$, then no ancestor of q can be in $\delta(P_3)$. Thus, if $\delta(P_3) = \emptyset$, then *parent* is the immediate ancestor of *top*, the extent determining position. *Expand* returns *top* if $\delta(P_3) \neq \emptyset$; otherwise, *Expand* returns the point of subtree replacement. Thus, the position returned by *Expand-node* is the extent determining position.

The first, bottom-up, pass of recompilation is completed after *Recompile* has called *Expand* (see line 10 of Figure 5.17). At this point, the extent of recompilation has been determined to be the subtree rooted by *top* and the state assignment of all nodes is consistent. The second pass can then regenerate object code as determined by the local rewrite assignment.

Reduce, shown in Figure 5.20, is passed the abstract syntax tree T, a position p in T and a goal symbol *goal*. It emits the object code resulting from the rewriting of the subtree of T rooted at p into the symbol *goal*. Thus, having determined the extent of recompilation, the call to *Reduce* in *Recompile* regenerates all object code necessary to make the object code consistent with the source code.

rvalue → *sum*(*rvalue*, *smallint.1*), cost 1

```
@1
add %17,$1.constant_value
```

(a) an operator rule

rvalue → *lvalue*, cost 1

```
@@
ld [%16],%17
```

(b) a chain rule

Figure 5.21 Template specifications for rewrite rules.

The initial instance of *Reduce* is created for the extent determining node by *Recompile* in Figure 5.17. If the extent determining node, denoted by *top* in Figure 5.17, is the root of the abstract syntax tree, then the goal symbol for the initial instance of *Reduce* is the fixed goal of the rewrite system; in this case, the initial instance is created on line 12 of Figure 5.17. Otherwise, the goal for the initial instance of *Reduce* is obtained from the intermediate information stored in the *goal* field of each node of the abstract syntax tree; in this case, the initial instance is created on line 14 of Figure 5.17.

No explicit representation of local rewrite assignments is kept. Instead, rewrite rule applications at the node p are implicit in the recursive instances of *Reduce*. Each such instance corresponds to the application of a single rewrite rule at a subtree of the abstract syntax tree. More precisely, the instantiation *Reduce*(T, p, *goal*) represents the application of a rule from the local rewrite assignment of $T@p$ that rewrites to the symbol *goal*. Prior to the creation, on line 10 of Figure 5.20, of the first recursive instance of *Reduce* at a node $p.j$ of T, the goal symbol is stored in the *goal* field of the node, denoted by $T@p.j.goal$ in Figure 5.20, and in all subsequent figures. The rule applicable to this rewrite is uniquely determined by the state of $T@p$ and the goal symbol. Recall that, for each state σ, the vector ν_σ maps goal symbols to rules (see Section 5.1.3, p. 105). Thus, line 2 of Figure 5.20 is able to infer the rule applicable to this instance of *Reduce*.

The recursive invocations of *Reduce* are guided by a list of templates that are associated with each rule. This list is derived from object code template specifications in an architecture description. An object code template specification is either

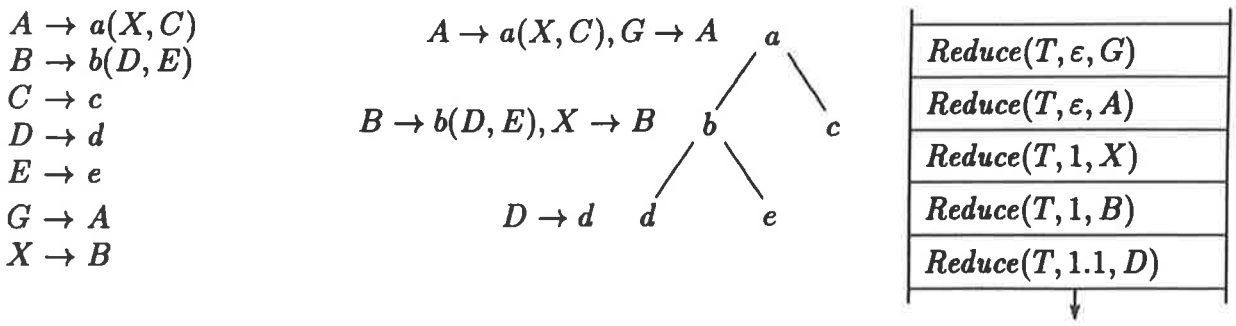
- an instruction fragment specification,
- the symbol @i, where i is an integer, or
- the symbol @@.

For example, Figure 5.21(a) and (b) show the list of template specifications associated with an operator rule and a chain rule, respectively. The specifications associated with the operator rule denote a (**child**$_1$, *rvalue*) and a (**fragment**, Δ) template. The specifications associated with the chain rule denote a (**chain**, *lvalue*) and a (**fragment**, Δ) template. The interpretations of these templates are described below.

The function *templates-of* called on line 3 of Figure 5.20, returns the list of templates associated with the rewrite rule applied by the particular instance of *Reduce*. This function is derived from the template specifications of the architecture description at code generator generation time.

A (**fragment**, Δ) template in an instantiation of *Reduce* at position p inserts a fragment of object code into the object code file; Δ is a string template synthesised from an instruction fragment template specification associated. The function call $Expand(\Delta, T, p)$ on line 6 of Figure 5.20 constructs a piece of object code from a pattern Δ, and the semantic information encoded in the abstract syntax tree node T. For example, expansion of the string template resulting from the instruction fragment specification in Figure 5.21(a) involves the substitution of the value of the instance of the attribute `constant_value` at the first child of p for the string "`$1.constant_value`" in the string "`add %17,$1.constant_value`".

A (**child**$_j$, *goal'*) template denotes the reduction and generation of object code for the j^{th} operand of the rewrite application corresponding to an instance of *Reduce* at position p. It is derived from a template specification of the form `@`j, associated with an operator rule, such as in Figure 5.21(a). The symbol *goal'*, given by the j^{th} operand of the rewrite rule, is stored as the goal symbol for the local rewrite assignment of $T@p.j$ on line 9 of Figure 5.20 and the recursive call to *Reduce* on line 10 creates an instance of *Reduce* that generates the object code associated with the subtree $T@p.j$. Storage of this goal symbol in a field of the node is required to enable later incremental recompilation after edit operations at $T@p$ or a descendant. The new instance of *Reduce* thus created represents the local rewrite assignment of the node $T@p.j$ that rewrites the subtree into the symbol *goal'*.



(a) a rewrite system (b) a local rewrite assignment (c) instances of *Reduce*

Figure 5.22 Implicit representation of a local rewrite assignment.

A (**chain**, *goal'*) template denotes the application of a chain rule to rewrite the subtree into the symbol *goal'*. It is derived from a template specification of the form @@, that is associated with a chain rule, such as in Figure 5.21(b). The recursive call to *Reduce* on line 12 of Figure 5.20 creates the new instance of *Reduce* that emits the object code associated with the chain rule that rewrites *T*@*p* into *goal'*. The new instance of *Reduce* thus created represents the inclusion of a chain rule into the local rewrite assignment of the node *T*@*p*.

The implicit representation of local rewrite assignments is illustrated in Figure 5.22. A simple rewrite system consisting of five pattern rules and two chain rules is defined in Figure 5.22(a). The subtree shown in Figure 5.22(b) can be rewritten to the symbol G with a normal form rewrite sequence. The local rewrite assignments for the nodes labelled with a, b and d are shown in the figure. At the point when code is being generated for the node labelled d, the code generator run-time stack will contain the instances of *Reduce* depicted in Figure 5.22(c), where T is used to denote the abstract syntax tree fragment in Figure 5.22(b). The two rewrite applications in the local rewrite assignment of the node labelled a are manifested in the two instances of *Reduce* at position ε. Similarly, the rewrite applications for nodes b and d are manifested in the other depicted instances of *Reduce*.

As the algorithm is specified in Figure 5.20, a recursive instantiation of *Reduce* is required for each chain rule application at a node. Chain rules can be eliminated from the rewrite system by generating additional pattern rules as follows:

- close the set of chain rules of the rewrite system,
- if $g \rightarrow P$, with cost C, is a pattern rule, then for each chain rule of the form $g' \rightarrow g$, with cost C', include the pattern rule $g' \rightarrow P$, with cost $C + C'$, and
- remove all chain rules from the rewrite system.

For example, chain rules can be eliminated from the rewrite system of Figure 5.22(a) by adding the pattern rules $G \rightarrow a(X, C)$ and $X \rightarrow b(D, E)$, and removing the chain rules $X \rightarrow B$ and $G \rightarrow A$. The local rewrite assignment to the node labelled a in Figure 5.22(b) would then consist of just the application of $G \rightarrow a(X, C)$ and only a single instance of *Reduce* for position ϵ would appear on the run-time stack.

This elimination of chain rules from the rewrite system guarantees that the local rewrite assignment of each node contains the application of a single pattern rule that is uniquely determined by the state of the node and the inherited goal symbol. Thus, the **chain** template components are no longer required. While this optimisation simplifies other aspects of the incremental instruction selection algorithm, it is not exploited in the prototype implementation described in Chapter 7. Further research is required to determine the effect of chain rule elimination and the introduction of additional operator rules on the size of the tables required by the BURS pattern matching automaton.

An additional optimisation is to coalesce the semantic relabelling of the abstract syntax tree with bottom-up pattern matching. Transformation would then be performed prior to descending to subtrees. If a subtree is not pruned, then its children are visited, followed by the state assignment to the node. While this eliminatation of a tree traversal would further speed recompilation, the prototype implementation does not exploit this optimisation, to avoid the additional complexity in the BURS pattern matcher generator.

As presented so far, the BURS-based instruction selection algorithm fails to address several important issues:

- handling of changes to declarations,
- management of the object code and the source to object code mapping, and
- branch updating.

Changes to declarations

An edit to a declaration has the potential to require a recompilation which has a large extent and consists of several independent subtrees of the abstract syntax tree. For example, the insertion of a new declaration in a block may affect the allocation of space in the activation record of other variables declared in the same block. Each use of a variable so affected must be included in the extent of the recompilation. Efficient handling of such edits is mandatory in an integrated programming environment. Careful consideration of the interaction between incremental semantic analysis and incremental code generation is required to handle declaration edits well. An analysis of the propagation of semantic information after the replacement of a subtree in the abstract syntax tree appears in Section 6.2.

Managing the object code

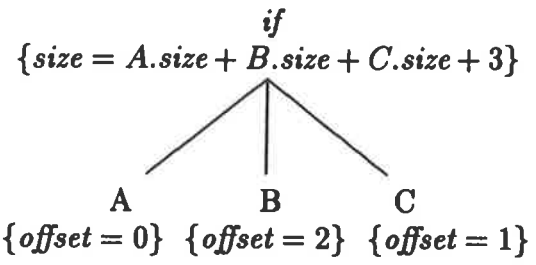
As presented so far in this section, the BURS-based incremental instruction selection algorithm has a very fine, sub-statement level granularity. Potentially, any node in the abstract syntax tree may be incrementally recompiled. Thus, the location and size of the object code for any node in the abstract syntax tree must be available.

Two pieces of integer data are stored in each node to implement the object code map. Firstly, the *size* attribute indicates the length of the object code associated with the subtree rooted at the node. Secondly, the *offset* attribute indicates the location of this code within the code associated with the local rewrite sequence assigned to the parent. The maintenance of these attributes is required only at abstract syntax tree nodes that are associated with object code.

rvalue → if (rvalue, rvalue, rvalue)

```
@1
subcc %17,1,%g0
be <<1>>
@2
ba <<2>>
<<1>> @3
<<2>>
```

(a) rewrite rule and local code



(b) attribution of a tree fragment

Figure 5.23 The *size* and *offset* attribution.

The *size* and *offset* attributes are illustrated in Figure 5.23. Figure 5.23(a) gives a rewrite rule for nodes labelled with the *if* operator in some unspecified expression-oriented language; it also depicts object code for such nodes, within which **@1**, **@2** and **@3** indicate the places for the object code for the child nodes of this *if* operator. Figure 5.23(b) shows a fragment of an abstract syntax tree that has been matched by this rule. The *size* attribute of the fragment is evaluated by summing the *size* attributes of the subnodes and adding the size of the code associated with the local rewrite sequence assigned to the node. No instructions precede the expansion of node A, and so A.*offset* is zero. The local rewrite sequence for the *if* node inserts two instructions between the expansions of its first and second children. Thus, B.*offset* is two. One instruction separates the expansions of the second and third children; hence, C.*offset* is one.

From the synthesised *size* attribute and the inherited *offset* attribute, the object code location associated with an abstract syntax tree node may be inferred. For example, if L is the object code location for the *if* node in Figure 5.23(b), then locations for its children are as follows:

$$\textit{location-of}(A) = L + 0$$

$$\textit{location-of}(B) = \textit{location-of}(A) + A.\textit{size} + 2$$

$$\textit{location-of}(C) = \textit{location-of}(B) + B.\textit{size} + 1$$

Evaluation of *size* and *offset* can occur during the second, top-down, pass of the BURS incremental instruction selection algorithm. A modification of the *Reduce* procedure,

```
1: Reduce(T, p, goal) :
2:   rule ← ν_σ(goal) where σ = T@p.state
3:   templates ← templates-of(rule)
4:   offset ← 0
5:   for i = 1, ..., length-of(templates) do
6:     if t_i = (fragment, Δ) then
7:       fragment ← Expand(Δ, T, p)
8:       T@p.size ← T@p.size + length-of(fragment)
9:       offset ← offset + length-of(fragment)
10:      Buffer-object-code-fragment(fragment)
11:    elsif t_i = (child_j, goal') then
12:      T@p.j.goal ← goal'
13:      T@p.j.size ← 0
14:      T@p.j.offset ← offset
15:      Reduce(T, p.j, goal')
16:      T@p.size ← T@p.size + T@p.j.size
17:      offset ← 0
18:    elsif t_i = (chain, goal') then
19:      Reduce(T, p, goal')
20:    fi
21:  od
```

Figure 5.24 Evaluation of *size* and *offset* in *Reduce*.

performing the requisite calculation, is presented in Figure 5.24; this is an extension of the algorithm in Figure 5.20.

The size attribute of the node at position p is zeroed prior to the creation of the first instance of *Reduce* for p (Figure 5.24, line 13). Similarly, the offset attribute for the j^{th} child of p is set, at line 14, to the current value of *offset* in the parent instance of *Reduce*, prior to creation of the instance for $p.j$. The size is subsequently adjusted when a new fragment of object code is added to the code buffer (line 8) and when code has been generated for a child (line 16).

The handling of object code fragments is very much dependent on the implementation details of the environment. However, conceptually, the object code can be considered to be a one-dimensional array of target machine instructions, referred to as the *code file*. Operations are provided to move blocks of code about the code file and to copy instructions from a buffer into the code file.

```
1: Recompile (T, p, goal, oldsize) :
2:   if is-pruned (T@p) then
3:     Prune-subtree (T, p)
4:   else
5:     Transform-subtree (T, p)
6:   fi
7:   Clear-object-code-buffer()
8:   let k = arity-of(T@p)
9:   if T@p.relabel ≠ φ_k then
10:    Match-subtree (T@p)
11:    top ← Expand (T, p)
12:    if top = p then
13:      Reduce(T, p, goal)
14:    else
15:      Reduce(T, top, T@top.goal)
16:    fi
17:  else
18:    T@p.size ← 0
19:  fi
20:  move block [location-of(T@top) + oldsize ... end-of-file]
21:    to location-of(T@top) + T@top.size
22:  Copy-code-buffer(location-of(T@top))
23:  q ← parent-of(top)
24:  while q ≠ ε do
25:    T@q.size ← T@q.size + T@top.size − oldsize
26:    q ← parent-of(q)
27:  od
```

Figure 5.25 Object code management in *Recompile*.

The buffer must be cleared prior to invoking *Reduce* in *Recompile*. A modified version of *Recompile*, originally presented in Figure 5.17, is shown in Figure 5.25. The buffer is cleared at line 7, prior to the generation any code. *Reduce* stores the generated code in the code buffer (see Figure 5.24, line 10). After generation, if the size of the object code fragment associated with $T@top$ has changed as a result of the recompilation, then the code file must be adjusted accordingly. If the new fragment is larger than the old fragment, then space must be created in the object file by moving the block of code that occurs after the code for $T@top$ down the code file. This is illustrated in Figure 5.26(a), where *oldsize* is the size of the old code fragment and *newsize* is the size of the new code fragment. Conversely, if the new code fragment is smaller than the old fragment then the following block must be moved up the code file; this is illustrated in Figure 5.26(b). The

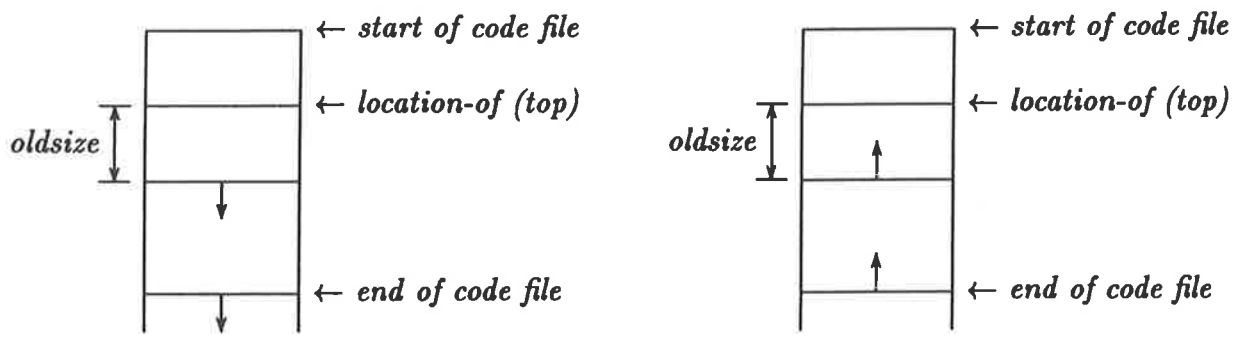


Figure 5.26 Code file adjustment.

block movement operation called on lines 20 and 21 of the modified version of *Recompile* in Figure 5.25 performs the required code file adjustment. Finally, the buffered object code is copied into the correct location of the code file on line 22 in this figure. This scheme is the same as the object code management scheme used in DICE [Fritzson95].

Variation of the size of the object code associated with the subtree *T@top* also affects the object code size of ancestors of *top*. The final part of the modified *Recompile* procedure, starting on line 23 of Figure 5.25, updates the size attribute of these nodes accordingly.

Storing adequate information to locate the object code for any node in the abstract syntax tree represents a significant cost in the storage requirements. This does not compare favourably with the DICE system (see Section 2.4.2). However, not every node of the abstract syntax tree requires the storage of object code information. Indeed, there are typically many nodes in an abstract syntax tree that are not associated with any object code. The fraction of nodes for which this information is required can be further reduced if the granularity of recompilation is made coarser (see p. 142).

Branch updating

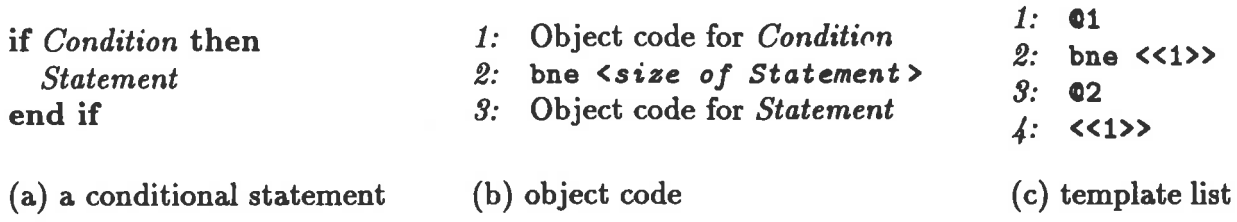
Branch instructions are used to implement the statement level sequence control constructs of programming languages; such constructs include loops and conditional statements. The targets of branch instructions must be modified after incremental recompilation. For example, consider the simple *if* construct in Figure 5.27(a). The object code

for this statement, shown in Figure 5.27(b), consists of the object code for the condition, then a conditional branch (on line 2) and finally the code for the statement. The offset field of the branch instruction is determined by the length of the object code emitted for *Statement*. If *Statement* is edited, resulting in a change of the length of its object code, then the offset field of the branch instruction must be altered.

A simple addition to the BURS-based incremental instruction selection algorithm is able to handle a restricted class of such branch instructions. Consider the list of code templates associated with a rewrite rule and assume that the rewrite system has also been extended to eliminate chain rules from the local rewrite assignments. The *Reduce* algorithm of Figure 5.20 must then handle child rewriting templates and instruction fragment templates. Introduce a new kind of template, the *label* template, which defines a position in the code generated due to the application of a rule to a node. The offset field of branch instruction in an instruction fragment template may then be defined as a reference to a label template that is associated with the same rewrite rule. The code templates applied to a node uniquely determine the occurrences of branch instructions in the code associated with that node. The templates, in turn, are uniquely determined by the rewrite rules applied to that node.

For example, the list of templates for the rewrite rule $stat \rightarrow if(expr, stat)$, that could apply to the abstract syntax tree representing the code in Figure 5.27(a), is shown in Figure 5.27(b). No code is emitted for the label template on line 4 of Figure 5.27(c). Instead, it denotes a possible destination for branch instructions occuring in the template. The branch instruction on line 2 uses the label `<<1>>` as its destination. *Reduce* can then be modified so that, after generating code for the children of a node, the offsets to each label in the list of templates can be calculated and the instruction fragment modified accordingly.

After incremental recompilation, that is after line 27 of *Recompile* in Figure 5.25, branch instructions that occur in nodes that are ancestors of *top* must be adjusted. Nodes that require branch updating can be visited in a tree walk from *top* to ε. Label offsets are



(a) a conditional statement (b) object code (c) template list

Figure 5.27 Branching in a conditional statement.

calculated and references to labels in the instruction templates patched when the node is visited.

An outline of a suitable algorithm for branch updating is shown in Figure 5.28. The elimination of chain rule applications from the rewrite sequence allows the inference, on line 2 of Figure 5.28, of the templates applicable at the node p from just the state assignment at p. The offsets of each label in the list of templates can then be calculated on lines 3 to 13 of Figure 5.28. Next, label references in the fragments of code are corrected on lines 14 to 21. Finally, any branches at ancestors of p are updated by the recursive call to *Update_Branches* on line 24.

This approach to incremental branch updating is similar to the scheme used in the DICE incremental code generator [Fritzson95].

While this scheme is adequate for the simpler control structures, the branching template and its destination label must appear in the list of templates for the same rewrite rule. This is inadequate for a number of common control structures, such as the Ada *exit* statement. For such structures, a more sophisticated scheme, requiring the propagation of branch information about the abstract syntax tree, is required.

In order to handle branch updating for control structures where the target of the branch does not appear in the list of templates for the same rewrite rule, information about branch targets must flow between nodes of the abstract syntax tree. This requires much closer integration with the incremental semantic analyser of the incremental code generator. For example, the prototype incremental code generator described in Chapter 7 uses an implementation of the incremental visit-sequence evaluator described in [Reps89a]. In the case of an Ada *exit* statement, the branch target must first be inferred

```
1: Update_Branches (T, p) :
2:   templates ← templates-of(T@p.state)
3:   offset ← 0
4:   for i = 1, ..., length-of (templates) do
5:     if t_i = (fragment, Δ) then
6:       fragment ← Expand(Δ, T, p)
7:       offset ← offset + length-of(fragment)
8:     elsif t_i = (child_j, goal') then
9:       offset ← offset + T@p.j.size
10:    elsif t_i = (label, l) then
11:      l_offset ← offset
12:    fi
13:  od
14:  for i = 1, ..., length-of (templates) do
15:    if t_i = (fragment, Δ) then
16:      fragment ← Expand(Δ, T, p)
17:      if fragment contains any label references then
18:        substitute label offsets for label references in fragment
19:        emit fragment to the object code file
20:      fi
21:    fi
22:  od
23:  if p ≠ ε then
24:    Update_Branches(T, parent-of(p))
25:  fi
```

Figure 5.28 Simple branch updating.

using appropriate rules in the semantic description; then, the offset can be inferred and propagated to the node that represents the *exit* statement. This is not implemented in the proof-of-concept prototype that is described in Chapter 7.

A weakness of this approach is that varying length jump instructions cannot be handled. In many architectures, the length of a jump instruction varies with the magnitude of the offset concerned. However, because instruction selection occurs before any branches are computed, a pessimistic assumption about the magnitude of the branch must be made. This has a detrimental effect on the quality of the generated code.

5.4.2 Generation from a target architecture description

Several aspects of the BURS-based incremental instruction selection algorithm described above depend on the target architecture description. This description consists of:

- declaration of the new operators used in the semantic relabelling,
- definition of the semantic relabelling function, and
- rewrite rules and associated lists of instruction templates.

The semantic relabelling function is generated from the first two of these. The transition function of the BURS pattern matcher, along with the vector ν_σ that maps states and goal symbols into rules, is generated from the rewrite system. Finally, the template table used by *Reduce* is generated from the template lists.

A formal specification of the relabelling and of the rewrite system are compiled into modules that are used in the prototype implementation described in Chapter 7. The prototype also supports the generation of the environment from a formal specification of the source programming language and its static semantics. Details of the generation of components of the integrated programming environment from a formal specification are to be found in Chapter 7.

5.5 Variations

The basic incremental instruction selection algorithm can be modified in a number of ways. For example,

- adjusting the granularity of recompilation, and
- integration with a systematic approach to register allocation.

Intermediate information required for incremental code generation can significantly increase the storage requirements of an integrated programming environment. The basic algorithm presented in this chapter requires that each node of the transformed abstract syntax tree be decorated with such information. However, the storage requirements can be reduced if the intermediate information is eliminated from certain classes of node.

For example, in an abstract syntax describing the Ada programming language, the sort *Expr* includes the operators that represent Ada expressions. Subtrees rooted at expression nodes are typically quite shallow; thus, calculating the intermediate information on demand is not expensive. Eliminating intermediate information from these nodes will reduce the storage requirements of the intermediate representation at a small cost in recompilation time. The specification of operators for which intermediate information will be maintained would be part of the architecture specification for a particular programming language and target architecture.

This variation has the effect of increasing the recompilation grain size. The BURS-based incremental instruction selection algorithm requires that, for a given subtree replacement, the recompiled subtree is rooted at the extent determining node, or an ancestor of this node. Potentially, sub-statement level granularity can be attained. The designer of an incremental code generator based on this algorithm must select a suitable granularity after consideration of:

- the storage requirements for intermediate information, and
- the demands of other components of the incremental code generator, such as a register allocator and an optimiser.

Reducing the storage demands of the BURS-based approach to incremental compilation is an area for future research.

The prototype implementation described in Chapter 7 uses an ad hoc approach to register allocation. The node-level granularity implies that good register allocation requires the propagation of information across grain boundaries. Bivens' incremental register allocation algorithm (see Section 2.4.6) is suitable in this case. However, this method requires the incremental maintenance of an interference graph, which is not immediately available from the intermediate representation maintained by the BURS-based incremental instruction selection algorithm. Future research into the BURS-based approach to incremental compilation should consider the reconciliation of the BURS-based incremental instruction selection algorithm with Biven's approach to the incremental maintenance of the interference graph.

5.6 Conclusions

This chapter has demonstrated the feasibility of basing an incremental instruction selection algorithm on a retargetable code generation technique used in non-incremental compilers. Pelegrí-Llopart's bottom-up rewrite system (BURS) based approach to instruction selection was chosen after consideration of several other retargetable instruction selection algorithms.

Consideration of the required input to the BURS pattern matcher led to the definition of an intermediate representation suitable for incremental instruction selection. The *Transform-subtree* algorithm in Figure 5.16(b) incrementally constructs the intermediate representation in a single top down pass of the update subtree. Furthermore, the intermediate representation overlays the abstract syntax tree.

After the abstract syntax tree has been incrementally transformed, the BURS-based incremental instruction selection algorithm generates object code in two passes of the extent of recompilation. The first, bottom-up pass incrementally updates the pattern matcher state assignment to each node. The second, top-down pass generates any necessary object code. Thus, the incremental instruction selection algorithm regenerates object code in time proportional to the number of nodes in the extent of recompilation.

The quality of the emitted code is determined by the quality of the architecture description. The BURS-based incremental instruction selection algorithm always chooses the least cost cover of the transformed abstract syntax tree, with respect to the rewrite rules, cost metrics and code fragments that comprise the architecture description. The code quality is constrained by the static cost metrics associated with the rules, the absence of any other optimisations and the sophistication of whatever register allocation strategy is used in the code generator.

The next chapter derives a greedy parallel incremental code generation algorithm from the BURS-based incremental instruction selection algorithm described in this chapter. A prototype implementation of the parallel algorithm is then described in Chapter 7.

Chapter 6

Parallel incremental compilation

The incremental instruction selection algorithm presented in Chapter 5, which is based on bottom-up rewrite system (BURS), regenerates consistent object code after the replacement of certain subtrees in the abstract syntax tree program representation. However, two issues have still to be dealt with:

- handling of arbitrary subtree replacement, and
- the integration of a code generator, based on this algorithm, into an integrated programming environment.

This chapter describes an incremental code generator, using the BURS-based incremental instruction selection algorithm, that is integrated into an integrated programming environment. The resulting code generator greedily recompiles object code, after each subtree replacement in the program source. Furthermore, the code generator executes in parallel with the remainder of the environment. In particular, code generation occurs concurrently with editing, thus minimising the impact that greedy incremental code generation has on the response time of the environment. In contrast, if the execution of the greedy incremental code generator is interleaved with the execution of the editor, then code generation potentially has a significant impact on response time.

Greedy parallel incremental recompilation is discussed in Section 6.1, partly through an analysis of the PSEP environment [Sawamiphakdi84, Ford85]. This leads to an approach to the construction of an incremental code generator, as a view of the MultiView distributed integrated programming environment that was described in Section 3.2. At the heart of this view is the parallel incremental instruction selection algorithm, to be described in Section 6.3.

As will be shown in Section 6.2, consideration of the interaction between incremental static semantic analysis and incremental instruction selection determines the effect that

arbitrary subtree replacement has on the extent of recompilation due to the propagation of semantic information. A naive design for a parallel incremental code generator, one that directly invokes the *Recompile* procedure of Figure 5.17 (see p. 126), is presented in Section 6.3 and demonstrates both the necessary expansion of the recompilation extent and the greedy code generation paradigm. Subsequently, a decomposition of *Recompile* leads to the synthesis of the improved parallel incremental code generation algorithm presented in Section 6.3. Careful handling of the expanded extent of recompilation due to arbitrary subtree replacements in the abstract syntax tree is a key component of the improved algorithm.

6.1 Parallel greedy incremental code generation

The choice of a strategy for the dynamic interaction between an incremental code generator and the remainder of the environment is a key decision in the design of an incremental compilation facility for an integrated programming environment. Two contrasting alternatives are illustrated by DICE (see Section 2.4.2, p. 30), which uses a demand-driven approach, and SEP (see Section 2.4.5, p. 38), which uses a greedy approach.

Recall the basic definitions from Chapter 2. During editing, a demand-driven incremental code generator marks updated fragments of the source program; marked sections of source code are not recompiled until required, either due to an explicit compile command or a request for execution by the user. A greedy incremental code generator regenerates object code immediately after an update to the program source. Observe that a greedy incremental code generator potentially recompiles a particular fragment many times, as repeated edits occur in the same program locality.

The greedy approach minimises preparation time, that is the delay between a request for execution and the object code becoming ready to execute, but at the expense of the response time of the environment: the recompilation of object code after each edit induces delays before further editing is possible. The demand-driven approach minimises response time, but at the expense of preparation time: less work is required between edits, but modified source code must be recompiled before execution is possible.

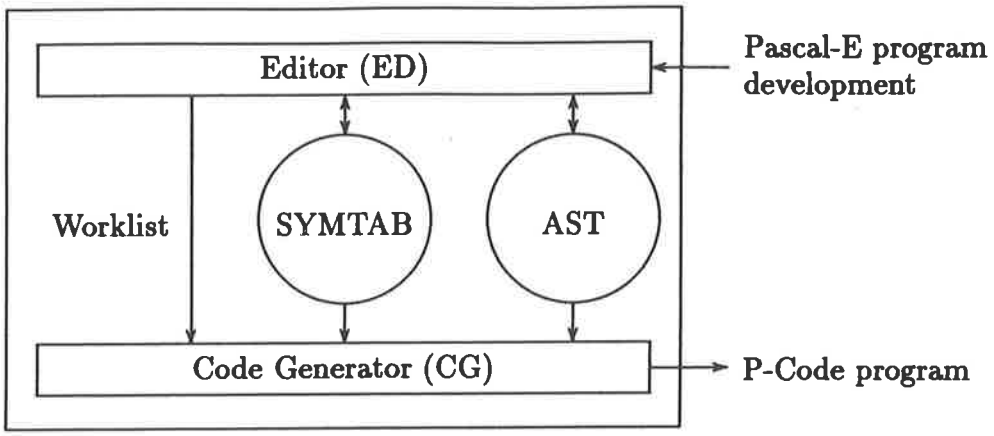


Figure 6.1 Structure of the PSEP system [Ford85, Figure 1].

Sawamiphakdi and Ford observe in [Ford85] that, if the code generator executes in parallel with the editor, then greedy incremental code generation will have less impact on the response time of the editor. The PSEP design in [Sawamiphakdi84, Ford85] is a parallel integrated programming environment that is based on SEP (see Section 2.4.5 p. 38). PSEP couples an incremental code generator with a concurrently executing program editor. In this environment, the code generator and editor concurrently share several data structures. Figure 6.1 illustrates the sharing of the abstract syntax tree (AST) program representation and the symbol table (SYMTAB) by the concurrently executing editor (ED) and code generator (CG) processes. When the editor updates the abstract syntax tree and symbol table, after an edit by the programmer, a message to this effect is placed on the worklist, for later processing by the code generator.

Sawamiphakdi classifies the interaction of the CG and ED processes as a reader/writer system, with the potential for the writer, ED, to interfere with the reader, CG. Potential interference between ED and CG is characterised by considering three locations in the abstract syntax tree at the time of an update. Using the notation from [Sawamiphakdi84, Chapter 4], these locations are:

- e, the location at which the abstract syntax tree was updated,
- c, the point at which the code generator was initiated, and
- t, the current location of the code generator, which is within the subtree rooted at c.

The subtrees of the abstract syntax tree rooted at e and c are denoted by ST(e) and ST(c), respectively, in [Sawamiphakdi84, Chapter 4]. When a message is delivered to CG, indicating that a program update has occurred, e, c and t are compared. Six cases are distinguished:

1. ST(e) and ST(c) are disjoint subtrees,
2. ST(e) is a subtree of ST(c), and e is in the pre-order sequence of ST(c) after t,
3. ST(e) is a subtree of ST(c), and e is in the pre-order sequence of ST(c) before t,
4. the change is made to the node that is being processed, that is $e = t$,
5. ST(c) and ST(e) are identical, that is $e = c$, and
6. ST(c) is a subtree of ST(e).

The algorithm for the PSEP parallel code generator is shown in Figure 6.2.[1] The code generator repeatedly fetches messages from a queue and invokes the code generator subroutine (CGSB), to generate P-code for updated subtrees. CGSB generates P-code in a single pre-order traversal of the updated subtree.

The "handler" MP at lines 6 to 15 of Figure 6.2 is the message processor. It is invoked whenever an update message is delivered to CG by ED. This handler categorises the current state of the pre-order traversal of ST(c) performed by the code generator in the context of the incoming message, with respect to the cases enumerated earlier, and carries out some subset of the following actions:

- queues the update for future processing,
- incorporates the update into the currently executing traversal of ST(c) by the code generator,
- incorporates the update into the currently executing traversal of ST(c) by the code generator and recovers by *rolling back* this traversal, or
- aborts the code generation pass.

First, consider the situation that the new update and the subtree currently being traversed by the code generator are independent. This corresponds to case 1 on p. 147

[1] This is the simplified version from [Sawamiphakdi84], in that it does not consider updates to identifier declarations.

```
1: process CG
2:   procedure CGSB(e)
3:     begin
4:       generate and link code for the subtree at e
5:     end
6:   handler MP(u_d(e, command))
7:     begin
8:       case Subtree (e, c, t)
9:         1 : queue (u_d)
10:        2 : incorporate
11:        3 : either queue (u_d) or incorporate/recover
12:        4 : either incorporate and/or recover
13:        5, 6 : abort CGSB
14:      end
15:    end
16:  begin
17:    initialisation
18:    repeat
19:      while worklist is null do wait
20:      Getnext (worklist, command, e)
21:      if command ≠ RUN then CGSB(e)
22:    unit command = RUN
23:    linearise and assemble
24:  end
```

Figure 6.2 Part of the simplified PSEP algorithm [Sawamiphakdi84, Figure 4-1].

and is processed on line 9 of Figure 6.2. In this case, code generation may continue, unimpeded, and the update is queued for future recompilation.

In contrast, the final cases, namely cases 5 and 6, are processed on line 13 of Figure 6.2 and require the abandonment of the current code generation. In these instances, the subtree being recompiled is either the same as, or a subtree of, the updated subtree. Thus, any code emitted during the traversal is redundant, and so the traversal of the cut subtree is aborted and the new update is queued for later processing.[2]

In case 2, an update has occurred in ST(c). However, the pre-order traversal has not yet reached the updated subtree. Thus, if the traversal continues unimpeded, the update is incorporated into the emitted P-code, as indicated at line 10 of Figure 6.2.

[2] Figure 4-1 in [Sawamiphakdi84] erroneously omits the queueing operation for this case, as does Figure 6.2 above. However, the accompanying text indicates that queueing of the update is needed.

The third case is more complex. If e is not an ancestor of t, then the code already generated during the current pre-order traversal is inconsistent with the new state of the source program. However, the remainder of ST(c) still requires recompilation. Thus, CGSB is allowed to continue, in order to recompile the remainder of ST(c), and the update is queued for future recompilation. Conversely, if e is an ancestor of t, then the subtree currently being traversed by CGSB has been cut from the abstract syntax tree by the editor. In this instance, the traversal is rewound to the node immediately preceding e in the pre-order traversal of ST(c).

The fourth case, when the update occurs at the node currently visited by the pre-order traversal of the code generator, is assumed to be prevented by the low-level mutual exclusion.

Implicit in the algorithm of Figure 6.2 is the assumption that the current location of code generation is a meaningful notion. In PSEP, the code generator emits P-code during a single pre-order traversal of the abstract syntax tree. Thus, the current location, t, is the node that is currently being visited by the traversal. However, a meaningful notion of the current location of code generation cannot be defined for any of the retargetable code generation algorithms surveyed in Chapter 5; certainly, no such notion exists for the incremental instruction selection algorithm described in the latter part of that chapter. Furthermore, the case split, used by MP in Figure 6.2, requires that the order in which nodes are visited, during code generation, is well defined. Again, such an ordering is not possible in a BURS-based instruction selector.

Coarse grained locking of the shared data structure is avoided in PSEP by noticing that the code generator only reads the shared data structures while the editor is able to both read and write. Sawamiphakdi appeals to the concurrent tree manipulation algorithms of [Kung80, Lehman81] to justify the elimination of all high-level locking of the tree structures during the concurrent manipulation of the abstract syntax tree by the editor and code generator [Sawamiphakdi84].

The MultiView system, described in Chapter 3, is a distributed integrated programming environment based on a client-server architecture. An incremental code generator,

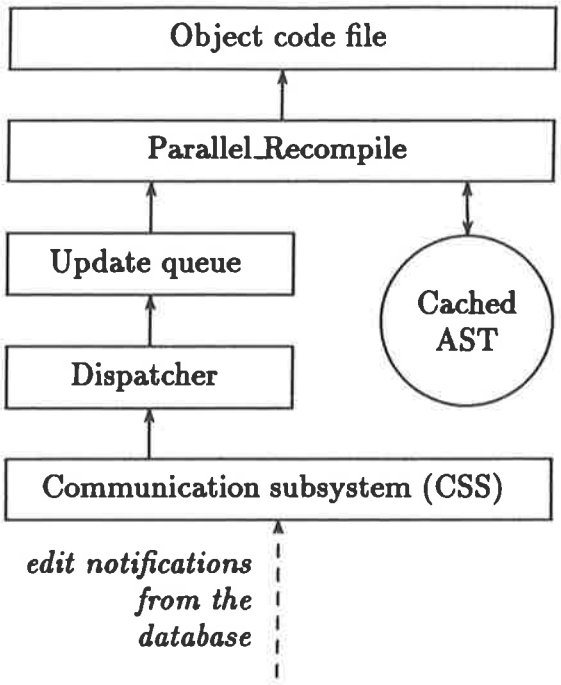


Figure 6.3 Structure of the MultiView code generator view.

implemented as a MultiView view, and depicted in Figure 3.9 (on p. 66), is notified of changes to the program source by the arrival of edit notification messages from the MultiView database. The code generator view, like most other MultiView views, caches a copy of the abstract syntax tree in which it is interested; in this case, the object code generated from this abstract syntax tree is maintained incrementally.

The internal structure of a greedy incremental code generator view for MultiView is shown in Figure 6.3. Edit notifications, arriving from the database, are delivered to the communication subsystem (CSS) and are passed to the dispatcher which is found in all MultiView views (see p. 65). In turn, the dispatcher queues the program update notification on the *Update queue* for later processing by *Parallel_Recompile*. *Parallel_Recompile*, executing in parallel with the dispatcher, removes update notifications from the update queue and incrementally updates the *Object Code File*. The *Object code file* must be updated in an incremental manner, as described in Chapter 5 (see p. 134). The cached copy of the abstract syntax tree is accessed only by *Parallel_Recompile*; thus, in contrast to the PSEP parallel incremental code generator, the program representation that is accessed by the incremental code generator is not shared by multiple processes.

The remainder of this chapter examines *Parallel_Recompile* in detail. In particular, a parallel incremental recompilation algorithm derived from the BURS-based incremental instruction selection algorithm of Chapter 5 is described.

6.2 Interaction with static semantic analysis

The BURS-based approach to incremental instruction selection described in Chapter 5 only deals correctly with edits to subtrees of the abstract syntax tree that correspond to executable fragments of the program. Furthermore, the extent of recompilation is always a complete subtree of the source abstract syntax tree. However, when arbitrary subtree replacements are considered, the extent of the ensuing recompilation is usually a set of subtrees and individual nodes at various positions of source abstract syntax tree. Further positions and subtrees are added to the extent of recompilation in response to the propagation of semantic information that must affect the emitted code.

The parallel incremental code generation algorithm described in this chapter must correctly handle the replacement of arbitrary subtrees of the source abstract syntax tree. This section constructs two sets of nodes, called $affected_1$ and $affected_2$, that characterise the necessary expansion of the extent of recompilation in response to the propagation of semantic information. These sets are subsequently exploited in the *Parallel_Recompile* algorithm that is presented in Section 6.3.

For example, consider the propagation of semantic information after an edit to the Ada block shown in Figure 6.4(a). Storage for the local variable Y is allocated in an activation record frame. In the figure, Y is allocated the 4 bytes of storage located 8 bytes from the start of the activation record. Thus, assuming that the address of the top of the activation record is in the `%fp` register, the following sequence of SPARC instructions implements the assignment statement on line 5 of Figure 6.4(a):

```
add  %g0,20,%l0
st   %l0,%fp[8]
```

Now suppose that the declaration for the variable X is inserted, as shown in Figure 6.4(b). X is allocated the 4 bytes of storage offset 8 bytes from the top of the activation record, and Y is now allocated the 4 bytes offset by 12 bytes from the top of the

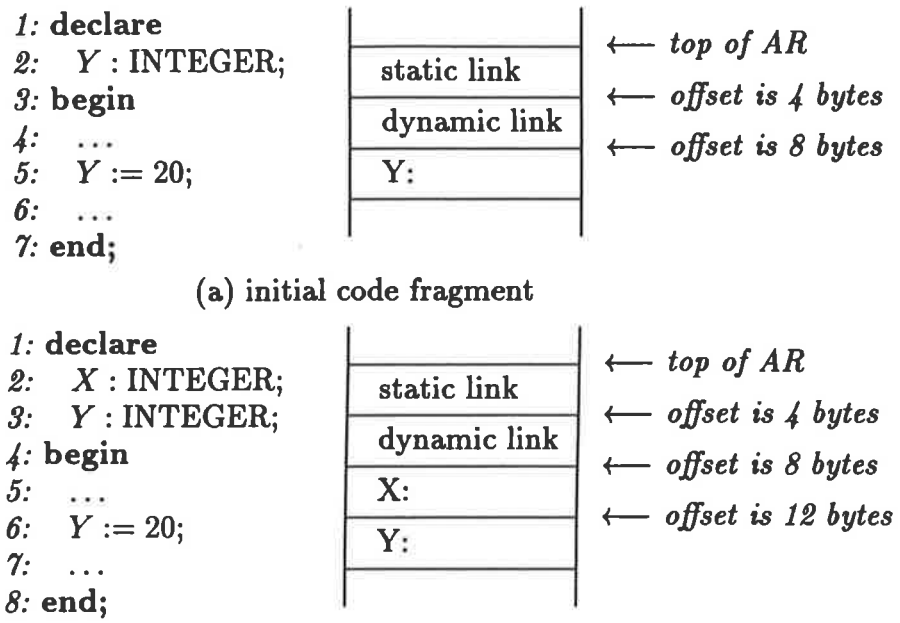


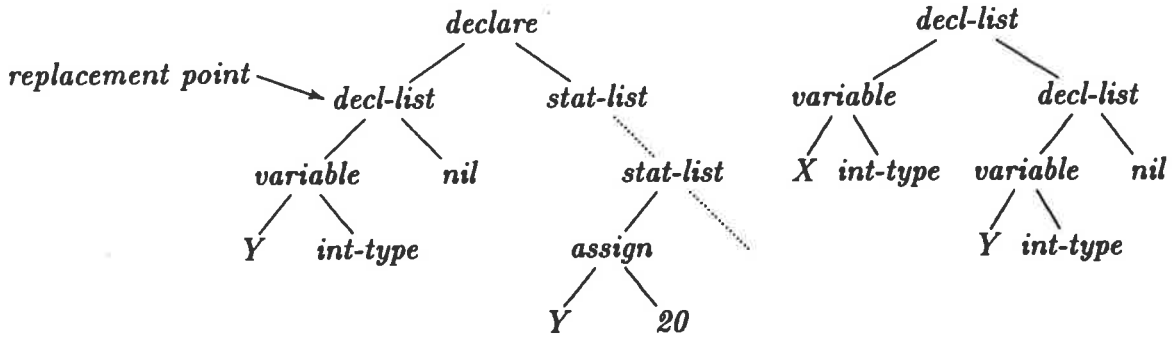
Figure 6.4 Effect of inserting a declaration on an activation record (AR).

activation record. The location in the activation record for the variable Y has changed. Thus, the assignment statement on line 6 of Figure 6.4(b) must be recompiled, to replace the store operation by "`st %10,%fp[12]`".

An abstract syntax tree for the block in Figure 6.4(a) is shown in Figure 6.5(a). The insertion of a declaration for the variable Y is effected by replacing the subtree indicated in Figure 6.4(a) with the tree fragment shown in Figure 6.5(b). However, although the object code for the *assign* node requires regeneration, this node is not initially in the extent of the recompilation caused by the subtree replacement. Some mechanism is required that augments the extent of recompilation with the subtree rooted by the *assign* node.

The derivation of a method for dealing with such arbitrary subtree replacements requires an examination of the means through which information, elicited by the static semantic analysis, percolates into the object code. The impact of semantic information on the emitted code is determined by the rewrite system based architecture specification. Semantic information can be identified at two points in this specification:

- the predicates of the relabelling transformation, and
- the insertion of semantically derived values into object code.



(a) the initial abstract syntax tree (b) the replacement subtree

Figure 6.5 Subtree replacement corresponding to Figure 6.4.

local-var, global-var $\in \text{Cut}_1$
variable $\mapsto$
static_offset $= 0 \Rightarrow$ *local-var*
static_offset $> 0 \Rightarrow$ *global-var*

local.var.1 $\rightarrow$ *local-var*
global.var.1 $\rightarrow$ *global-var*

$\lambda \rightarrow$ *assign (local-var.1, global-var.1)*
`st %l7,%fp[$1.offset]`

$\lambda \rightarrow$ *assign (global-var.1, global-var.1)*
`st %l7,$1.address`

(a) semantic relabelling (b) rewrite rules and code templates

Figure 6.6 Semantic relabelling and code templates for assignment.

In this thesis, semantic information is assumed to be stored in attributes that decorate the nodes of the abstract syntax tree. However, as noted in Section 5.3, this is not mandatory, as long as semantic information can be accessed via nodes of the abstract syntax tree.

A suitable relabelling transformation for the assignment operator, along with SPARC code templates, will now be used to illustrate how any attribute values used in the architecture description impact on the BURS-based incremental instruction selection algorithm.

The relabelling specified in Figure 6.6(a) relabels an abstract syntax tree node, which is initially labelled with the operator *variable*; it is relabelled with either *local-var* or *global-var*, depending on the value of the attribute *static_offset*. If an update of the source abstract syntax tree induces a change in the value of the instance of the *static_offset* attribute for a node labelled with *variable*, then that node must be considered for inclusion

in the extent of recompilation. Thus, any changes to the value of attribute instances that may affect the relabelling of a node must be detected.

The code templates in Figure 6.6(b) relate to two kinds of assignment. For assignment to a local variable, a value from register %17 is stored into a location of the current activation record, the destination argument of the store instruction being constructed from the *offset* attribute of the node. The store instruction associated with assignment to a global variable utilises the *address* attribute of the node. Thus, in this example, if the value of the instance of either the *offset* or *address* attributes for a node labelled with *variable* changes as the result of an update, then that node must be considered for inclusion in the extent of recompilation.

Observe that the relabelling function of Figure 6.6(a) defines pairs for the operator *variable* that differ in form from that described in Section 5.3. Each pair in Figure 6.6(a) consists of a boolean expression and a relabelling operator, rather than the boolean-valued attribute and relabelling operator specified in the construction of Section 5.3. If each such guard expression is considered to define an anonymous, boolean-valued, local attribute of *variable*, then the construction of Section 5.3 is applicable.

The preceding example illustrates the two means by which information inferred during incremental semantic analysis forces the consideration of a node for inclusion in the extent of recompilation:

- if the value of an attribute instance used in a relabelling specification changes, or
- if the value of an attribute instance used in a code template changes.

The following construction defines two sets of positions in the source abstract syntax tree, called $affected_1$ and $affected_2$, that are induced by a subtree replacement. Any node, such that an attribute instance used in the specification of the relabelling transformation has changed value, will be in $affected_1$. Similarly, any node, such an attribute instance used in a code template has changed value, will be in $affected_2$.

Recall the definitions of $pairs_\omega$ and $\mathcal{P}_\omega$ from Section 5.3 (see p. 119). For each operator ω, define the set $\mathcal{P}'_\omega \subseteq \mathcal{P}_\omega$ by

$$\mathcal{P}'_\omega = \{a \in \mathcal{P}_\omega; \exists \omega' \in \Omega', (a, \omega') \in pairs_\omega\}$$

We call $\mathcal{P}'_\omega$ the set of *semantically significant predicates* for ω.

If the guards of the relabelling transformation for an operator ω are specified as boolean-valued expressions, rather than as boolean-valued attributes, as is the case in Figure 6.6(a), then the attributes of ω that appear in the guard expressions are called the *semantically significant attributes* of ω.

Let q be a position of the abstract syntax tree T. Suppose that the subtree of T rooted at q is replaced by t. Let $T' = T_{q \leftarrow t}$, and let q' be the extent determining position of T induced by the subtree replacement (see p. 125). Let p be some other position of T such that $\overline{q' \ anc \ p}$. Thus, the subtree of T rooted at p will not be recompiled by the *Recompile* algorithm of Figure 5.25 on p. 137. However, the subtree replacement and subsequent static semantic analysis could potentially change the value of an instance of an attribute from $\mathcal{P}'_\omega$ at $T@p$. Consequently, the relabelling of $T@p$ may become invalid due to this subtree replacement. For each node p of the abstract syntax tree T not recompiled by *Recompile*, define the set $changed_1$ of semantically significant predicates whose value has changed as a result of the subtree replacement:

$$changed_1(T, T', p) = \{a \in \mathcal{P}'_\omega;\ \omega = label\text{-}of(T@p),\ eval(T, p, a) \neq eval(T', p, a)\}$$

Thus, the set $affected_1$, that includes all the positions of T' that require relabelling after the subtree replacement at q, is given by

$$affected_1(T, T', q) = \{p \in dom(T);\ \overline{q \ anc \ p},\ changed_1(T, T', p) \neq \emptyset\}$$

Referring to the example of Figure 6.6, the only semantically significant attribute of the operator *variable* is *static_offset*. The subtree replacement that is illustrated in Figure 6.5 does not affect the value of *static_offset*; thus, $affected_1$ is empty in this example.

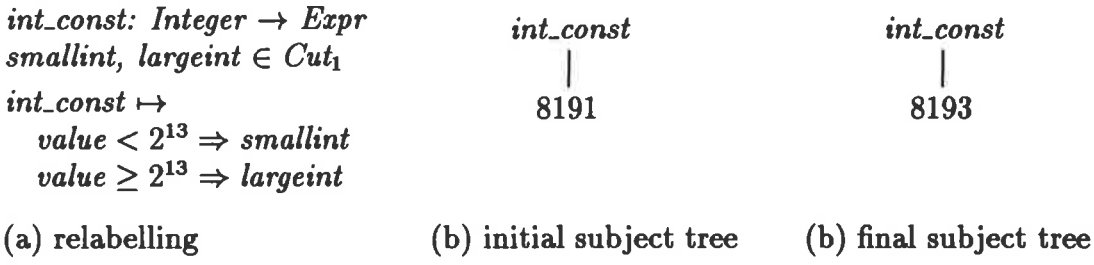


Figure 6.7 Change of relabelling after a source code update.

A different situation is illustrated in Figure 6.7. Recall the example of Figure 5.13 (see p. 116), illustrating the way in which the appropriate choice of a SPARC instruction fragment to load a constant into a register is determined by the value of the constant. The relabelling in Figure 6.7(a) relabels nodes, initially labelled with *int_const*, with either *smallint* or *largeint* depending on the *value* attribute of *int_const* which is, presumably, synthesised from the value of an integer at a leaf node of the abstract syntax tree. Thus, the node labelled *int_const* in the subtree fragment shown in Figure 6.7(b) is initially labelled with *smallint*. Suppose that the integer-valued leaf of the tree fragment is replaced, resulting in the subtree fragment shown in Figure 6.7(c). Now, *smallint* is a cutting operator (see p. 119) and so the *Recompile* algorithm in Figure 5.25 on p. 137 will not regenerate any object code, since the point of replacement, namely the leaf, is initially pruned from the relabelled abstract syntax tree. However, *value* is a semantically significant attribute of *int_const*. Furthermore, its value has changed and so the position of the node labelled with *int_const* is included in $affected_1$, indicating that relabelling at this node may be required.

After a subtree replacement, the relabelling of any node in $affected_1$ may be invalid. Thus, for each node p in $affected_1$, the relabelling operator must be calculated and, if it has changed, then the subtree rooted at p must be added to the extent of recompilation.

An attribute may appear in the code templates associated with a rewrite rule for some operator ω, without being a semantically significant attribute of ω. For example, the *address* and *offset* attributes in Figure 6.6 appear in the code templates for *assign*, but are not semantically significant attributes of *assign*. Thus, if the value of an instance of either of these attributes changes, then the object code for the affected node must

be regenerated. The appearance of attribute instances in code templates must also have some impact on the extent of recompilation. The set of nodes $\mathit{affected}_2$ will include each node of the abstract syntax tree for which the value of an attribute instance used in a code template has changed due to a subtree replacement. This set is constructed by first deducing the complete set of code templates that can apply at nodes labelled by each operator.

Consider a rewrite system, R, from a BURS architecture description consisting of m pattern rules of the form, $X_i \to \omega_i(Y_1, \ldots, Y_{k_i})$, where $k_i = \mathit{arity\text{-}of}(\omega_i)$, and n chain rules of the form, $A_i \to B_i$. For each operator, ω, define the set, R'_ω, of pattern rules from R that match ω by

$$R'_\omega = \{X_i \to \omega_i(Y_1, \ldots, Y_{k_i}) \in R;\ \omega = \omega_i,\ k_i = \mathit{arity\text{-}of}(\omega_i)\}$$

Let R_ω be the transitive closure of R'_ω under the set of chain rules of R. That is, R_ω is the smallest subset of R such that

- if $X \to \omega(Y_1, \ldots, Y_{\mathit{arity\text{-}of}(\omega_i)}) \in R_\omega$ and $Z \to X \in R$, then $Z \to X \in R_\omega$, and
- if $Y \to X \in R_\omega$ and $Z \to Y \in R$, then $Z \to Y \in R_\omega$.

Thus, the set R_ω is the set of all rewrite rules of R that can appear in the local rewrite assignment for a node labelled with ω. For each rewrite rule r from the rewrite system R, define used_r to be the set of attributes that occur in the code templates for r in the architecture description. Then, for each operator ω, the set used_ω of *code determining attributes* that are accessed in code templates for ω is given by

$$\mathit{used}_\omega = \bigcup_{r \in R_\omega} \mathit{used}_r$$

Let q be a position of the abstract syntax tree T. Suppose that the subtree of T rooted at q is replaced by t. Let $T' = T_{q \leftarrow t}$, and let q' be the extent determining position of T induced by the subtree replacement (see p. 125). Then, for each node p of T which is not a descendant of q', define the set $\mathit{changed}_2$ of code determining attributes whose value has changed as a result of subtree replacement:

$$\mathit{changed}_2(T, T', p) = \{a \in \mathit{used}_\omega;\ \omega = \mathit{label\text{-}of}(T@p),\ \mathit{eval}(T, p, a) \neq \mathit{eval}(T', p, a)\}$$

Thus, the set $affected_2$, that includes all the positions at which an instance of a code determining attribute has changed in value due to the subtree replacement at q, is defined by

$$affected_2(T, T', p) = \{p \in dom(T);\ \overline{q'\ anc\ p},\ changed_2(T, T', p) \neq \emptyset\}$$

Together, the sets $affected_1$ and $affected_2$ include all the positions not recompiled by the *Recompile* procedure in Figure 5.25 (see p. 137), that may require recompilation after a subtree replacement. Thus, in order to consistently recompile an abstract syntax tree to object code after an arbitrary subtree replacement, these sets must be synthesised and each member considered for inclusion in the extent of recompilation. The manner in which the members of $affected_1$ and $affected_2$ are inferred will be determined by the technique chosen for incremental semantic analysis. In the prototype implementation described in Chapter 7, the incremental attribute evaluator is modified to generate members of $affected_1$ and $affected_2$ during incremental attribute evaluation (see p. 187). This incremental semantic evaluator is an implementation of the incremental visit-sequence evaluator described in [Reps89a] (see p. 180).

6.3 Parallel BURS-based incremental instruction selection

The greedy incremental code generator view for MultiView, depicted in Figure 6.3 on p. 151, receives notifications of subtree replacements from the MultiView database. The dispatcher places update notifications on a queue for processing by the *Parallel_Recompile* task. Recompilation of updated code is performed by *Parallel_Recompile*. The remainder of this chapter describes an algorithm for *Parallel_Recompile* that is derived from the BURS-based incremental instruction selection algorithm of Chapter 5.

The update queue is the focus of the interaction between the incremental code generator and the remainder of the environment. Each entry in this queue consists of a position p, in the abstract syntax tree, and a subtree t, specifying that the subtree of the abstract syntax tree rooted at p has been replaced by t. The update queue is concurrently accessed by two processes: *Parallel_Recompile* and the dispatcher. Together, *Parallel_Recompile*

```
1: Parallel_Recompile :
2:   ... initialise T, the cached tree ...
3:   Recompile (T, ε, goal)          /* see Figure 5.25, p. 137 */
4:   do
5:     Fetch(Update_queue, p, t)
6:     ... update the cached abstract syntax tree ...
7:     ... update the semantic information ...
8:     ... regenerate the affected object code ...
9:   od
```

Figure 6.8 Outline of the greedy parallel incremental code generation algorithm.

and the dispatcher represent an instance of the producer/consumer problem, and the update queue represents a buffer between producer and consumer. Numerous solutions to this producers and consumers problem have appeared in the literature, such as that for the bounded buffer described in [Hoare78].

An outline of *Parallel_Recompile* is shown in Figure 6.8. First, the cached copy of the abstract syntax tree is initialised, on line 2. The *Recompile* procedure of Figure 5.25 is then called to compile the entire cached abstract syntax tree. At this point, the cached abstract syntax tree, the goal and state annotations, and the object code are all initialised. After this initialisation phase, the incremental code generator repeatedly fetches notification of source code updates from the update queue, updates the cached abstract syntax tree, incrementally updates the static semantic information, and finally incrementally regenerates the affected object code. The object code file is consistent with the program source when processing of the last update is completed and the update queue is empty.

The greedy nature of this approach to incremental code generation is apparent from the structure of the code generator, shown in Figure 6.3, and the outline of the parallel recompilation algorithm in Figure 6.8. As soon as update notifications arrive, they are queued for processing. *Parallel_Recompile* fetches updates from the queue as soon as possible, and immediately recompiles the affected portions of the program. Thus, the incremental compiler interacts greedily with editing by the programmer using the environment.

```
1: Parallel_Recompile :
2:   ... initialise T the cached tree ...
3:   Recompile (T, ε, goal)                    /* see Figure 5.25, p. 137 */
4:   do
5:     Fetch(Update_queue, p, t)
6:     old_goal ← T@p.goal
7:     old_size ← T@p.size
8:     T ← T_{p←t}
9:     ... update the semantic information ...
10:    Recompile (T, p, old_goal, old_size)    /* see Figure 5.25, p. 137 */
11:    for q ∈ affected_1 ∪ affected_2 do
12:      Recompile (T, q, T@q.goal, T@q.size)  /* see Figure 5.25, p. 137 */
13:    od
14:  od
```

Figure 6.9 Naive greedy parallel incremental recompilation.

As explained in Section 6.2, the extent of incremental recompilation must include those parts of the program affected by the propagation of semantic information throughout the abstract syntax tree. A complete version of *Parallel_Recompile* that uses a naive method for dealing with the propagation of semantic information when recompiling the program after an arbitrary subtree replacement, is illustrated in Figure 6.9. After updating the cached abstract syntax tree on line 8, the algorithm updates the static semantic information. The *Recompile* algorithm from Figure 5.25 is invoked in order to generate code for the new subtree. Finally, in a subtle application of brute force, *Recompile* is called, on line 12, for each position in $affected_1$ and $affected_2$.

It is assumed that the sets $affected_1$ and $affected_2$, induced by the subtree replacement, are calculated during incremental semantic analysis. The prototype implementation described in Chapter 7 performs incremental semantic analysis with an incremental visit-sequence attribute evaluator; an adaptation of this evaluator enables the synthesis of $affected_1$ and $affected_2$ during semantic analysis (see p. 187).

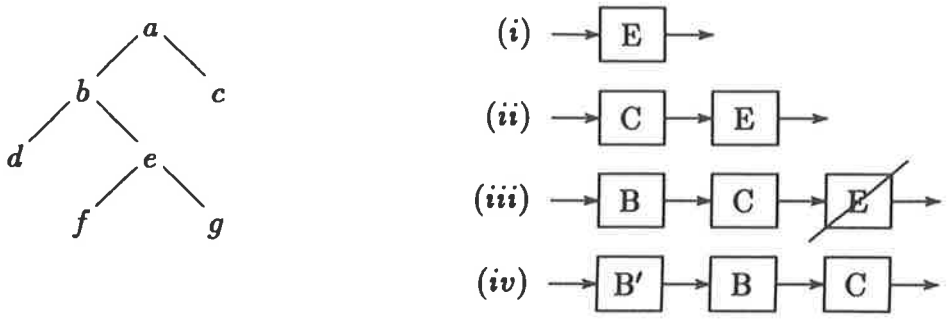
An implementation of the naive parallel recompilation algorithm depicted in Figure 6.9 is a fully functioning incremental code generator that correctly recompiles the program after an arbitrary subtree replacement. However, it evinces several obvious deficiencies:

- while processing a source edit at p, another source edit at q may be delivered that makes the recompilation at p redundant,
- parts of the abstract syntax tree may be recompiled several times during the processing of a single edit, and
- parts of the abstract syntax tree may be unnecessarily recompiled during the processing of a single edit.

The first redundancy occurs when the notification of an edit at position p' is placed on the update queue by the dispatcher, while the code generator is still processing a source edit at position p, where p' *anc* p. The extent of recompilation induced by the edit at p' will certainly contain the extent of the recompilation induced by the edit at p. Thus, any new object code generated due to the edit at p is subsequently overwritten when the update at p' is processed. Hence, newly arriving update notifications should preempt recompilation induced by an earlier update, if the recompilation induced by the earlier update is redundant as a result of the later subtree replacement.

Furthermore, consider the case where the update queue contains an entry for the replacement at p by t and, before the edit at p is fetched by *Parallel_Recompile*, the dispatcher places an entry for the replacement at p', where p' *anc* p, by t' onto the update queue. If the queue is processed in a first-in-first-out (FIFO) manner, then the update of p will be processed before the update at p'; after both entries have been processed, the resulting object code will be consistent with the program source. However, if the FIFO ordering is not observed, then the object code will be incorrect, because the object code for p' will be (at least partially) overwritten by the object code for p. Note however, that $(T_{p \leftarrow t})_{p' \leftarrow t'} = T_{p' \leftarrow t'}$, because p' *anc* p. In summary, when the updates are processed in a FIFO ordering, any object code emitted during the recompilation induced by the update at p, is overwritten when the update at p' is processed.

Modifying the update queue so that update notifications are filtered when new entries are added by the dispatcher eliminates such redundant edits. That is, when an update at the position p is added to the update queue, the queue is scanned for any entries that correspond to updates at descendants of p. Any such entries are removed when the



(a) the initial abstract syntax tree (b) a sequence of update queue configurations

Figure 6.10 Filtering of the update queue.

entry for p is inserted. Thus, these redundant update notifications are never delivered to *Parallel_Recompile*.

For example, consider the tree depicted in Figure 6.10(a). Suppose that an update at node e occurs; thus, an edit E is added to the update queue to represent this update, as shown in line (i) of Figure 6.10(b). Next, suppose that an update at node c occurs; c is not an ancestor of e and hence an edit C representing the update at c is added to the update queue, as depicted on line (ii). Now, suppose that an update at node b occurs; b is an ancestor of e and so the edit E is removed from the update queue before the addition of the edit B for the update at b, as depicted on line (iii). Finally, suppose that an update at a node b' occurs where b' is a descendant of the updated b; b' is neither an ancestor of b nor an ancestor of c, and an edit B′ to represent the b' update is added to the queue. Note that the edit at b' is not redundant because it occurs after the edit at b.

This filtering of the queue, when a new update is added, is assumed for the remainder of this chapter.

The remaining two inefficiencies arise due to interactions between members of $affected_1$ and $affected_2$ arising from an update. Suppose that two distinct positions p and q, such that p *anc* q, are induced into $affected_1$ by a subtree replacement. The *Recompile* procedure recompiles entire subtrees. Thus, assuming that q is processed before p, the object code generated by the recompilation at q is subsequently completely overwritten when the subtree rooted at p is recompiled. Hence, not every position p in $affected_1$ should induce

a separate matching and reduction. In fact, if $p, q \in \mathit{affected}_1$, such that p *anc* q, are both relabelled, then the matching and reduction of the subtree rooted at p makes the subsequent matching and reduction at q redundant. In this instance, the naive algorithm of Figure 6.9 matches and reduces each subtree in its entirety.

Next, suppose that p is induced into $\mathit{affected}_2$ by a subtree replacement. The local rewrite assignments of p, and all the descendants of p, are unaffected. However, changes are required to the object code derived from the local rewrite assignment at p by the application of the appropriate templates. However, the naive algorithm of Figure 6.9 recompiles the entire subtree rooted at p. In fact, only the object code induced by the local rewrite assignment at p requires regeneration.

Finally, suppose that $p \in \mathit{affected}_1$ and $q \in \mathit{affected}_2$ are such that p *anc* q. If p is relabelled, then the entire subtree at p is recompiled. In particular, the object code induced by the node at q is regenerated. Thus, the regeneration of object code at q, induced by its membership in $\mathit{affected}_2$, is redundant. The naive recompilation algorithm performs this redundant recompilation.

In short, three circumstances leading to redundant recompilation by the naive parallel incremental recompilation algorithm in Figure 6.9 have been identified:

- if $p, q \in \mathit{affected}_1$, p *anc* q, and are such that the relabelling of both p and q is changed, then the matching and reduction of the subtree rooted at q is redundant due to the matching and reduction at p,
- if $p \in \mathit{affected}_2$, then the reduction of the complete subtree rooted at p is redundant, and
- if $p \in \mathit{affected}_1$ is relabelled and $q \in \mathit{affected}_2$, and are such that p *anc* q, then reduction at q is redundant.

Tracking the state of recompilation as updates are processed so that these circumstances can be detected is required to efface the above redundant recompilation. One suitable embodiment of the current recompilation state is some explicit representation of the work remaining to be done before the object code is consistent with the source code. An appropriate state representation is found by considering the building blocks

```
1: Recompile (T, p, goal) :
2:   SR ← {p}, MR ← ∅, RR ← ∅
3:   T@p.goal ← goal
4:   while SR ≠ ∅ do
5:     q ← choose (SR), SR ← SR \ q
6:     if is-pruned (T@q) then
7:       Prune-subtree (T, q)          /* see Figure 5.16(a), p. 123 */
8:     else
9:       Transform-subtree (T, q)      /* see Figure 5.16(b), p. 123 */
10:    fi
11:    if T@q.relabel ≠ φ_k, k = arity-of(T@q) then
12:      MR ← MR ∪ {q}
13:    fi
14:  od
15:  while MR ≠ ∅ do
16:    q ← choose (MR), MR ← MR \ q
17:    Match-subtree (T@q)             /* see Figure 5.18, p. 127 */
18:    top ← Expand (T, q)             /* see Figure 5.19, p. 128 */
19:    RR ← RR ∪ {top}
20:  od
21:  while RR ≠ ∅ do
22:    q ← choose (RR), RR ← RR \ q
23:    Reduce (T, q, T@q.goal)         /* see Figure 5.24, p. 136 */
24:  od
```

Figure 6.11 Worklist-based variant of *Recompile* derived from Figure 5.25.

of the BURS-based incremental recompilation algorithm. The *Recompile* algorithm of Figure 5.25 can be decomposed into three subproblems:

- the semantically-driven relabelling of the abstract syntax tree,
- the bottom-up assignment of pattern matcher states, and
- the top-down reduction and generation of object code.

Each of these subproblems operates on subtrees of the abstract syntax tree. Thus, three sets are defined to encapsulate the state of each subproblem of *Recompile* during the processing of updates:

- SR', containing subtrees that require relabelling,
- MR', containing consistently relabelled subtrees that require pattern matcher state assignment, and

- RR', containing matched subtrees that require reductions and object code generation.

A worklist-based adaptation of the sequential *Recompile* procedure of Figure 5.25 is shown in Figure 6.11. The sets SR', MR' and RR' are realised as worklists in this algorithm. The worklist SR corresponds to the set SR' and is initialised to contain p, the subtree to be recompiled. The worklists MR and RR correspond to the sets MR' and RR', respectively, and are initially empty. Recompilation proceeds by successively removing an arbitrary position q from SR, using the *choose* operator, relabelling the subtree rooted at q and then adding q to MR, until SR is empty. Next, positions are successively taken from MR and the subtrees matched, until MR is empty. Finally, positions are successively taken from RR and reduced, until RR is empty.

The sequential version of the worklist-driven *Recompile* in Figure 6.11 is not particularly interesting. In fact, SR never contains more than one position. Likewise, neither MR nor RR ever contain more than a single position. However, it does illustrate the decomposition of the algorithm into subproblems so that each part is associated with a worklist. The status of each subproblem is, in fact, encapsulated in a worklist: successive items are taken from each worklist and processed, and when each worklist is empty, recompilation is complete and the object code is again consistent with the program source.

Similarly, the greedy parallel recompilation algorithm can be based on worklists of positions. Like the worklist-based sequential *Recompile* of Figure 6.11, the parallel version of the greedy incremental recompilation algorithm, shown in Figure 6.12, uses the worklists SR, MR and RR defined earlier. In addition, $affected_1$ and $affected_2$ are treated as worklists that characterise the recompilations induced by the propagation of semantic information.

The algorithm of Figure 6.12 will now be examined in more detail. Like the naive *Parallel_Recompile* of Figure 6.9, the cached abstract syntax tree and the code file are initialised on lines 2 and 3. Next, each worklist is initialised to the empty set because the object code file is, at this stage of execution, consistent with the current state of the program source. Then, notifications of source edits are successively removed from the update queue and processed accordingly.

```
1: Parallel_Recompile :
2:   ... initialise the cached tree, T, and the worklists ...
3:   Recompile (T, ε, goal)              /* see Figure 5.25, p. 137 */
4:   SR ← ∅, MR ← ∅, RR ← ∅
5:   do
6:     Fetch(Update_queue, p, t)
7:     oldsize ← T@p.size
8:     oldgoal ← T@p.goal
9:     T ← T_{p←t}
10:    Update semantic information
11:    SR ← [SR ∪ {p}]
12:    affected_1 ← affected_1 ↓ p
13:    affected_2 ← affected_2 ↓ p
14:    T@p.size ← oldsize
15:    T@p.goal ← oldgoal
16:    while is-empty (Update_queue) and not is-empty (SR) do
17:      Process-relabelling          /* see Figure 6.13, p. 169 */
18:    od
19:    while is-empty (Update_queue) and not is-empty (MR) do
20:      Process-matching             /* see Figure 6.14, p. 170 */
21:    od
22:    while is-empty (Update_queue) and not is-empty (affected_1) do
23:      Process-structure-change     /* see Figure 6.15, p. 171 */
24:    od
25:    while is-empty (Update_queue) and not is-empty (affected_2) do
26:      Process-detail-change        /* see Figure 6.16, p. 172 */
27:    od
28:    while is-empty (Update_queue) and not is-empty (RR) do
29:      p ← choose (RR), RR ← RR \ p
30:      Reduce (T, p, T@p.goal)      /* see Figure 5.24, p. 136 */
31:    od
32:  od
```

Figure 6.12 The parallel BURS-based incremental code generator.

The cached abstract syntax tree and the static semantic information are updated on lines 9 and 10 of Figure 6.12. The processing of source edits is then determined by the contents of the worklists. The position p, at which the subtree was replaced, requires relabelling, matching and reduction, and so is added to the SR worklist for subsequent relabelling of the subtree. The sets $affected_1$ and $affected_2$ include every position of the abstract syntax tree that requires recompilation due to the propagation of static semantic information.

The redundant recompilation noted on p. 164 is eliminated by filtering the worklists. Two operations on positions and sets of positions are used: the *normalisation* of a set of positions, and the *pruning* of a set of positions with respect to a position.

The *normalisation* of P, a set of positions, maps P into its set of *representative positions*. For all $p \in P$, the representative of p in the set P is that node $q \in P$ such that $q \ anc \ p$ and, if $q, q' \in P$ and $q' \ anc \ q$, then $q' = q$. The *normalisation* of P, denoted by $[P]$, is then the smallest subset of P such that

(*i*) for all $p \in P$, the representative of p is in $[P]$, and

(*ii*) if $p, p' \in [P]$ are such that $p \ anc \ p'$, then $p = p'$.

The *pruning* of the set of positions P with respect to a position p is the set $P \downarrow p$, defined by

$$P \downarrow p = \{q \in P;\ \overline{p \ anc \ q}\}$$

In particular, for all $p \in P$, observe that $p \notin P \downarrow p$.

The filtering of SR, after the addition of p, on line 11 of Figure 6.12, avoids the occurrence of distinct positions $q, q' \in SR$, such that $q \ anc \ q'$. The presence of the representative of p in SR guarantees that each descendant of p is matched and reduced. Thus, p and all its descendants are pruned from $affected_1$ and $affected_2$ on lines 12–13.

After the adjustment of the SR worklist due to the source edit, and of the worklists that represent $affected_1$ and $affected_2$ after incremental semantic analysis, the state of recompilation is encapsulated in the contents of the worklists. Thus, recompilation entails the successive removal of positions from each worklist, followed by appropriate processing. However, a newly arrived update notification can induce new redundancies. For example, if $q \in SR$ and an update notification at p, where $p \ anc \ q$, is delivered, then the processing of q is redundant. Thus, the processing of SR, and the other worklists, is preempted by newly arrived update notifications. Further filtering of the worklists, to be described shortly, eliminates the circumstances noted on p. 164 that lead to other redundant recompilation.

```
1: Process-relabelling :
2:   p ← choose (SR), SR ← SR \ p
3:   if is-pruned (T, p) then
4:     Prune-subtree (T, p)          /* see Figure 5.16(a), p. 123 */
5:   else
6:     Transform-subtree (T, p)      /* see Figure 5.16(b), p. 123 */
7:   fi
8:   if T@p.relabel = φ_k, where k = arity-of(T@p) then
9:     MR ← MR ↓ p, affected_1 ← affected_1 ↓ p, affected_2 ← affected_2 ↓ p
10:    ... adjust the code file ...
11:  else
12:    MR ← [MR ∪ {p}]
13:  fi
```

Figure 6.13 Process-relabelling.

The *SR* worklist contains the positions of subtrees that may require relabelling. *Process-relabelling* in Figure 6.13 selects a position p from *SR* and relabels the subtree rooted at p. If p is from an excised section of the abstract syntax tree, then the subtree rooted at p is pruned. Otherwise, the subtree is relabelled by the *Transform-subtree* procedure in Figure 5.16(b) on p. 123. If, after the relabelling, p has been excised, then no object code is derived from p or from any of its descendants, and the code file is adjusted accordingly on line 10. Otherwise, p is added to *MR* for subsequent matching and reduction.

Process-relabelling filters the worklists to eliminate potential redundant recompilations. Firstly, if p is excised by relabelling, it is pruned from *MR*, $affected_1$ and $affected_2$ at line 9. Secondly, *MR* is normalised when p is inserted at line 12, because the matching of p implies that each descendant of p is also matched.

Compare the *Process-relabelling* algorithm of Figure 6.13 with the original *Recompile* algorithm depicted in Figure 5.25 on p. 137. In *Recompile*, if the position p is not excised by relabelling, then matching, reduction and code generation occurs. This is mirrored, in *Process-relabelling*, by adding p to the *MR* worklist for subsequent matching and reduction.

Positions in *MR* denote subtrees that require pattern matching. *Process-matching* in Figure 6.14 selects a position p from *MR*, on line 2, and invokes the BURS pattern

```
1: Process-matching :
2:   p ← choose (MR), MR ← MR \ p
3:   Match-subtree (T, p)          /* see Figure 5.18, p. 127 */
4:   top ← Expand (T, p)           /* see Figure 5.19, p. 128 */
5:   RR ← [RR ∪ {top}]
6:   affected_2 ← affected_2 ↓ top
```

Figure 6.14 Process-matching.

matching automaton on line 3. The extent determining position is deduced by the *Expand* function from Figure 5.19 on p. 128, and stored in *top*. The subtree, rooted at *top*, is then added to the *RR* worklist for reduction.

Both *RR* and $affected_2$ are filtered by *Process-matching*. Firstly, *RR* is normalised, after the inclusion of *top*. Secondly, $affected_2$ is pruned at *top*, because the presence of the representative of *top* in *RR* guarantees the reduction of each descendant of *top*.

The members of $affected_1$ are those positions at which the value of a semantically significant attribute instance has changed. For each position p selected from $affected_1$ by *Process-structure-change* in Figure 6.15, the new relabelling operator is calculated.[3] If the relabelling operator is unchanged, then no further processing at p is required. Otherwise, the change in semantic information induces a change in the object code.

Two special cases of the incremental relabelling require consideration. The first occurs when the new label is a cutting operator (see p. 119). Thus, the subtree rooted at p is cut from the abstract syntax tree. In this instance, p is pruned from *MR*, $affected_1$ and $affected_2$, on line 13 of Figure 6.15; the code file is then adjusted on line 14 and no further processing of p is required. The second case occurs when the original relabelling operator was the operator, φ_k, or any other cutting operator, but the new label is an inner label. In this instance, the entire subtree rooted at p is relabelled, matched and added to *RR* for later reduction; this is shown at lines 16–21 of Figure 6.15.

If neither special case is applicable, then the node p is relabelled, on line 23, and the new state calculated on line 24 using the state transition function of the BURS pattern

[3] Note that the relabelling, as defined in Section 5.3, considers as erroneous the case that $active(T, p)$ contains more than one position.

```
1: Process-structure-change :
2:   p ← choose (affected_1), affected_1 ← affected_1 \ p
3:   if active(T,p) = ∅ then          /* compare to Figure 5.16(b), p. 123 */
4:     ω ← label-of(T,p)
5:   elsif active(T,p) = {ω'}, ω ∈ Ω_k then
6:     ω ← ω'
7:   else
8:     erroneous
9:   fi
10:  if ω' ≠ T@p.relabel then
11:    if ω' = φ_k then
12:      Prune-subtree (T,p)          /* see Figure 5.16(a), p. 123 */
13:      MR ← MR ↓ p, affected_1 ← affected_1 ↓ p, affected_2 ← affected_2 ↓ p
14:      ... adjust the code file ...
15:    elsif T@p.relabel = φ_k or (T@p.relabel ∈ Cut_k and ω' ∈ Inner_k) then
16:      T@p.relabel ← ω'
17:      Transform-subtree (T,p)      /* see Figure 5.16(b), p. 123 */
18:      Match-subtree (T,p)          /* see Figure 5.18, p. 127 */
19:      top ← Expand (T,p)           /* see Figure 5.19, p. 128 */
20:      MR ← MR ↓ p, affected_1 ← affected_1 ↓ p, affected_2 ← affected_2 ↓ p
21:      RR ← [RR ∪ {top}]
22:    else
23:      T@p.relabel ← ω'
24:      T@p.state ← transition (T)
25:      top ← Expand (T,p)           /* see Figure 5.19, p. 128 */
26:      affected_2 ← affected_2 ↓ top
27:      RR ← [RR ∪ {top}]
28:    fi
29:  fi
```

Figure 6.15 Processing structural changes in the parallel incremental code generator.

matcher. A change in the state assignment at p can affect the state assignment at an ancestor, as described in Section 5.4.1 (see p. 128), and so the extent determining operator is calculated by the *Expand* function and added to RR, for subsequent reduction. Finally, the extent determining position is pruned from $affected_2$.

The positions in $affected_2$ indicate nodes at which an instance of a code determining attribute has changed value. As such, for each node $p \in affected_2$, the object code associated with the local rewrite assignment at p must be regenerated. However, as the local rewrite assignments for p and each descendant of p are unchanged, their reduction is not required. *Process-detail* in Figure 6.16(a) selects an element p of $affected_2$ and regenerates the object code for the node at p. *Reduce-node* in Figure 6.16(b) is an adaptation

```
1: Process-detail-change :
2:   p ← choose(affected_2), affected_2 ← affected_2 \ p
3:   Reduce-node (T, p, T@p.goal, location-of (T))
4:     /* location-of is defined on p. 135 */
```

(a) *Process-detail-change*

```
1: Reduce-node (T, p, goal, in out location) :
2:   rule ← ν_σ(goal) where σ = T@p.state
3:   templates ← templates-of(rule)
4:   for i = 1, ..., length-of (templates) do
5:     if t_i = (chain, goal') then
6:       Regenerate-node(T, p, goal', location)
7:     elsif t_i = (child_j, goal') then
8:       location ← location + T@p.j.size
9:     elsif t_i = (fragment, Δ) then
10:      fragment ← Expand(Δ, T@p)
11:      Put fragment at location in the code file
12:      location ← location + length (fragment)
13:    fi
14:  od
```

(b) *Reduce-node*

Figure 6.16 Processing detail changes in the parallel incremental code generator.

of *Reduce* in Figure 5.20 (see p. 129) that recreates the local rewrite assignment to p, in order to regenerate the object code induced by p, but does not visit any descendants of p.

Finally, *Parallel_Recompile* in Figure 6.12 successively selects a position p from RR, reduces the subtree and thus generates the object code for the subtree rooted at p. When RR is empty, and no edit notifications are pending in the *Update_queue*, the object code is again consistent with the program source.

6.4 Conclusions

This chapter has shown that a greedy parallel incremental code generation algorithm can be derived from the BURS-based incremental instruction selection algorithm of Chapter 5, and an analysis of the effect of the propagation of semantic information after an arbitrary subtree replacement. The internal state of the code generator is embodied in several worklists. These worklists are derived from a decomposition of the BURS-based incremental instruction selection algorithm and the sets of positions $affected_1$ and $affected_2$

that characterise the effect of the propagation of semantic information. The circumstances that cause redundant recompilation have been identified, and filtering of the update queue and the worklists eliminates any such unnecessary recompilations.

The implementation of a prototype greedy parallel incremental code generator based on the *Parallel_Recompile* algorithm in Figure 6.12 is then described in the next chapter.

Chapter 7

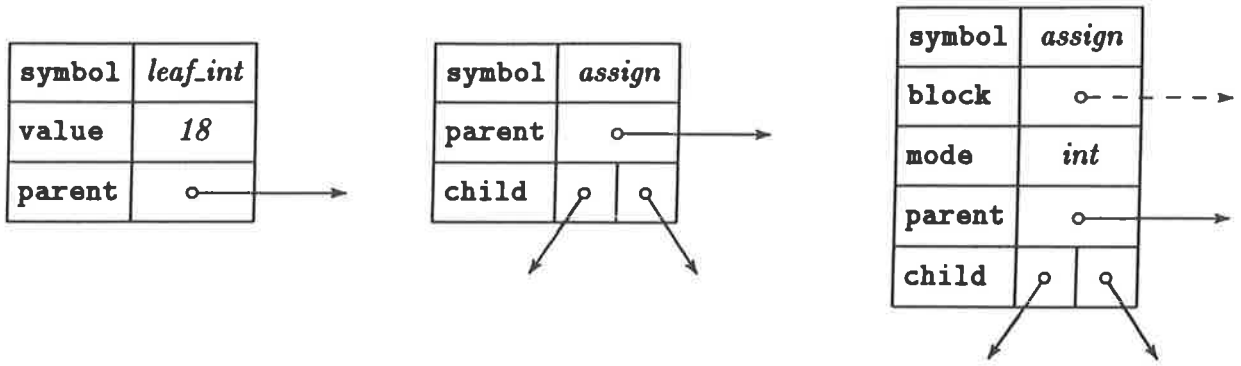
A prototype implementation

I have written a prototype implementation of both the *Parallel_Recompile* algorithm described in Chapter 6 and the MultiView distributed integrated programming environment described in Chapter 3. The prototype is implemented in the Ada programming language [ALRM83]. This chapter describes both the MultiView implementation and the implementation of the parallel incremental code generator as a MultiView view. Instrumentation of the incremental code generator enables timing the speed of recompilation. The resulting experimental data verifies that the recompilation time depends approximately linearly on the number of recompiled nodes.

The implementation described in this chapter serves only as a *proof of concept* implementation of the parallel incremental code generation strategy presented in Chapter 5 and Chapter 6. The prototype is intended only to demonstrate the practicality and soundness of this approach, rather than to operate as a practical incremental code generator. Consequently a number of simplifications have been made:

- the code generator emits lines of assembly code into an array of strings that is incrementally maintained, and
- a naive register allocation strategy is used.

A practical parallel incremental code generator based on the algorithm in Chapter 6 would have to address each of these simplifications. For instance, the additional overhead of a pre-execution assembly pass should be eliminated by directly generating object code, as exemplified by the DICE system (see Section 2.4.2).



(a) a generator instance (b) an operator application (c) semantic decoration

Figure 7.1 Basic MultiView abstract syntax tree nodes.

7.1 The MultiView prototype

The architecture of the MultiView distributed integrated programming environment was described in Section 3.2. Four basic design principles for MultiView were enumerated:

- the use of a tree-based canonical representation of the program source code,
- language independence,
- the exploitation of parallelism, and
- the provision of multiple, simultaneously updated, views of the source code.

The prototype implementation explicitly reflects each of these principles.

7.1.1 The canonical program representation

Abstract syntax trees are the canonical program representation used in MultiView. Recall the definition of an abstract syntax tree from Chapter 4 (see p. 73): an abstract syntax tree is either

- a generator instance, or
- the application of an operator to a vector of operands.

A generator instance $g \in \mathcal{G}_s$ is implemented as a record instance, as shown in Figure 7.1(a). The *symbol* field contains an integer indicating the set to which the generator belongs. The *value* field contains a representation of g; the type of which is specified as an Ada type in the language specification, as described later.

An operator application $T = \omega(t_1, \ldots, t_n)$ is also implemented as a record instance, as shown in Figure 7.1(b). An integer that represents the labelling operator ω is stored in the *symbol* field. The *child* field contains a vector of pointers to the representations of $t_1, \ldots, t_n$. The *parent* field of each operand t_i of T, that occurs in the representation of both generator instances and operator applications, points to the representation of T.

Nodes are decorated by adding fields to the record type. For example, Figure 7.1(c) shows the attribution, due to an attribute grammar, of the record that represents applications of the *assign* operator to a variable and an expression. The compiler for the language specification language (LSL) generates the appropriate Ada type declarations for the semantic decoration of abstract syntax tree nodes from the attribute grammar specification.

The storage requirements of an abstract syntax tree program representation is proportional to the number of nodes that are required to represent the program. As an indication of these storage requirements, the table in Figure 7.2 summarises the number of nodes used to represent various Ada compilation units from the MultiView implementation. The first file, `ada_parser_body.ada`, is the LALR Ada parser that is generated from a context free grammar, the second file, `view_css_body.ada`, is from the implementation of the communication subsystem, and the third file, `unit_manager_body.ada`, is from the implementation of the MultiView database. For each file, the entries of the table indicate the number of lines and bytes in the source file, the total number of abstract syntax tree nodes, the number of nodes corresponding to executable statements, the average number of nodes per line of source code and the average number of nodes per executable statement. The line of the table labelled "Total" summarises these properties for 34 Ada compilation units from the MultiView implementation. For these 11,000 lines of Ada code, approximately 160,000 abstract syntax tree nodes are required, of which 2,400 are for executable statements, yielding an average of 14 nodes for each line of source and 67 nodes for each executable statement. The final line of the table relates to `vse_body.ada`, the

Source file			Nodes		Average nodes per	
Name	Lines	Bytes	All	Statement	Line	Statement
`ada_parser_body.ada`	3712	76304	66718	544	17	122
`view_css_body.ada`	145	4643	953	12	6	79
`unit_manager_body.ada`	1265	45558	21054	400	16	52
...	...	...	...	...	...	...
Total	10789	304930	161297	2385	14	67
`vse_body.ada`	12463	501991	314494	7097	25	44

Figure 7.2 Typical sizes of abstract syntax trees.

visit-sequence evaluator generated by the language compiler from the attribute grammar in the language specification that appears, in part, in Appendix B.

This data illustrates the substantial storage requirements for abstract syntax tree representations of program source. For instance, suppose that an abstract syntax tree consists of predominately binary operator applications, of the kind shown in Figure 7.1(b). Assuming that two bytes are required for the symbol and four for each pointer, then an average of 14 bytes of storage are required for each node. In this case, the abstract syntax tree representation for `unit_manager_body.ada` would require almost 300,000 bytes of storage, whereas the textual representation requires only 45,000 bytes. Thus, the abstract syntax tree requires almost an order of magnitude more storage than the textual representation for the same piece of Ada code. Environments based on an abstract syntax tree canonical representation place considerable demands on the memory of the host computer. The functionality of the environment must justify such demands on computing resources.

The storage requirements for an abstract syntax tree can be reduced both by reducing the number of nodes used to represent a given program, and by reducing the amount of information stored at each node. The first is attained by careful design and refinement of the abstract syntax. The Ada abstract syntax, from which the trees summarised in Figure 7.2 were derived, has been naively synthesised from a context free grammar for

Ada. A careful redesign will reduce the number of nodes used to represent a given piece of Ada code. The second reduction is attained by eliminating unnecessary fields in each node. For instance, removing the *parent* field in Figure 7.1 saves four bytes of storage at each node, for a saving of 80,000 bytes in the representation of *unit_manager_body.ada*; this field is not strictly necessary because references to the parent can be kept on a stack during any traversal of the abstract syntax tree, as is the case in the DICE system [Fritzson95]. The elimination of data from the tree nodes comes at the expense of computing necessary information on demand.

Incremental code generation requires that additional data is stored at nodes of the abstract syntax tree. In the prototype implementation, this data requires an extra ten bytes of information to be stored at each node, consisting of:

- a field indicating the operator assigned to the node by the relabelling transformation of the abstract syntax tree,
- the state and goal symbols used during the incremental instruction selection algorithm (see Figure 5.17 on p. 126 and Figure 5.20 on p. 129), and
- code size and offset fields (see p. 134).

The amount of additional information required to support incremental instruction selection can be reduced by eliminating the code size and offset fields from nodes that do not induce any object code, and by exploiting the optimisations detailed in Section 5.5.

The current MultiView implementation of abstract syntax trees is particularly inefficient in its use of storage in abstract syntax tree nodes On average, the current prototype consumes approximately forty bytes for each node. This excessive use of storage is caused by two factors:

- limitations in the Ada system used to compile MultiView that preclude a number of storage optimisations, and
- compromises in the design of the node data type to allow specialisation by the different types of view.

The Ada system used to compile MultiView sometimes generates incorrect code if record fields are not aligned to 32 bit boundaries. Thus, 4 bytes are used to store the

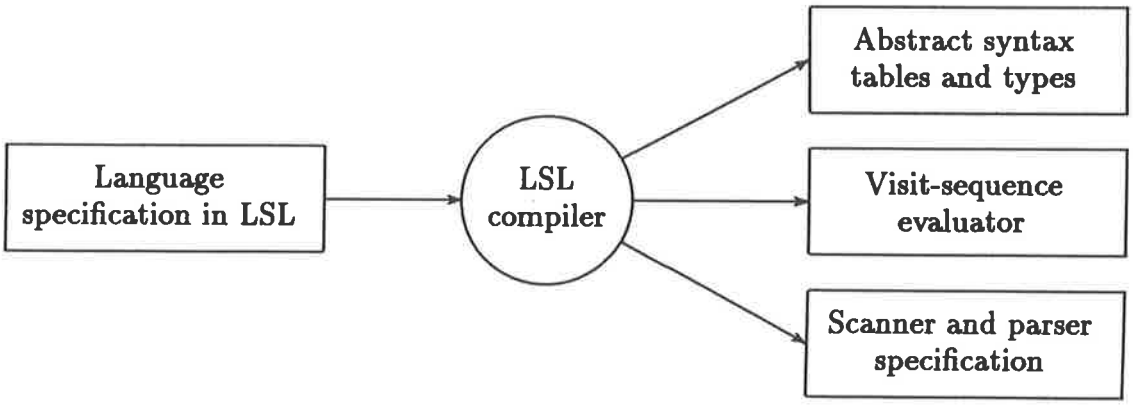


Figure 7.3 Generation of language specific components.

encoding of the operator symbol in each node. In fact, 2 bytes is adequate to represent an operator in the version of MultiView that supports the development of Ada source code.

The data type that implements abstract syntax tree nodes in the current prototype is designed to allow specialisation of the node type, such as by the definition of additional decoration to aid incremental unparsing in a textual editing view. Consequently, the symbol field that is the discriminant of both the node type and the specialising data is duplicated several times in each node.

It is essential that the storage requirements of the abstract syntax tree are reduced in any future development of the MultiView system. Such reduction can be attained by

- eliminating unnecessary record fields, such as the parent field, and
- redesigning the mechanism that allows views to specialise the abstract syntax tree node type.

7.1.2 Language independence

Language independence in the MultiView environment is facilitated by a compiler for LSL; see Section 4.6 for the earlier discussion of LSL. As depicted in Figure 7.3, the LSL compiler analyses a language specification and generates

- tables that represent the abstract syntax of the language,
- data types for implementing the canonical representation,
- a visit-sequence evaluator for the attribute grammar, and
- a scanner and parser specification.

```
type IDENTIFIER is
  regexp "[a-zA-Z](_?[a-zA-Z0-9])*";
  with IDENTIFIERS;
  represent with IDENTIFIERS.IDENTIFIER_TYPE;
end;
type INTEGER is
  regexp "[0-9]+";
  represent with INTEGER;
end;
```

Figure 7.4 Specification of the type of generator instance representations.

The tables that represent the abstract syntax and data structure declarations for generator instances are encapsulated in the `LANGUAGE` package that is emitted by the language compiler. Symbols of the abstract syntax are implemented as small integers. For each type that is declared in the language specification, a generator set is defined. Generator instances are implemented as records that are discriminated by the index of the generator set. The type of the value field for each generator set is defined in the type declaration of the language specification. For example, Figure 7.4 illustrates the definition of the data types used in the value fields of the Ada record for the generator sets `IDENTIFIER` and `INTEGER`. In the first instance, the data field is of type `IDENTIFIER_TYPE`, from the Ada package `IDENTIFIERS`. In the second instance, the type of the data field is the standard Ada `INTEGER` type.

The algorithm from [Reps89a, Chapter 12] is used to synthesise a visit-sequence attribute evaluator from the attribute grammar specification. The attribute definition functions (see p. 79 in Chapter 4) are compiled directly into Ada, as are the plans used by the visit-sequence evaluator. The use of a visit-sequence evaluator requires that the attribute grammar belongs to the class of ordered attribute grammars [Kastens80].

A scanner and parser specification are generated from the concrete syntax specification. Furthermore, for each generator set, the specified regular expression is used to construct a rule in the scanner specification to recognise generator instances (see Figure 4.3, p. 84). The tree constructors associated with each production are compiled into actions in the generated parser specification that construct an abstract syntax tree when

the program source is parsed. An LALR parser generator [Taback88] constructs a bottom-up parser from the generated parser specification, and so the context-free grammar must belong to the class of LALR(0) grammars.

An noted in Chapter 4, Appendix B contains an extract of the LSL specification for the small language used to test the prototype incremental code generator.

7.1.3 Parallelism

The MultiView architecture is designed to allow the exploitation of any parallelism of the underlying hardware. The implementation exploits parallelism in two ways:

- the implementation of the MultiView database and views as concurrently executing Unix processes, and
- the implementation of the individual processes as multitasking Ada programs.

Concurrency between the database and views

The database and views are implemented as Unix processes that communicate via message passing. The communication subsystem (CSS) implements a message passing system over the Unix interprocess communication primitives.

The CSS is implemented in two layers. The top layer is the generic layer, in which message handling policies are implemented. For example, this layer includes the signalling of a timeout if a reply does not arrive within a predetermined interval, and the selective broadcasting of a single database message to a collection of views. The bottom layer is the transport layer, which manages the encoding and decoding of messages, and connections between the database and the views.

The *stream transport layer* is a transport layer implementation that manages connections that are implemented using Internet domain Unix stream sockets [Leffler83]. This connection-oriented interprocess communication mechanism provides a bidirectional byte stream between the database and views. Figure 7.5 illustrates the encoding of a query into a text string for transmission by the stream transport layer via a Unix stream socket. This transport layer allows the database and views to execute on different hosts on a local

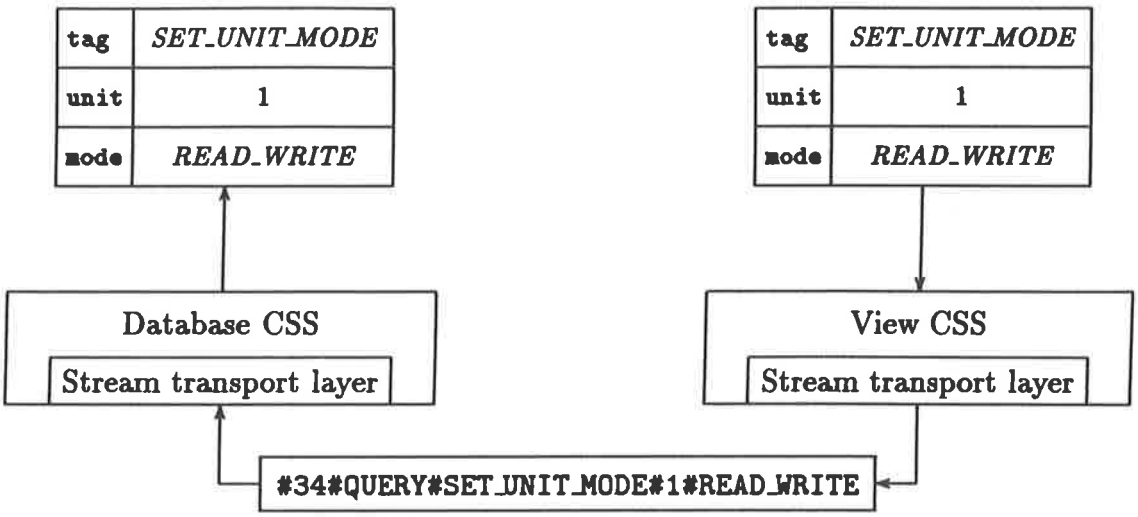


Figure 7.5 Transmission of a query to the database.

area network. Thus, parallelism between multiple Unix hosts on a local area network is exploited.

Protocol-specific components of both the generic layer and the stream transport layer are generated from the high-level protocol specification described in Section 3.2 (see p. 63). Figure 7.6 depicts the compilation of a protocol specification by the protocol compiler into a package containing the message data structure declarations and the executable code for encoding messages in stream transport layer.

The transport layer and its interfacing with the generic layer are designed to allow the substitution of different transport layers that exhibit different properties. For example, the textual message encoding used by the stream transport layer is useful for debugging the MultiView implementation. However, the translation of messages into text and the corresponding decoding is expensive. The implementation and evaluation of alternative transport layers is an avenue for future research.

Concurrency within the database and views

The database and views are implemented as multitasking Ada programs. Recall the structure of a MultiView view illustrated in Figure 3.8 on p. 65. The CSS, the view task, the dispatcher and the graphical user interface are each implemented as separate Ada

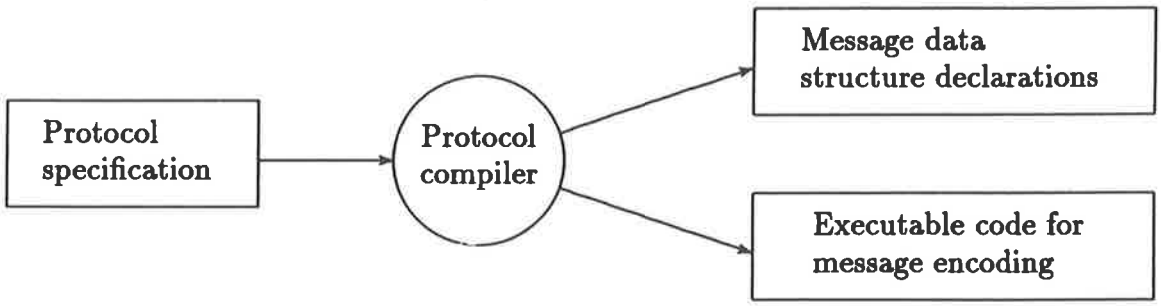


Figure 7.6 Generation of CSS components from the protocol specifications.

tasks. Furthermore, the CSS is implemented using several concurrently executing Ada tasks.

The exploitation of any parallelism provided by the underlying hardware by a multitasking Ada program is limited by the particular Ada run time system. A run time system, such as that described in [Dewar94], that maps Ada tasks to operating system threads allows MultiView to exploit the parallelism inherent in a shared memory multiprocessor. However, on a uniprocessor, or if the Ada run time system does not map tasks onto parallel processors, then a single processor is shared between all tasks, and so no true concurrency is possible within the database or a view.

7.1.4 Multiple views

A typical MultiView view enables the editing or browsing of a single compilation unit. Views are constructed from a number of components, enumerated in Figure 3.8 on p. 65.

- a communication subsystem (CSS),
- a query handler,
- a dispatcher,
- a cached copy of an abstract syntax tree from the database,
- a view task specific to the type of view, and
- an optional user interface.

The view task and the user interface must be implemented by hand, in full, for each new type of view. The implementation of the other view components is aided by the

```
1: view  TEXTVIEW (AST_CACHE) is
2:   broadcast
3:     when  REPLACE_SUBTREE ⇒
4:       {S1}
5:     when  DELETE_UNIT ⇒
6:       {S2}
7:     ...
8: end  TEXTVIEW;
```

Figure 7.7 Extract of the dispatcher specification for TextView.

protocol compiler. In particular, several protocol-specific and view-specific packages are generated from specifications of their behaviour.

The dispatcher is generated from a specification that is expressed in terms of the high-level protocol specification language (see p. 63). The outline of a typical dispatcher specification in this language appears in Figure 7.7. A more complete example is given in Appendix D. Statement templates, such as S_1 on line 4 of Figure 7.7, are translated by the protocol compiler into Ada statements to be executed on the receipt of the indicated message. For example, the code constructed from the template S_1 is associated with the receipt of an EDIT message, indicating that it is to be executed whenever an EDIT message is delivered to the view. A simple macro expansion mechanism allows accessing of message fields from within a statement templates; this macro facility is apparent in the complete dispatcher specification in Appendix D.

Behaviour which is common to a number of views may be encapsulated into a view component. For example, the AST_CACHE component implements an abstract syntax tree cache. This component may then be inherited into the dispatcher, as is the case in Figure 7.7. The protocol compiler weaves the dispatcher specification, along with the specifications of any inherited view components, into executable code that is incorporated into the view.

A similar mechanism to the dispatcher specification allows actions to be associated with queries in the query handler. For example, in the AST_CACHE component

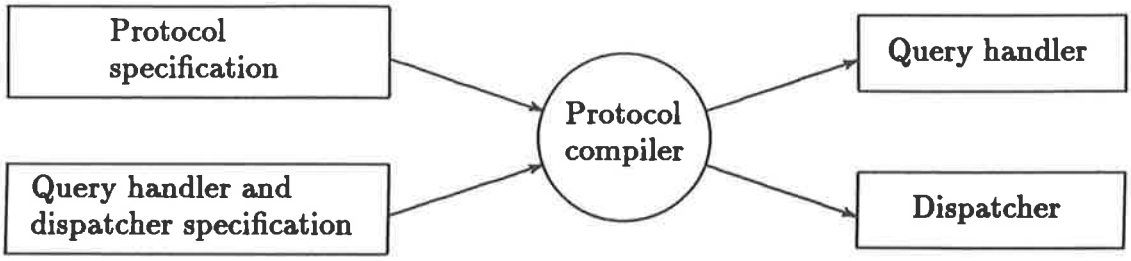


Figure 7.8 Generation of view components from the message specifications.

statements that update the cached abstract syntax tree are associated with the REPLACE_SUBTREE. Thus, the cache is automatically updated whenever the view sends a REPLACE_SUBTREE message to the database.

The generation of view components by the protocol compiler is depicted in Figure 7.8. In summary, the protocol compiler supports the implementation of views by

- generating the protocol-specific components of the CSS, as shown in Figure 7.6,
- generating the dispatcher and query handler from the protocol specification and view-specific query handler and dispatcher specifications, as shown in Figure 7.7 and Figure 7.8, and
- providing a mechanism for inheriting behaviour, such as the updating of the cache on receipt of subtree replacement messages.

The protocol specification languages, its compiler and the structure of the CSS are described at length in [McCarthy94]; in particular, the details of the low-level protocol, messages classes and the language features that combine view components are all described in this paper.

The language compiler is also designed to provide explicit assistance to a view implementor. Extensions to LSL can be constructed for particular view types. For example, LSL and its compiler were extended to support the specification of textual unparsing schema for TextView [Read93]. In the TextView extension, each operator of the abstract syntax is associated with a non-empty set of unparsing schema. The TextView display engine uses these unparsing schema to derive a textual representation of abstract syntax trees.

Appendix D contains the dispatcher specification for the incremental code generator view in a form suitable for processing by the protocol compiler.

7.2 Architecture independence

The architecture-dependent components of the incremental code generator are derived from the abstract syntax specification and an architecture description. The architecture description consists of two parts:

- the specification of the relabelling transformation, and
- the specification of the bottom-up rewrite system.

The specification of the relabelling of the `int_const` operator is illustrated in Figure 7.9(a). First, the cutting operators `smallint` and `largeint` are declared. Next, the relabelling transformation for the `int_const` operator is specified; the `otherwise` keyword denotes a expression that is true if and only if each of the other guards is false. Anonymous, boolean-valued local attributes of the operator `int_const`, dubbed b_1 and b_2 in the following discussion, are implicitly defined. The dependency and attribute definition functions of b_1 and b_2 are

$$dependencies_{\texttt{int\_const}}(b_1) =< 0, \texttt{value} >$$

$$dependencies_{\texttt{int\_const}}(b_2) =< 0, b_1 >$$

$$value_{\texttt{int\_const},b_1,0}(x) = x < 2^{13}$$

$$value_{\texttt{int\_const},b_2,0}(x) = not(x)$$

Thus, if the `value` attribute of a node labelled with *int_const* is less than 2^{13}, then the node is relabelled with `smallint`; otherwise, it is relabelled with largeint. Furthermore, the `value` attribute of the operator `int_const` is inherited by the new cutting operators, due to the "{`value`}" construct.

Four rewrite rules and associated code templates are illustrated in Figure 7.9(b). The first two rules are required so that the rewrite system is in normal form (see p. 111). The next rule reduces subtrees labelled with `assign` to the symbol `rvalue`, with cost 1, when the left and right operands have been reduced to `register_var.1` and `smallint.1`, respectively. The associated code template expands to a fragment of SPARC assembly

```
smallint[cut];
largeint[cut];

int_const =>
  value < 2**13 =>
    smallint {value}
  otherwise =>
    largeint {value}
```

(a) a relabelling specification

```
smallint.1 -> smallint, 0
largeint.1 -> largeint, 0

rvalue -> assign(register_var.1, smallint.1), 1
  // add %g0,$2.value,%i$1.register

rvalue -> assign(register_var.1, largeint.1), 2
  // sethi %hi($2.value),%l0
  // or %l0,%lo($2.value),%i$1.register
```

(b) four rewrite rules

Figure 7.9 Extract from an architecture description.

code after the substitution of the `value` attribute of the second operand and the `register` attribute of the first, as described in Section 5.4.1.[1] The final rewrite rules matches when the right operand reduces to `largeint.1`.

The architecture specification is compiled into a collection of Ada procedures and functions that implement the relabelling transformation, the BURS pattern matcher, and the list of code templates associated with each rewrite rule. These functions are encapsulated in the BURS package that is emitted by the protocol compiler. The relabelling transformation described in Section 5.3 is realised as an Ada procedure that relabels the abstract syntax tree during a top-down traversal. The bottom-up pattern matching automaton is synthesised using Proebsting's worklist driven BURS pattern matcher generation algorithm [Proebsting92]. The resulting transition function for the BURS pattern matching automaton is encoded in an Ada function. A representation of the code templates for each rewrite rule is returned by a function that is encapsulated in the BURS package.

The BURS compiler also emits diagnostic procedures. For example, a procedure that prints a textual representation of the BURS state assignment of a given node has proven invaluable during the implementation of the incremental code generator view.

Figure 7.10 illustrates the generation of the BURS functions and templates from the architecture description by the BURS compiler. A representation of the abstract syntax

[1] In the SPARC architecture, the `%g0` register always has value zero; thus, the instruction "`add %g0,n,%l0`" loads the constant n into register `%l0`.

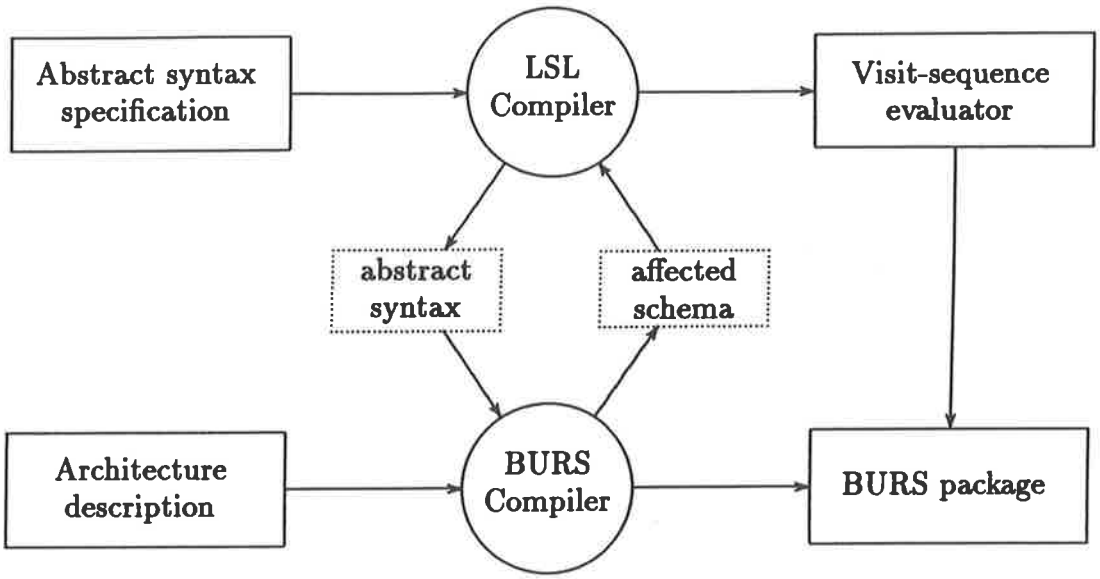


Figure 7.10 Generation of the code generator from formal specifications.

and semantic decorations is written by the LSL compiler and read by the BURS compiler. Furthermore, the BURS compiler emits dependency information required for the inference of $affected_1$ and $affected_2$ during incremental semantic analysis. The LSL compiler uses this dependency information in order to insert code into the visit-sequence evaluator so that, when semantically significant and code determining attribute instances change value, the appropriate positions are added to $affected_1$ and $affected_2$.

Generation of the architecture-dependent components of the incremental code generator is a three-stage process:

- the language compiler generates a representation of the abstract syntax suitable for the BURS compiler,
- the BURS compiler generates the BURS package and an encoding of the semantically significant and code determining attributes, and
- the language compiler generates the adapted visit-sequence evaluator that infers $affected_1$ and $affected_2$ during incremental semantic analysis.

Appendix C includes an extract of a SPARC architecture description for the language specified, in part, in Appendix B. The results reported in Section 7.4 are for an incremental code generator derived from this specification.

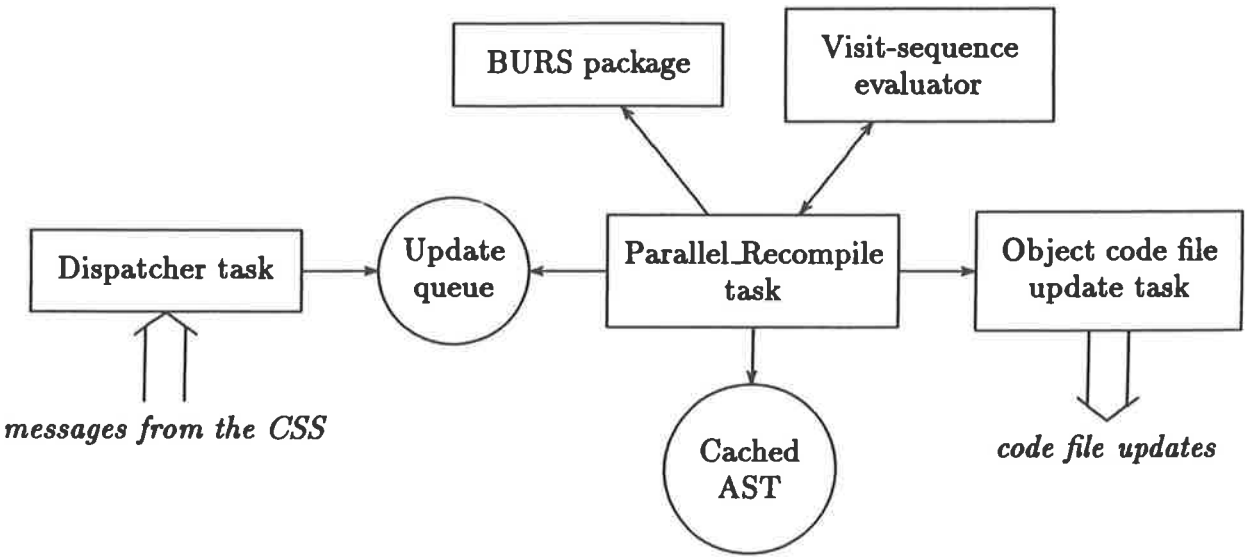


Figure 7.11 Structure of the code generator view.

7.3 The incremental code generator

The prototype implementation of the parallel incremental code generation algorithm of Chapter 6 is implemented as a MultiView view. Figure 7.11 illustrates the internal structure of this view.[2]

Messages received from the MultiView database are decoded by the CSS and passed to the dispatcher task. The dispatcher constructs update notifications from program edit notification messages and places them on the update queue. The update queue is a first-in-first-out queue that has been modified to filter redundant updates, as detailed in Section 6.3 (see p. 163).

Parallel_Recompile is a faithful implementation of the *Parallel_Recompile* algorithm detailed in Figure 6.12 on p. 167, as an Ada task. It greedily removes update notifications from the Update queue, recompiles the affected parts of the program source, and outputs code file updates.

The prototype code file implementation stores lines of assembly code. Each line corresponds to a single machine instruction. Code file updates consist of a destination address

[2] Compare this figure to Figure 6.3 (see p. 151). In Figure 7.11, the CSS is omitted, but the BURS package and the visit-sequence evaluator are included.

and a text string containing the assembly code that represents the machine instruction that is to be stored at that address. Two variant implementations of the code file exist:

- a variant in which the code file executes as a concurrent task that updates a graphical display of the generated code, and
- a simpler variant that merely stores the generated assembly code in an array.

The first variant is used for testing and demonstration of the parallel incremental code generator; the second is used for running the timing trials detailed in Section 7.4.

The prototype implementation of the incremental instruction selector requires several extra pieces of information to be added to each node of the abstract syntax tree:

- the relabelling operator assigned during transformation,
- the state assigned by the BURS pattern matching automaton,
- the code size and code offset attributes (see p. 134), and
- the goal symbol inferred during reduction.

Given the large number of nodes required for an abstract syntax tree program representation, the allocation of storage for each piece of information will add considerably to the storage requirements of the environment. However, it is possible to reduce the storage requirements of some of this data, and, in fact, eliminate some fields altogether.

The relabelling operator can be subsumed by the symbol field. Suppose that the relabelling transformation is such that each relabelling operator ω' appears only in relabelling of nodes labelled with the operator ω. If a node n is initially labelled with ω, and the desired relabelling operator is ω', then the symbol field n can be directly set to ω' without any loss of information.

Storage of the state assignment is not necessary at all nodes of the abstract syntax tree. In particular, it can always be eliminated for subtrees that do not contain executable statements. Furthermore, if a coarser granularity of recompilation is acceptable, then the state assignments need not be stored at each node. Instead, the state assignments of particular subtrees can be stored in temporary memory during recompilation and discarded when recompilation is complete. Similarly, the persistent storage of the code size and offset attribution within particular subtrees of the abstract syntax tree can be eliminated.

The goal symbol for any node can be inferred in a walk from the root of the abstract syntax tree to that node. In fact, in the case of the prototype implementation, this walk is performed when resolving tree positions into node references. Thus, during resolution of a tree position p, a vector can be constructed that contains the goal symbols for each node between p and the root of the abstract syntax tree. The stored goal symbol is used in the *Recompile* algorithm of Figure 5.25 on p. 137 only in the call to *Reduce*. If the goal is obtained from the vector constructed during resolution of the tree position, then persistent storage of the goal symbol at nodes of the abstract syntax tree is obviated.

Further investigation and experimentation of these strategies for reducing the memory requirements of the BURS-based parallel incremental code generator is a goal for future research.

7.4 Timing trials

Instrumenting the implementation of the *Parallel_Recompile* algorithm enables measurement of the speed of incremental recompilation. To facilitate the gathering of data, a new message type is introduced into MultiView that instructs the incremental code generator to perform a timing trial. This `RECOMPILE_TRIAL` trial message includes two fields:

- a tree position, and
- a trial duration.

On receipt of a trial message the incremental code generator initiates a timing trial. Such a trial consists of repeatedly relabelling the indicated subtree of the cached abstract syntax tree, matching the subtree with the BURS pattern matching automaton, and then generating assembly code from the matched subtree. The iteration terminates when the elapsed time exceeds the trial duration. The recompilation time is then inferred by dividing the elapsed time by the number of iterations.

The recompilation time inferred by this trial measures the speed of recompilation in the prototype implementation of the BURS-based incremental code generator. No incremental semantic analysis or abstract syntax tree updating is performed during a

Size		Results		Rate (per second)	
Nodes	Instructions	Iterations	Average Time	Nodes	Instructions
67	28	2031	0.012268	5443	2274
133	56	1021	0.024475	5431	2287
199	84	671	0.037292	5341	2254
265	112	486	0.051453	5151	2177
331	140	379	0.066040	5017	2122
397	168	307	0.081604	4875	2063
463	196	255	0.098145	4722	1999
529	224	217	0.115295	4591	1944
595	252	189	0.132874	4498	1905
661	280	164	0.153137	4336	1836
727	308	143	0.175171	4158	1761
793	336	128	0.195496	4060	1720
859	364	115	0.218384	3951	1674
925	392	105	0.239563	3885	1646
991	420	95	0.264099	3765	1596
1057	448	87	0.288147	3678	1559
1123	476	81	0.311340	3638	1542
1189	504	75	0.334656	3567	1512
1255	532	69	0.366333	3463	1468
1321	560	65	0.390259	3434	1456

Figure 7.12 Results from a series of trials of duration 25 seconds.

trial. The effect of time slicing between concurrent tasks by the Ada run time system is reduced by performing the trial for a fixed duration rather than a fixed number of iterations.

No attempt has yet been made to optimise the implementation of *Parallel_Recompile*. In fact, there is considerable scope for improvement, such as that offered by the optimisation noted on p. 133.

The results of a series of trials are tabulated in Figure 7.12. For the trials, the code generator view was executing on a 55MHz HyperSPARC processor. No paging occurred during any of the trials, and the code generator was the only active process on the system

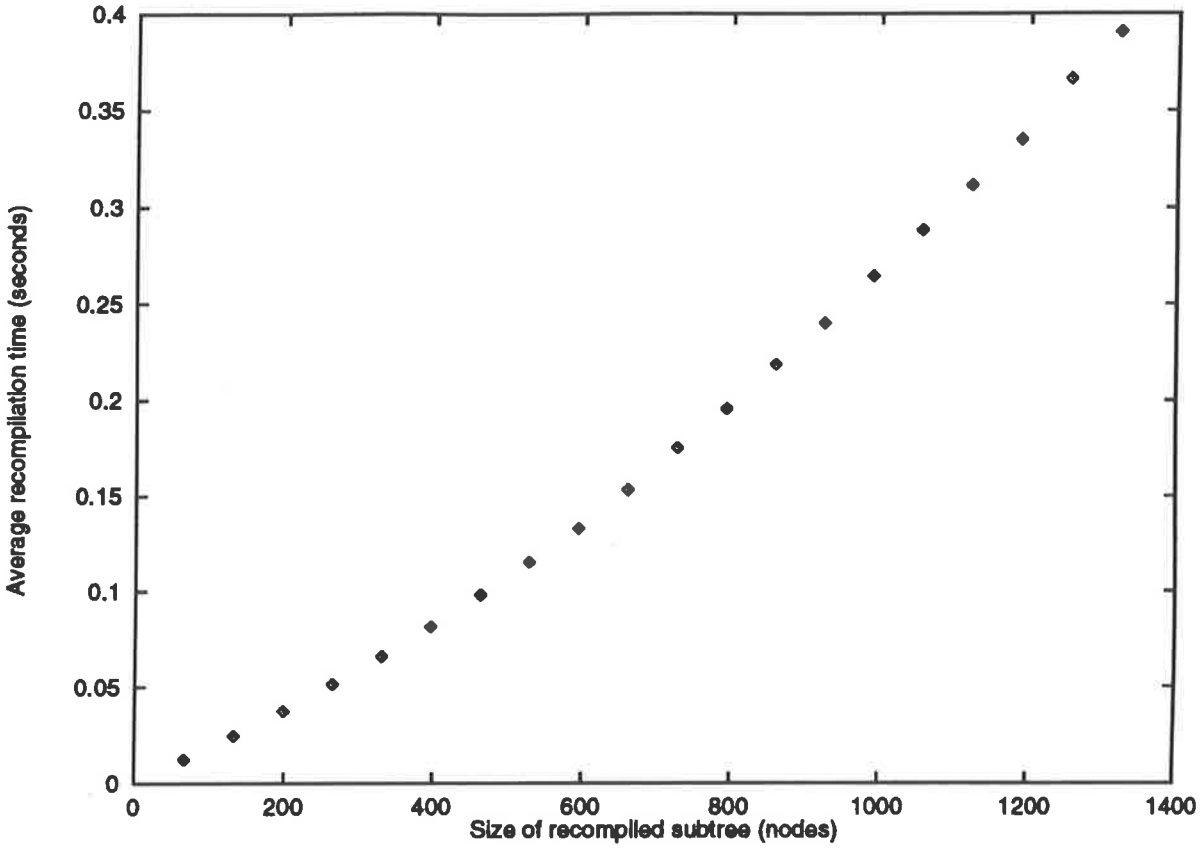


Figure 7.13 Average recompilation time plotted against the size of recompilation.

during the experiment. The source program consisted of a block that was replicated twenty times. Each twenty-five second trial recompiled a subset of these blocks.

The trial summarised in the first entry in Figure 7.12 recompiled a single block; the trial summarised in the last entry recompiled the entire twenty blocks. The "Nodes" column in Figure 7.12 indicates the number of nodes recompiled on each iteration in the trial. The "Instructions" column details the number of assembly language instructions generated during a single iteration of the trial. The "Results" columns present the number of iterations and the average time of a single iteration. Finally, the "Rates" columns indicate the average number of nodes recompiled per second during the trial, and the average number of instructions emitted per second during the trial.

The graph in Figure 7.13 plots the average recompilation time over the trial against the size of the recompiled subtree for the trials detailed in Figure 7.12. The linear relationship between the size of the extent of recompilation and the recompilation time is apparent

from the graph. Thus, the incremental code generator recompiles the program, after the replacement of a subtree in the abstract syntax tree, in time proportional to the size of the new subtree.

The execution cost of incremental code generation must be considered in the context of the performance of the entire environment. Two metrics are significant:

- the *response time* of the editor, and
- the *preparation time* when execution is requested.

Interleaved execution of a program editor and a greedy incremental code generator adversely affects the response time of the editor. After an edit, the code generator immediately recompiles the affected parts of the program. Further editing is prevented until recompilation is completed. The prototype parallel greedy incremental code generator view for MultiView has no such effect on the response time of MultiView editing views. Instead, in the MultiView environment, the response time of the editor is affected by the overhead of communicating with the database. Quantifying, and reducing, this overhead is a future research direction for the MultiView project. Editing can continue before recompilation is completed.

The preparation time of the prototype incremental code generator is poor as a result of the decision to incrementally generate assembly code, rather than machine code. An assembly and link phase is required before execution is possible (see p. 174).

7.5 Conclusions

The implementation of the prototype incremental code generator has demonstrated the integration of a greedy parallel incremental code generator into a distributed integrated programming environment.

The prototype incremental code generator exhibits subexpression-level granularity. The extent of recompilation is the subtree rooted by the extent determining node of the abstract syntax tree after an edit has occurred (see p. 125). The experimental results indicate that recompilation takes time proportional to the number of nodes of the abstract syntax tree that are recompiled, that is the number of nodes in the extent of recompilation. That is, the recompilation takes time proportional to the *size* of the edit (see p. 125).

Together, the MultiView prototype and the incremental code generator view form a language-independent integrated programming environment that includes a retargetable incremental code generator. An instance of this environment for a particular programming language is generated from a formal specification of the language and the target computer architecture on which the programs in the language are to be executed.

Future research will focus on the use of the incremental code generator to implement run-time views such as debuggers and graphical displays of executing programs. The fine-grained incremental code generator may be used to efficiently insert fragments of code into the object file to monitor assertions or update a display.

Chapter 8

Conclusions and future work

8.1 Conclusions

This thesis has shown that

- a retargetable incremental instruction selection algorithm can be systematically derived from a non-incremental algorithm within the framework of a precise model of the underlying program representation,
- an efficient parallel incremental code generation algorithm can be derived from the incremental instruction selection algorithm, and
- a parallel incremental code generator can be integrated into a distributed integrated programming environment.

An algebraic model of abstract syntax was described in Chapter 4. This model considers abstract syntax trees as words freely generated from the abstract syntax. It is equipped with a notion of program semantics based on attribute grammars and a notion of program editing based on subtree replacement. It then serves two purposes in the subsequent chapters:

- the foundation for the derivation of the BURS-based instruction selection algorithm and the greedy parallel incremental code generation algorithm, and
- the underlying semantics of the language specification formalism for the MultiView environment.

Bottom-up rewrite system (BURS) based instruction selection was chosen from the approaches to retargetable instruction selection surveyed in Chapter 5 as the most suitable basis for an incremental instruction selection algorithm. The BURS-based incremental instruction selection algorithm was synthesised from:

- an analysis of the effect on the BURS pattern matcher state assignment of the replacement of a subtree in the subject tree, and

- a semantically-based transformation from program representations based on abstract syntax trees to subject trees suitable for BURS pattern matching.

The resulting BURS-based incremental instruction selection algorithm recompiles a program after the replacement of a subtree that corresponds to executable code. However, neither recompilation after an arbitrary subtree replacement, nor the integration of a BURS-based incremental code generator into an integrated programming environment is considered in this chapter.

In Chapter 6, a greedy parallel incremental code generation algorithm is derived from the BURS-based incremental instruction selection algorithm. The code generator successively fetches source update notifications from an update queue and immediately recompiles the affected portions of the abstract syntax tree. The internal state of the code generator is embodied in several worklists. In turn, the worklists are based on sets of positions:

- sets that characterise the state of BURS-based incremental instruction selection during recompilation, and
- sets that characterise the effect of the propagation of static semantic information after an arbitrary subtree replacement.

An analysis of the relationships between these sets allows filtering of the worklists, in order to eliminate redundant recompilation in the greedy parallel incremental code generation algorithm.

The integration of an incremental code generator, based on the algorithm of Chapter 6, is demonstrated by a prototype implementation described in Chapter 7. The incremental code generator is realised as a view in the MultiView distributed integrated programming environment. Recompilation occurs in parallel with program editing. Experimental results also presented in this chapter indicate that recompilation time increases approximately linearly with the size of the recompiled subtree.

8.2 Future work

8.2.1 Further refinement of the algorithms

Several improvements to the basic BURS-based incremental recompilation algorithm were noted in Chapter 5:

- further elimination of BURS pattern matcher state reassignment in certain subtree of an updated subject tree (see p. 113), and
- coalescence of the relabelling and pattern matching traversals (see p. 133).

An adaptation of the *Recompile* algorithm of Figure 5.25 that encompasses these optimisations would exhibit speedups on the current one. The first optimisation reduces the number of nodes that must be visited in order to regenerate object code, and the second eliminates a complete traversal of the modified abstract syntax tree.

The considerable storage requirements of program representations based on abstract syntax trees were noted in Chapter 7. The intermediate information stored in the abstract syntax tree exacerbates the demand for memory of the integrated programming environment. However, as noted in Section 7.3, some of the intermediate information can be eliminated and the omitted information generated on demand. For example, the goal symbol of a node can be calculated during a walk from the root of the abstract syntax tree to that node; this walk is already performed in the prototype implementation when resolving tree positions to nodes.

Other intermediate information, such as the code size, code offset and state information could be eliminated at particular nodes. For example, suppose that this information is stored only at nodes that correspond to executable statements. Then the recompilation algorithm must be adapted to regenerate the object code for the entire statement if any subtree of the statement node is replaced, effectively trading the storage requirements against the granularity of recompilation.

8.2.2 Extensions of the algorithm

While the BURS-based incremental instruction selection algorithm performs locally optimal instruction selection, no consideration has been given to either register allocation or optimisation. The work of Bivens, described in Section 2.4.6, showed that good-quality register allocation can be performed in conjunction with fine-grained incremental recompilation. Pollock's work, described in Section 2.4.7, detailed a scheme for the incremental implementation of certain local and global optimisations.

Reconciliation of the BURS-based incremental instruction selector with Bivens' incremental register allocation algorithm and Pollock's incremental optimisation algorithm would result in an algorithm that incrementally generates very high quality object code. Each of these algorithms requires the derivation and incremental maintenance of additional intermediate information. In the case of incremental register allocation, interference graphs and interval graphs are required; for the incremental optimiser, Pollock's MFAD representation is necessary (see Figure 2.15).

The interference and interval graphs required for register allocation can be derived from the object code emitted by the instruction selection. Indeed, the register demand and the boundaries of the basic blocks will not be known until code has been generated. Bivens' algorithm could probably be implemented in a separate pass after incremental instruction selection has completed.

The efficiency of the prototype incremental instruction selector can be considerably improved. The encoding of the BURS pattern matcher and the templates is both large and inefficient to access. It is possible to construct a much better encoding of the BURS tables. For example, [Fraser91b] describes an implementation of a BURS-based code generator in which the tables are compiled into a combination of hard code and data. In this implementation, the VAX code generator is reported to have occupied 21.4Kbytes and identified locally optimal assembly code in 50 VAX instructions per node. Given that the underlying algorithms are similar, this result indicates that there is considerable scope for improving the MultiView incremental code generator view.

8.2.3 Applications of fine-grained incremental compilation

The ability of a fine-grained incremental code generator to efficiently insert and remove fragments of code from the object code file may be exploited in the development of new tools in a programming environment.

The incremental code generator of the DICE environment had been used to implement a semi-automatic debugging technique [Fritzson92]. Likewise, the prototype incremental code generator described in Chapter 7 can be used to explore sophisticated debugging techniques.

The code generator may also be used to implement the monitoring and display of an executing program. In the MultiView environment, the existing views enable the browsing and manipulation of a static representation of the program source. Run-time views would display information concerning the dynamic state of the program as it executes. The ability of incremental code generator to efficiently insert and remove fragments of the object code for the executable program will facilitate the implementation of these views. For example, a dynamic view of the run time stack of selected procedures may be realised by inserting the appropriate instructions at the entry points of these procedures. More ambitiously, a similar mechanism might be used for program animation.

8.2.4 The MultiView system

The prototype implementation of the MultiView system has demonstrated the practicality of the client-server architecture for an integrated programming environment. Several future research directions for MultiView are apparent:

- refinement of the communication mechanism,
- the implementation of run-time views, and
- more sophisticated incremental semantic analysis.

The current implementation of the transport layer of the MultiView communication subsystem (CSS) encodes messages as text. The performance of the CSS would be enhanced by a more efficient encoding and message transport mechanism. Speeding up the

CSS would be reflected in improved response time for users of the MultiView environment. An improved CSS would result from the development and systematic evaluation of alternative transport layers.

Incremental semantic analysis in MultiView is performed by a visit-sequence evaluator. While easy to implement, this technique is limited. In particular, the implementation of inter-module semantic analysis is difficult. Several techniques that address this problem are mentioned in Section 2.3; the suitability of their application to the BURS-based incremental instruction selection algorithm is briefly addressed at the end of Section 5.3.

Appendix A

Glossary of Symbols

Symbol	Page	Meaning
$\mathcal{S}$	72	The set of sort symbols.
$\Omega = \{\Omega_n\}$	72	The set of operator symbols.
α	72	An Ω structure on $\mathcal{S}$ and Ω.
$\Sigma = (\mathcal{S}, \alpha)$	72	A operator scheme over $\mathcal{S}$.
$\mathcal{G} = \{\mathcal{G}_s\}$	73	An $\mathcal{S}$ indexed family of generator sets.
$\mathcal{L} = (\Sigma, \mathcal{G}, \xi)$	73	An abstract syntax.
$\Gamma_s(\mathcal{L})$	73	The set of abstract syntax trees of type $s \in \mathcal{S}$ over the abstract syntax $\mathcal{L}$.
$\Gamma(\mathcal{L})$	73	The set of all abstract syntax trees over the abstract syntax $\mathcal{L}$.
arity-of(T)	73	The arity of the abstract syntax tree, T.
$dom : \Gamma(\mathcal{L}) \rightarrow \mathbb{D}$	76	The domain function from abstract syntax trees to tree domains.
$p \ anc \ q$	76	The tree position p is an ancestor of the tree position q.
$p \perp q$	76	p and q are independent tree positions.
label-of(T)	76	The label of the root of the abstract syntax tree T.
$\Gamma(\mathcal{L}, \mathcal{X})$	78	The set of tree patterns, generated by $\mathcal{L}$, with variables from $\mathcal{X}$.
$var(\mathcal{P})$	78	The positions labelled by variables in the tree pattern $\mathcal{P}$.
type-of	79	The type function for attribute symbols.
$dependencies_\omega$	79	The dependency function for the output attributes of ω.
$value_{\omega,a,k}$	79	The value function for the output attribute instance $< k, a >$ of ω.

$eval(T, n, a)$	80	The value of the instance of the attribute a, at position n, in the abstract syntax tree T.
$T_{p \leftarrow t}$	82	The abstract syntax tree resulting from replacing the subtree of T at p with the subtree t.
Cut_n	119	Cutting operators of arity n.
$Inner_n$	119	Non-cutting operators of arity n.
$pairs_\omega$	119	Semantic transformation tuples for the operator ω
$\mathcal{P}_\omega$	155	The semantically significant predicates of ω.
$affected_1$	156	The set of nodes that may require relabelling, due to the propagation of semantic information after a subtree replacement.
$affected_2$	159	The set of nodes for which code must be regenerated, due to the propagation of semantic information after a subtree replacement.
$[P]$	168	The normalisation of the set of positions, P.
$P \downarrow p$	168	The pruning of the set of positions P at p.

Appendix B

Extract from a language specification

This appendix contains extracts from the specification of a small language using the MultiView language specification language (LSL).

```
-------------------- extracts of toy.lsl -------------------------------

-- Specification of the language TOY for the MultiView environment

language TOY;
start PROGRAM;                              -- the start sort

-------------------- generators ----------------------------------------

-- identifiers
oper id:  placeholder -> ID;
type ID is
  regexp "[a-zA-Z](_?[a-zA-Z0-9])*";
  complete with id;                         -- operator for unexpanded ID
  with IDENTIFIERS;                         -- Ada compilation context
  represent with IDENTS.IDENT_TYPE;         -- data type for ID instances
  print with IMAGE;                         -- convert to text
  scan with "INTERN (#)";                   -- convert from text
end;

-- integers
oper int:  placeholder -> INT;
type INT is
  regexp "[0-9]+";
  complete with int;                        -- operator for unexpanded INT
  represent with INTEGER;                   -- data type for INT instances
  compatible with INTEGER;                  -- allow INTEGER operations
  print with "INTEGER'IMAGE (#)";           -- convert to text
  scan with "INTEGER'VALUE (#)";            -- convert from text
end;
```

```
------------------- sorts and operators -------------------------------

-- programs are sequences of expressions.

sort PROGRAM;
oper program:      EXP_SEQ -> PROGRAM;

<program> ::= <exp_seq>        { program (<exp_seq>) };

-- If a sort is deemed to be a list sort, then it must include two
-- operators:  a nullary operator and a right-recursive binary operator.
-- The binary operator is specified as "list of <sort>".

-- lists of expressions

sort EXP_SEQ is
  list;                                  -- EXP_SEQ is a list sort
end;
-- EXP_SEQ is a list sort

oper exp_seq_nil:  placeholder  -> EXP_SEQ;
oper exp_seq:           list of EXP  -> EXP_SEQ;

<exp_seq> ::=
  <exp>                         { exp_seq (<exp>, .)  }
| <exp> ';' <exp_seq>           { exp_seq (<exp>, <exp_seq>) };

-- expressions

sort EXP is
  complete with empty;                    -- operator for unexpanded EXP
end;

oper empty: placeholder -> EXP;
oper break: EXP EXP     -> EXP;
oper assign:VAR EXP     -> EXP;
oper eq:    EXP EXP     -> EXP;
oper neq:   EXP EXP     -> EXP;
oper le:    EXP EXP     -> EXP;
oper ge:    EXP EXP     -> EXP;
oper lt:    EXP EXP     -> EXP;
oper gt:    EXP EXP     -> EXP;
oper sum:   EXP EXP     -> EXP;
oper diff:  EXP EXP     -> EXP;
oper prod:  EXP EXP     -> EXP;
oper quot:  EXP EXP     -> EXP;
oper minus: EXP         -> EXP;
oper plus:  EXP         -> EXP;
```

```
oper length:      EXP                 -> EXP;
oper address:     VAR                 -> EXP;
oper call:        EXP ARGS            -> EXP;
oper void_const:                      -> EXP;
oper int_const:   INT                 -> EXP;
oper compound:    EXP_SEQ             -> EXP;
oper proc_const:  FORMALS EXP_SEQ     -> EXP;
oper block:       DECLS EXP_SEQ       -> EXP;
oper loop:        %TYPE ID EXP_SEQ    -> EXP;
oper %if:         EXP EXP_SEQ EXP_SEQ -> EXP;
oper value_of:    VAR                 -> EXP;

<exp> ::= <exp0>                      { <exp0> };

-- The bracketed operators are used to generated precedence
-- rules in the LR parser specification.

<exp0> ::=
  <exp1>                              { <exp1> }
| <exp0> "break" <exp0>  ["break"]    { break (<exp0_1>, <exp0_2>) }
| ID ":=" <exp0>         [":="]       { assign (variable (ID), <exp0>) }
| <exp2> '^' ":=" <exp0> [":="]       { assign (deref (<exp2>), <exp0>) }
| <exp2> '[' <exp> ']' ":=" <exp0>    [":="]
                         { assign (arrayref (<exp2>, <exp>), <exp0>) }
| <exp0> "="  <exp0>     ["="]        { eq (<exp0_1>, <exp0_2>) }
| <exp0> "/=" <exp0>     ["/="]       { neq (<exp0_1>, <exp0_2>) }
| <exp0> "<=" <exp0>     ["<="]       { le (<exp0_1>, <exp0_2>) }
| <exp0> ">=" <exp0>     [">="]       { ge (<exp0_1>, <exp0_2>) }
| <exp0> "<"  <exp0>     ["<"]        { lt (<exp0_1>, <exp0_2>) }
| <exp0> ">"  <exp0>     [">"]        { gt (<exp0_1>, <exp0_2>) }
| <exp0> "+"  <exp0>     ["+"]        { sum (<exp0_1>, <exp0_2>) }
| <exp0> "-"  <exp0>     ["-"]        { diff (<exp0_1>, <exp0_2>) }
| <exp0> "*"  <exp0>     ["*"]        { prod (<exp0_1>, <exp0_2>) }
| <exp0> "/"  <exp0>     ["/"]        { quot (<exp0_1>, <exp0_2>) }
| "length" <exp0>        ["length"] { length (<exp0>) }
| "-" <exp0>             ["length"] { minus (<exp0>) }
| "+" <exp0>             ["length"] { plus (<exp0>) };

<exp1> ::=
  <exp2>                              { <exp2> }
| "address" <exp2> '^'                { address(deref (<exp2>)) }
| "address" <exp2> '[' <exp> ']'      { address (arrayref(<exp2>, <exp>)) };
```

```
<exp2> ::=
  <var>                              { value_of (<var>) }
| '(' <exp_seq> ')'                  { compound (<exp_seq>) }
| <exp2> '(' <args> ')'              { call (<exp2>, <args>) }
| <const>                            { <const> }
| "loop" <%type> ID "do" <exp_seq> "od"
                                     { loop (<%type>, ID, <exp_seq>) }
| "if" <exp> "then" <exp_seq> "else" <exp_seq> "fi"
                                     { %if (<exp>, <exp_seq_1>, <exp_seq_2>) }
| "new" <decls> "in" <exp_seq> "ni"
                                     { block (<decls>, <exp_seq>) };

-- variable references

sort VAR;
oper var_empty:  placeholder -> VAR;
oper variable:   ID          -> VAR;
oper deref:      EXP         -> VAR;
oper arrayref:   EXP EXP     -> VAR;

<var> ::=
  ID                               { variable (ID) }
| <exp2> '^'                       { deref (<exp2>) }
| <exp2> '[' <exp> ']'             { arrayref (<exp2>, <exp>) };

-------------------- The attribute grammar -----------------------------

-- Declarations of attributes used in Appendix C:
-- (defining expressions are omitted)

sort VAR is
  -- register utilization
  syn is_register:  boolean;     -- iff the variable is in a register
end;

oper variable is
  local static_offset:  int;     -- number of steps up the static chain
  local offset:  int;            -- offset in the activation record
  local register:  int;          -- the register number if is_register
end;
```

```
-- Attributes for detecting constant expressions and determining
-- the value of some constant expressions:

sort EXP is
  syn is_constant:  boolean;            -- iff a constant expression
  syn constant_value:  int;             -- value of constant expressions
end;

-- The int_const operator has a single operand:  INT
-- An instance of the generator INT is considered to have a single
-- synthesised attribute of type INT, called "value", that is the
-- particular instance.

oper int_const is
  $$.is_constant = true;
  $$.constant_value = $1.value;
end;

-- The binary expressions are constant expressions iff both operands
-- are constant.  The defining equations for the sum operator are shown:

oper sum is
  $$.is_constant    = $1.is_constant and $2.is_constant;
  $$.constant_value = $1.constant_value + $2.constant_value;
end;

-- The evaluation of the constant value for the quotient operator shows
-- the error mechanism used in the attribute grammar specification.  An
-- error is associated with the node if the denominator is both constant
-- and zero.

oper quot is
  $$.is_constant    = $1.is_constant and $2.is_constant and not error;
  $$.constant_value =
    ($2.constant_value = 0 ?  0 :  $1.constant_value / $2.constant_value);
error
  when $2.is_constant and $2.value = 0 =>
    "divide by zero in constant expression";
end;
```

Appendix C

Extract from an architecture description

This appendix contains extracts from the SPARC architecture description for a small language. An extract from the MultiView language specification language (LSL) description of the same source language is contained in Appendix B. The architecture-dependent components of the code generator described in Chapter 7 are derived from the complete version of the description below.

```
-------------------- extract from toy.burs -----------------------------

-- The fixed goal of the rewrite system is "rvalue":
goal rvalue;

-------------------- semantic extension --------------------------------

-- Addition operators for the semantic extension of the
-- abstract syntax must be explicitly declared.

-- "smallint" and "largeint" are used to resolve ranges
-- of integer literals:
smallint[cut];
largeint[cut];

-- "local_var", "non_local_var", and "register_var" distinguish
-- the assorted flavours of variable references:
local_var[cut];
register_var[cut];
non_local_var[cut];
```

```
-------------------- relabelling specification -------------------------

-- Integer constants are resolved in small and large integers:

int_const =>
  constant_value < 2**13 => smallint { constant_value }
  otherwise              => largeint { constant_value };

  -- The "{ constant_value }" construction indicates that the
  -- constant_value attribute should be accessible in code templates.

-- Variable references are resolved into three flavours:

variable =>
  is_register       => register_var  { register }
  static_offset = 0 => local_var     { offset }
  otherwise         => non_local_var { static_offset, offset };

-------------------- normalisation noise --------------------------------

-- Additional non-terminals and zero cost rewrite rules are needed to
-- ensure that the grammar is in normal form:

largeint.1 -> largeint, 0
smallint.1 -> smallint, 0

register_var.1 -> register_var, 0
local_var.1 -> local_var, 0
non_local_var.1 -> non_local_var, 0

-------------------- rewrite rules and code templates -------------------

-- The root of the abstract syntax tree will always be labelled with
-- the "program" operator.  The emitted object code ensures that the
-- local environment is correctly configured.

rvalue -> program (rvalue), 4
  // .text
  // .align       4
  // save %sp,-SA(MINFRAME+4),%sp !  just a static link on the AR
  // st   %g0,[%fp-4]             !  the end of the static chain
  // @1                           !  evaluate program
  // jmpl %i7+8,%g0               !  return to caller
  // restore      %17,0,%o0       !  and return the value
```

```
-- In the absence of a systematic approach to register allocation, an
-- expression stack is used.  Global register %g7 is the expression
-- stack pointer.  Rvalues are evaluated into the register %l7.  Lvalues
-- are evaluated in %l6.

-- An rvalue can be derived from an lvalue:

rvalue -> lvalue, 1
  // @@                    !  insert code to evaluate the lvalue
  // ld    [%l6],%l7       !  load the value address by the lvalue

-- The lvalue of local and global variables can be evaluated in different
-- ways.

-- Local variables are easy:

lvalue -> local_var.1, 1
  // add  %fp,$1.offset,%l6      !  address of the local variable

-- ...  but to find a non-local variable we have to follow the static
-- chain up the call stack:

lvalue -> non_local_var.1, 7
  // add  %g0,$1.static_offset,%l0      !  number of links to follow
  // add  %fp,0,%l1                     !  top of current AR
  // 0:subcc %l0,1,%l1                  !  decrement counter
  // bl,a 0b                            !  not there yet
  // ld   [%l0-4],%l0                   !  follow the chain
  // add  %l0,offset,%l6                !  address of the variable

-- However it is always cheaper to get the rvalue directly:

rvalue -> local_var.1, 1
  // ld   [%fp-$$.offset],%l7           !  value of the local variable

rvalue -> non_local_var.1, 7
  // add  %g0,$1.static_offset,%l0      !  number of links to follow
  // add  %fp,0,%l1                     !  top of current AR
  // 0:subcc %l0,1,%l1                  !  decrement counter
  // bl,a 0b                            !  not there yet
  // ld   [%l0-4],%l0                   !  follow the chain
  // ld   [%l0-offset],%l7              !  value of the variable

-- For register variables, it's easy to get the rvalue:

rvalue -> register_var.1, 1
  // add %g0,%i$$.register,%l7 !  value of the register variable
```

```
-- The most obvious way to implement assignment is via the lvalue and
-- the expression stack:

rvalue -> assign(lvalue,rvalue), 5
  // @1                          !  evaluate the lvalue
  // st   %16,[%g7]              !  store it on e-stack
  // add  %g7,4,%g7              !  and push e-stack
  // @2                          !  evaluate the rvalue
  // ld   [%g7-4],%16            !  recover the lvalue
  // sub  %g7,4,%g7              !  and pop e-stack
  // st   %17,[%16]              !  store the value

-- ...  but there will often be a cheaper alternative

rvalue -> assign(local_var.1, rvalue), 1
  // @2                          !  evaluate the rvalue
  // st   %17,[%fp-$1.offset]    !  store the value

rvalue -> assign(register_var.1, rvalue), 1
  // @2                          !  evaluate the rvalue
  // st   %17,%i$1.register      !  store the value

-- ...  and its even cheaper for constants
rvalue -> assign(register_var.1, smallint.1), 1
  // add  %g0,$2.constant_value,%i$1.register

rvalue -> assign(register_var.1, largeint.1), 2
  // sethi %hi($2.constant_value),%l0
  // or %l0,%lo($2.constant_value),%i$1.register
```

```
-- Binary arithmetic operations will always work with an rvalue
-- and an rvalue:

rvalue -> sum(rvalue,rvalue), 5
  // @1                              !  lhs
  // st   %17,[%g7]                  !  save lhs on e-stack
  // add  %g7,4,%g7                  !  and push e-stack
  // @2                              !  rhs
  // ld   [%g7-4],%10                !  recover the lhs from e-stack
  // add  %10,%17,%17                !  form the sum
  // sub  %g7,4,%g7                  !  and pop e-stack

-- ...  but there a several special cases when much better code
-- can be emitted:

-- when the left operand is a small constant:
rvalue -> sum(smallint.1,rvalue), 1
  // @2                                    !  rhs
  // add  %17,$1.constant_value,%17        !  form the sum

-- when the right operand is a small constant:
rvalue -> sum(rvalue,smallint.1), 1
  // @1                                    !  lhs
  // add  %17,$2.constant_value,%17        !  form the sum

-- when the left operand is a large constant:
rvalue -> sum(largeint.1,rvalue), 3
  // @2                                      !  rhs
  // sethi %hi($1.constant_value),%10
  // or   %10,%lo($1.constant_value),%11     !  lhs
  // add  %17,%10,%17                        !  form the sum

-- when the right operand is a large constant:
rvalue -> sum(rvalue,largeint.1), 3
  // @1                                      !  lhs
  // sethi %hi($2.constant_value),%10
  // or   %10,%lo($2.constant_value),%11     !  rhs
  // add  %17,%10,%17                        !  form the sum
```

Appendix D

A dispatcher specification

This appendix contains the MultiView dispatcher specification for the parallel incremental code generator view.

```
-------------------- codegen.psl ---------------------------------------
-- Specification of the dispatcher for the MultiView code generator view.
protocol MULTIVIEW version 1;
view CODEGEN is
  broadcast
    -- Notification of abstract syntax tree edits:
    when NOTIFY_EDIT =>
      { DRIVER.SOURCE_EDIT (?AT_NODE, SUBTREES.VALUE (?SUBTREE)); }
      -- "?AT_NODE" is replaced by code to access the AT_NODE field
      -- of the incomming NOTIFY_EDIT message.
    -- Notification of the deletion of a unit in the database:
    when NOTIFY_DELETE_UNIT =>
      -- "inner" causes the acknowledgment to the view to be sent
      -- before performing the specified actions.
      inner;
      { if ?UNIT = COMPILED_UNIT then  }
      {   CODEGEN_EVENTS.UNIT_DELETED; }
      { end if;                        }
    -- Directive to perform a recompilation timing trial:
    when RECOMPILE_TRIAL =>
      inner;
      { if ?UNIT = COMPILED_UNIT then                        }
      {   AST.RECOMPILE_TRIAL (?AT_NODE, ?TRIAL_DURATION); }
      { end if;                                              }
    -- Notification that the communication link to the database
    -- has been closed:
    when CLOSED =>
      { DRIVER.QUIT; }
end CODEGEN;
```

Bibliography

[Aho85] A. V. Aho and M. Ganapathi. Efficient tree pattern matching: an aid to code generation. *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (New Orleans, Louisiana), pages 334–40. New Orleans, Louisiana, January 1985.

[Aho86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[Aho89] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, **11**(4):491–516. Association for Computing Machinery, October 1989.

[ALRM83] United States Department of Defense. *Reference manual for the Ada programming language.* ANSI/MIL-STD-1815A-1983, U.S. Department of Defense, Washington, D.C., 1983.

[Altmann86] R. A. Altmann. An abstract syntax tree editor for the MultiView programming environment. Department of Computer Science, University of Adelaide, Adelaide, South Australia, October 1986. Honours dissertation.

[Altmann88] R. A. Altmann, A. N. Hawke, and C. D. Marlin. An integrated programming environment based on multiple concurrent views. *The Australian Computer Journal*, **20**(2):65–72. Australian Computer Society, May 1988.

[Altmann91] R. A. Altmann and C. D. Marlin. Prototype semantic analysis, code generation and run-time views for an integrated software development environment. *Proceedings of the Sixth Australian Software Engineering Conference* (Sydney, Australia), pages 57–68. Australian Computer Society, July 1991.

[Bahlke86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, **8**(4):547–76. Association for Computing Machinery, October 1986.

[Balachandran90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, **15**(3):127–40. Pergamon Press, 1990.

[Ballance90] R. A. Ballance, S. L. Graham, and M. L. van de Vanter. The Pan language-based editing system for integrated development environments. *Proceedings of the SIGSOFT'90 Symposium on Software Develepement Environments.* Published as *Software Engineering Notes*, **15**(6):77–93. Association for Computing Machinery, December 1990.

[Beck92] D. Beck. Extending graphical views in MultiView. Discipline of Computer Science, The Flinders University of South Australia, Adelaide, South Australia, November 1992. Honours dissertation.

[Bergstra89] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic specification.* Association for Computing Machinery, 1989.

[Birkhoff70] G. Birkhoff and J. D. Lipson. Heterogenous Algebras. *Journal of Combinatorial Theory*, **8**:115–33, 1970.

[Bivens87] M. P. M. H. Bivens. *Incremental generation of high-quality target code.* PhD thesis, published as Technical Report 87-2. Department

of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania, 1987.

[Bivens90] M. P. Bivens and M. L. Soffa. Incremental register reallocation. *Software – Practice and Experience*, **20**(10):1015–47, October 1990.

[Borras88] P. Borras, D. Clément, T. Despeyrous, J. Incerpi, G. Khan, B. Lang, and V. Pascual. Centaur: the system. *Proceedings of ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments.* Published as *Software Engineering Notes*, **13**(5):14–24. Association for Computing Machinery, November 1988.

[Brook90] J. B. Brook. TextView: a textual editing view for MultiView interfaced to the X Window System. Department of Computer Science, University of Adelaide, Adelaide, South Australia, November 1990. Honours dissertation.

[Cattell79] R. G. G. Cattell, J. M. Newcomer, and B. W. Leverett. Code generation in a machine-independent compiler. *Proceedings of ACM SIGPLAN Symposium on Compiler Construction.* Published as *ACM SIGPLAN Notices*, **14**(8):65–75, August 1979.

[Cattell80] R. G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, **2**(2):173–90, April 1980.

[Chaitin82] G. J. Chaitin. Register allocation and spilling via graph coloring. *Proceedings of ACM SIGPLAN Symposium on Compiler Construction.* Published as *ACM SIGPLAN Notices*, **17**(6):98–105. Association for Computing Machinery, June 1982.

[Chow84] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. *Proceedings of ACM SIGPLAN Symposium on Compiler*

Construction. Published as *ACM SIGPLAN Notices*, **19**(6):222–32. Association for Computing Machinery, June 1984.

[Christopher84] T. W. Christopher, P. J. Hatcher, and R. C. Kukuk. Using dynamic programming to generate optimized code in a Graham-Glanville style code generator. *Proceedings of ACM SIGPLAN Symposium on Compiler Construction.* Published as *ACM SIGPLAN Notices*, **19**(6):25–36, June 1984.

[Delisle84] N. M. Delisle, D. E. Menicosy, and M. D. Schwartz. Viewing a programming environment as a single tool. *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Published as *Software Engineering Notes*, **9**(3):49–56. Association for Computing Machinery, May 1984.

[Dewar94] R. Dewar, C. Comar, F. Gasperoni, and E. Schonberg. *The GNAT Project: a GNU-Ada9X compiler.* New York University, 1994.

[DonzeauGouge80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the Mentor experience. Rapports de Récherche, No. 26. INRIA, Domaine de Voluceau, Rocquencourt, France, July 1980.

[Earley70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, **13**(2):94–102. Association for Computing Machinery, February 1970.

[Earley72] J. Earley and P. Caizergues. A method for incrementally compiling languages with nested statement structure. *Communications of the ACM*, **15**(12):1040–4. Association for Computing Machinery, December 1972.

[Emmelmann89] H. Emmelmann, F. W. Schröer, and R. Landwehr. BEG – A generator for efficient back ends. *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation.* Published as *ACM SIGPLAN Notices*, **24**(7):227–37, July 1989.

[Feiler82] P. H. Feiler. *A language-oriented interactive programming environment based on compilation technology.* PhD thesis, published as Technical Report CMU-CS-82-117. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1982.

[Feldman79] S. I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, **9**:255–65, 1979.

[Ford85] R. Ford and D. Sawamiphakdi. A greedy approach to incremental code generation. *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (New Orleans, Louisiana), pages 165–78. Association for Computing Machinery, January 1985.

[Fraser91a] C. W. Fraser and D. R. Hanson. A code generator interface for ANSI C. *Software – Practice and Experience*, **21**(9):963–88, September 1991.

[Fraser91b] C. W. Fraser and R. R. Henry. Hard-coding bottum-up code generation tables to save time and space. *Software – Practice and Experience*, **21**(1):1–12, January 1991.

[Fraser91c] C. W. Fraser, R. R. Henry, and T. A. Proebsting. *BURG – fast optimal instruction selection and tree parsing.* Department of Computer Science, University of Wisconsin, Madison, Wisconsin, 1991.

[Fraser92] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, **1**(3):213–26, September 1992.

[Fritzson82] P. Fritzson. Fine-grained incremental compilation for Pascal-like languages. Research Report LiTH-MAT-R-82-15. Software Systems Research Center, Linköping Institute of Technology, Linköping, Sweden, July 1982.

[Fritzson83] P. Fritzson. Symbolic debugging through incremental compilation in an integrated environment. *Journal of Systems and Software*, **3**:285–94, 1983.

[Fritzson84a] P. Fritzson. The architecture of incremental programming environments and some notions of consistency. In *Towards a distributed programming environment based on incremental compilation*, pages 59–79. Department of Computer and Information Science, Linköping University, Linköping, Sweden, Linköping Studies in Science and Technology, Dissertation No. 109, 1984.

[Fritzson84b] P. Fritzson. *Towards a distributed programming environment based on incremental compilation.* PhD thesis, published as Linköping Studies in Science and Technology, Dissertation No. 109. Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1984.

[Fritzson84c] P. Fritzson. Preliminary experience from the DICE system – a distributed incremental compiling environment. *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Published as *ACM SIGPLAN Notices*, **19**(5):113–23. Association for Computing Machinery, May 1984.

[Fritzson92] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Letters on Program-*

ming Languages and Systems, **1**(4):303–22. Association for Computing Machinery, December 1992.

[Fritzson94] P. Fritzson, M. Auguston, and N. Shahmehri. Using assertions in declarative and operational models for automated debugging. *Journal of Systems and Software*, **25**:223–39, June 1994.

[Fritzson95] P. Fritzson. Private communication, September 1995. Correspondence with the author.

[Gafter90] N. M. Gafter. *Parallel incremental compilation.* PhD thesis, published as Technical Report 349. Department of Computer Science, The University of Rochester, Rochester, New York, June 1990.

[Ganapathi82] M. Ganapathi, C. N. Fischer, and J. L. Hennessy. Retargetable Code Generation. *ACM Computing Surveys*, **14**(4):573–92, December 1982.

[Ganapathi85] M. Ganapathi and C. N. Fischer. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems*, **7**(4):560–99, October 1985.

[Glanville77] R. S. Glanville. *A machine independent algorithm for code generation and its use in retargetable code generators.* PhD thesis. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California – Berkeley, Berkeley, California, 1977.

[Glanville78] R. S. Glanville and S. L. Graham. A new method for compiler code generation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (Tucson, Arizona), pages 231–40. Association for Computing Machinery, January 1978.

[Gray92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: a complete flexible compiler construction system. *Communications of the ACM*, **35**(2):121–31. Association for Computing Machinery, February 1992.

[Hatcher85] P. J. Hatcher. *A tool for high-quality code generation.* PhD thesis. School of Advanced Studies, Illinois Institute of Technology, Chicago, Illinois, 1985.

[Hedin92] G. Hedin. *Incremental semantic analysis.* PhD thesis, published as Technical Report LUTEDX/(TECS-1003)/1-276/(1992). Department of Computer Science, Lund University, Lund, Sweden, 1992.

[Henry84] R. R. Henry. *Graham-Glanville code generators.* PhD thesis, published as Report UCB/CSD 84/184. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California – Berkeley, Berkeley, California, May 1984.

[Henry89] R. R. Henry. Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Technical Report 89-02-04. Department of Computer Science, University of Washington, Seattle, Washington, February 1989.

[Higgins62] P. J. Higgins. Algebras with a Scheme of Operators. *Mathematische Nachrichten*, **27**:115–32, 1962.

[Hoare78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–76. Association for Computing Machinery, August 1978.

[Hoffmann82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, **29**(1):68–95. Association for Computing Machinery, January 1982.

[Hoover86] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. *Proceedings of ACM SIGPLAN Symposium on Compiler Construction.* Published as *ACM SIGPLAN Notices*, **21**(7):39–50, July 1986.

[Horwitz85] S. B. Horwitz. *Generating language-based editors: a relationally–attributed approach.* PhD thesis, published as Technical Report 85-696. Department of Computer Science, Cornell University, Ithaca, New York, August 1985.

[Horwitz86] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, **8**(4):577–608, October 1986.

[Jacobs91] D. A. Jacobs. FlowView: a flowchart editing system for the MultiView programming environment. Department of Computer Science, University of Adelaide, Adelaide, South Australia, November 1991. Honours dissertation.

[Jacobs95] D. A. Jacobs and C. D. Marlin. Unparsing flowcharts from abstract syntax trees in a multiple view software development environment. *Proceedings of the Eighteenth Australasian Computer Science Conference* (Adelaide, South Australia). Published as R. Kotagiri, editor, *Australian Computer Science Communications*, **17**(1):217–26, February 1995.

[Jensen74] K. Jensen and N. Wirth. *Pascal user manual and report.* Springer-Verlag, Berlin, 1974.

[Johnson78] S. C. Johnson. A portable compiler: Theory and practice. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages* (Tucson, Arizona), pages 97–104. Association for Computing Machinery, January 1978.

[Kahn83] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. METAL: a formalism to specify formalisms. *Science of Computer Programming*, pages 151–88. North-Holland, Amsterdam, 1983.

[Kastens80] U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, **13**:229–56, 1980.

[Klint93] P. Klint. A meta-environment for generating programming environments. *TOSEM*, **2**(2):176–201. Association for Computing Machinery, April 1993.

[Kron75] H. H. Kron. *Tree templates and subtree transformational grammars.* PhD thesis. Department of Computer Science, University of California – Santa Cruz, Santa Cruz, California, 1975.

[Kung80] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search tree. *ACM Transactions on Database Systems*, **5**(3):354–83. Association for Computing Machinery, September 1980.

[Lee87] C. S. Lee. A textual view for the MultiView programming environment. Department of Computer Science, University of Adelaide, Adelaide, South Australia, October 1987. Honours dissertation.

[Leffler83] S. J. Leffler, R. S. Fabry, and W. N. Joy. A 4.2bsd interprocess communication primer. Report UCB/CSD 83/145. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California – Berkeley, Berkeley, California, 1983.

[Lehman81] P. L. Lehman and S. Yao. Efficient locking for concurrent operations on B-tree. *ACM Transactions on Database Systems*, **6**(4):650–70. Association for Computing Machinery, December 1981.

[Linton84] M. A. Linton. Implementing relational views of programs. *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on*

Practical Software Development Environments. Published as *ACM SIGPLAN Notices*, **19**(5):132–40, May 1984.

[Marlin86] C. D. Marlin. MultiView: an integrated incremental programming environment with multiple concurrent views. *Proceedings of Seminar on Parallel Computing Architectures* (Telecom Research Laboratories, Clayton, Victoria, Australia), pages 171–80, February 1986.

[Marlin90] C. D. Marlin. A distributed implementation of a multiple view integrated programming environment. *Proceedings of Fifth Knowledge-Based Software Assistant Conference* (Syracuse, New York), pages 388–402, 1990.

[Marlin93] C. D. Marlin, B. Peuschel, M. J. McCarthy, and J. Harvey. Multi-View-Merlin: An experiment in tool integration. *Software Engineering Environments* (Reading, United Kingdom, July 7-9, 1993), pages 35–48. Institute of Electrical and Electronics Engineers, 1993.

[McCarthy85] M. J. McCarthy. Towards an integrated programming environment based on multiple concurrent processes. Department of Computer Science, University of Adelaide, Adelaide, South Australia, November 1985. Honours dissertation.

[McCarthy94] M. J. McCarthy and C. D. Marlin. Interprocess communication support in a distributed integrated software development environment. *Proceedings of the Seventeenth Annual Computer Science Conference* (Christchurch, New Zealand). Published as G. Gupta, editor, *Australian Computer Science Communications*, **16**(1):363–72, January 1994.

[MedinaMora81a] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, **7**(5):472–82, September 1981.

[MedinaMora81b] R. Medina-Mora and D. S. Notkin. ALOE users' and implementors' guide. Technical Report CMU–CS–81–145. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1981.

[Mughal88] K. A. Mughal. Generation of incremental indirect threaded code for language based environments. In D. Hammer, editor, *Compiler compilers and high speed compilation*, Volume 371 of Lecture Notes in Computer Science, pages 230–42. Springer-Verlag, Berlin, October 1988.

[Notkin85] D. Notkin. The GANDALF project. *Journal of Systems and Software*, 5:91–106, May 1985. Special Issue on the GANDALF project.

[PelegríLlopart88a] E. Pelegrí-Llopart. *Rewrite systems, pattern matching and code generation.* PhD thesis. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California – Berkeley, Berkeley, California, 1988.

[PelegríLlopart88b] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: An application of BURS theory. *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (San Diego, California), pages 294–308. San Diego, California, January 1988.

[Pollock85] L. L. Pollock and M. L. Soffa. Incremental compilation of locally optimized code. *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 152–64, January 1985.

[Pollock86] L. L. Pollock. *An approach to incremental compilation of locally optimized code.* PhD thesis, published as Technical Report 86-3. De-

partment of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania, 1986.

[Pollock92] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems*, **14**(2):173–200, April 1992.

[Proebsting92] T. A. Proebsting. *Code generation techniques.* PhD thesis. University of Wisconsin, Madison, Wisconsin, 1992.

[Read93] M. C. Read. TextView: a textual view for the MultiView environment. Discipline of Computer Science, The Flinders University of South Australia, Adelaide, South Australia, November 1993. Honours dissertation.

[Reiss84a] S. P. Reiss. An approach to incremental compilation. *Proceedings of ACM SIGPLAN Symposium on Compiler Construction.* Published as *ACM SIGPLAN Notices*, **19**(6):144–56. Association for Computing Machinery, June 1984.

[Reiss84b] S. P. Reiss. Graphical program development with PECAN program development systems. *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Published as *ACM SIGPLAN Notices*, **19**(5):30–41, May 1984.

[Reiss90a] S. P. Reiss. Connecting tools using the message passing in the Field environment. *IEEE Software*, **7**(4):57–66. Institute of Electrical and Electronics Engineers, July 1990.

[Reiss90b] S. P. Reiss. Interacting with the FIELD environment. *Software – Practice and Experience*, **20**(S1):89–115, June 1990.

[Reps83] T. Reps. *Generating Language Based Environments.* PhD thesis. Department of Computer Science, Cornell University, Ithaca, New York, 1983.

[Reps89a] T. Reps and T. Teitelbaum. *The Synthesizer Generator, a system for constructing language-based editors*, Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1989.

[Reps89b] T. W. Reps and T. Teitelbaumn. *The Synthesizer Generator reference manual, Third edition.* Springer-Verlag, Berlin, 1989.

[Santi89] Ø. Santi. Retargeting of an incremental code generator to MC68020. Research Report LiTH-IDA-R-889-13. Software Systems Research Center, Linköping Institute of Technology, Linköping, Sweden, March 1989.

[Sawamiphakdi84] D. Sawamiphakdi. *A multiprocess design for an integrated programming environment.* PhD thesis, published as Technical Report 84-08. Department of Computer Science, The University of Iowa, Iowa City, Iowa, July 1984.

[Schwartz84] M. D. Schwartz, N. M. Delisle, and V. S. Begwani. Incremental compilation in Magpie. *Proceedings of ACM SIGPLAN Symposium on Compiler Construction.* Published as *ACM SIGPLAN Notices*, **19**(6):122–31, June 1984.

[Stallman94] R. M. Stallman. *Using and porting GNU CC.* Free Software Foundation, Cambridge, Massachusetts, September 1994.

[Sun87] Sun Microsystems, Inc. *The SPARC Architecture Manual*, Part No. 800-1399-07, August 1987.

[Taback88] D. Taback, D. Tolani, and R. J. Schmalz. *AYACC user's manual*, Technical Report UCI-88-16. Arcadia Environment Research Project,

Department of Information and Computer Science, University of California, Irvine, California, May 1988.

[Teitelbaum81] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax directed programming environment. *Communications of the ACM*, **24**(9):563–73. Association for Computing Machinery, September 1981.