HIERARCHICAL MODULARITY AND INTERMODULE OPTIMIZATION

MATTHIAS BLUME

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

NOVEMBER 1997

© Copyright by Matthias Blume, 1997.

All Rights Reserved

Abstract

Separate compilation is an important tool for coping with design complexity in large software projects. When done right it can also be used to create software libraries, thus promoting code reuse. But separate compilation comes in various flavors and has many facets: namespace management, linking, optimization, dependencies.

Many programming languages identify modular units with units of compilation, while only a few extend this to permit hierarchies of language-level modules within individual compilation units. When the number of compilation units is large, then it becomes increasingly important that the mechanism of separate compilation itself can be used to control namespaces.

The group model implemented in SML/NJ's compilation manager CM provides the necessary facilities to avoid unwanted interferences between unrelated parts of large programs. Compilation units are arranged into groups, and explicit export interfaces can be used to control namespaces. When there are many groups, they can be organized into supergroups, and so on, thus forming a hierarchical modular structure. CM provides automatic dependency analysis, but automatic dependency analysis is NP-hard for general SML code. We show two simple restrictions that avoid intractability.

To remove the penalties for efficiency usually incurred by modularization and separate compilation, I designed an algorithm for automatic inline expansion across compilation unit boundaries that works in the presence of higher-order functions and free variables; it rearranges bindings and scopes as necessary to move non-expansive code from one module to another. I describe—and implement—the algorithm as transformations on λ-calculus. The inliner is efficient, robust, and practical enough for everyday use in the SML/NJ compiler. It preserves separate compilation and has been integrated with CM.

I briefly investigate macro systems as an alternative approach—driven by programmer

directives—to achieve cross-module inlining and discuss a variety of problems that arise. I show a solution to the problem of integrating macro systems with ML-style modules that use long identifiers and show an implementation technique for reliable name resolution during linking. But I also discover that other problems continue to impede large-scale programming.

Acknowledgments

First of all, I want to thank Andrew Appel for being an excellent, caring, and very dependable advisor, for helping me stay on track with this work, for many insights he provided me with, and for shaping my appreciation of higher-order typed programming languages.

Thanks to faculty and staff of Princeton University's Computer Science Department for creating a stimulating research environment and for providing outstanding technical facilities.

There are many people to whom I am indebted for their direct or indirect support. Princeton faculty: Douglas Clark, Edward Felten, David Hanson, Anne Rogers. The SML/NJ team: Emden Gansner, Lal George, Lorenz Huelsbergen, Dave MacQueen, John Reppy. My former officemate—also a member of the SML/NJ team and Andrew's advisee: Zhong Shao. At ATR in Kyoto: Ed Gamble.

There are too many fellow graduate students and friends to list them all. But I want to mention the many hours of useful discussions I had with my housemates Seth Bruder and Elçin Yildirim.

Special thanks to my undergraduate advisor, Professor Christoph Polze in Berlin, and to Professor Axel T. Schreiner in Osnabrück.

Of course, my love goes to my parents—whose lifelong support has laid the foundations.

To my parents Gesine and Klaus-Dieter Blume.

Contents

Abstract iii

1 Introduction **1**

1.1 Hierarchical modularity 4

1.2 The compilation manager CM 5

1.3 Automatic dependency analysis 7

1.4 Cross-module optimization 11

1.5 Macro systems 19

1.6 Contributions 21

2 The design of CM **23**

2.1 Simple groups 26

2.2 Restrictions 27

2.3 Hierarchies 28

2.4 Caches, missing sources, and stable groups 35

3 Hierarchical Modularity **38**

3.1 Division into groups 38

3.2 The compilation manager CM 43

3.3 Environments and linking 46

3.4 Definability and availability 50
3.5 Groups 53
3.6 How interferences can be resolved 56
3.7 Groups vs. modules, namespaces, and packages 59
3.8 Realization in CM 60

4 Automatic dependency analysis **62**

4.1 Multiple definitions 63
4.2 Opening structures 70
4.3 The analysis algorithm 79
4.4 Optimizations 86
4.5 Implications for language design 88

5 Cross-Module Optimizations **89**

5.1 Example 91
5.2 Separate compilation and linking 95
5.3 λ-splitting 97
5.4 Functions 109
5.5 Heuristics 115
5.6 Implementation and results 121

6 Macros **131**

6.1 A review of hygienic macro expansion 134
6.2 Macros and modules 136
6.3 Combining macros and modules 139
6.4 Separate compilation and linking 148
6.5 Compilation management 156

6.6 Macros in large-scale programming 159

7 Conclusions **160**

7.1 Other languages 161

7.2 Lessons for language design 163

7.3 Future work 165

7.4 Enabling technology 166

A CM reference manual **168**

A.1 Group description syntax 168

A.2 Examples 169

A.3 Preprocessor 170

A.4 Default classes 172

A.5 Available tools 172

A.6 Export rules 173

A.7 Binfile names 174

A.8 Feature summary 174

A.9 All CM commands 182

A.10 Adding new tools to CM 184

A.11 Using CM for compiling the compiler 190

B Source code for FFT benchmark **197**

B.1 An abstract datatype for complex numbers 197

B.2 Sample Implementation of FFT 198

C Benchmark results in numbers **200**

C.1 Compiler passes as benchmarks 200

C.2 Fast Fourier Transform 203

Chapter 1

Introduction

The need to split programs into smaller pieces and to compile the resulting fragments separately arose from very practical considerations: machines were too small and compilers were not efficient enough to handle large bodies of code at once. But the so-established physical boundaries—when placed cleverly—can have a profoundly positive impact on overall program structure. Separate compilation was soon adopted as a key element of modern software engineering [Car97].

Divide-and-conquer. Abstraction and modularization are the "divide-and-conquer" of software engineering. The ability to verify partial designs in isolation from each other enables programmers to operate in teams. When hundreds, or even thousands, of people work on the same body of code, it is important that individual pieces be well separated. Otherwise, the need for communication between programmers with the purpose of coordinating their individual tasks quickly gets out of hand.

To illustrate this point, let us assume that one person can handle programming tasks with some limited number of subcomponents and their interdependencies. Now consider a big software system that is several orders of magnitude larger in size. Ideally, we would hire

just enough programmers so their individual task sizes add up to that of the big software project. But if a component can potentially depend on any one of the other components in the entire program, then each programmer still faces a problem comparable in size to that of the whole project. This is much more than he or she could possibly handle. The worst aspect of this situation is that it cannot be improved significantly—not even by hiring more programmers.

One could argue that in reality a typical component only depends on a small number of others. However, it is not enough to know that *there are a few* other modules that we need to be aware of—we must know *which of the many* existing components those are. Otherwise it is of no help that the dependency graph is sparse—we still must consider the dense graph of potential dependencies.

Modularization is of great help, even in single-person projects, because it provides a way of serializing the work. The programmer can focus efforts on one part without having to worry about too many possible implications for others.

Type checking and interfaces. Type systems are compile-time decidable formal systems that can track inter-module dependencies. With type-safe linking, the programmer will be notified of any existing incompatibilities at the time the pieces that have been compiled separately are patched together to form an executable program. But that is often much too late; during development it is important to detect problems early, because then there will be less work that needs to be revised or redone.

Explicit interfaces are a way of writing down where and how modules can depend on each other. Thus, they can cut the graph of potential dependencies from dense to sparse. Adherence to the constraints laid out in interface definitions can be verified at compile time, which saves precious development time.

The need for separate compilation. However, abstraction, modularization, and separate design do not necessarily imply a need for separate code generation. Computers have become much faster and much bigger in terms of storage capacity. Many tasks that were previously infeasible even on large mainframes can now easily be accomplished by personal computers or desktop workstations. Of course, programs themselves have also grown proportionally with the increased power of the computers on which they run. Thus, separate compilation, including separate code generation, is still important for simple practical reasons.

But these considerations are not the only motivations. Another advantage of separate code generation is the ability to reuse executable code. Over time many useful algorithms and data structures have been packaged and put into libraries, so they can be used unchanged as part of different projects. As an alternative, libraries could store source code, but then the effort of compiling individual components would have to be duplicated every time such components are used. However, potential gains in efficiency of the resulting code rarely justify significantly increased time and space requirements incurred by repeated recompilation. Also, in many cases, software vendors are reluctant to ship source instead of compiled versions of their products.

Code sharing can be seen as an extreme case of code reuse. In multitasking environments it is possible for two or more programs to physically share the code of common components—even at runtime. But unless *all* these programs are compiled together, it will then be necessary for the code in shared components to be generated separately. It is also possible to generate code for certain parts of a program that is already running. Runtime code generation sometimes improves efficiency because information that only becomes available at runtime can be used for optimizations. Specialized versions of a critical portion often help to boost overall performance [Kep91, LL96].

1.1 Hierarchical modularity

We have seen how important it is for parts of large programs to be well separated. But since the pieces of the program must be able to address each other, a form of naming is required. Unfortunately, the common practice of using one global namespace for this purpose undermines the original goal of separation. The names inhabiting the global namespace can clash, and the potential of name clashes creates new non-modular dependencies between software components.

Names are needed when one program component wants to refer to something defined by another. I call such components *related*. Certainly, the programmer of the first component must have been aware of the existence of the second, otherwise he or she would not have wished to refer to a definition therein. Therefore, it is relatively easy to coordinate the development of related components because name clashes will be detected early in the development cycle. Furthermore, the occasional conflict can be resolved where it occurs. It does not require modifications to unrelated components.

On the other hand, the potential of clashes between unrelated modules is much more serious. Sometimes one simply cannot use the libraries sold by two different vendors, even though they are not conceptually connected and are used for different purposes in unrelated parts of the program.

Many library designers try to minimize the danger of name clashes by prefixing all names exported by their libraries. But since we do not have a central authority that governs the choice of such prefixes, there is no guarantee that the prefixes themselves will not clash. If two C libraries export two different definitions under the same name, then there will be a link-time error. In general, the conflict must be resolved by modifying one of the libraries. Thus, an unfortunate choice of names by the library vendor will be detected by its customer only much later, at a time when it is often too late to fix the problem. Even when the entire

source code for all libraries is available, it can be a challenging problem to resolve all naming conflicts [FBB+97, section 4.7.2].

The C programming language [Ans90] gives us a distinction between "external" and "static" (meaning "non-external") names. Block-structure extends this to permit many levels of "non-external" names. But there should also be many levels for being "external" because groups of separately compiled components often need to share among themselves without also sharing with the rest of the world.

1.2 The compilation manager CM

In order to address and solve these problems, I introduce *groups* as a way of structuring large software systems. Groups were implemented in CM, the compilation manager for Standard ML of New Jersey [AM91]. Chapter 2 provides an overview of CM; chapter 3 rationalizes the idea of groups and gives a formal development.

The purpose of CM is comparable to that of other compilation and configuration managers. Examples are **make** [Fel79], Odin [Cle94], the System Modelling language of Mesa and Cedar [MMS79, LS83b, LS83a, SZH85][1], and Vesta [LM93, HL93, CL93].

Group descriptions play the same role for CM that system models play for Vesta and makefiles play for **make**.

The most important new aspect of CM and its configuration languages is the way it supplements the underlying programming language. This language—Standard ML [MTH90, MTHM97]—has very powerful means of controlling scope within compilation units. CM extends this to control the scope of names that are used for the communication between compilation units. As I will explain in chapter 3, the result of this combination, the CM

[1] It is a coincidence that this language was also called *SML*; Cedar's SML and the programming language known as SML are unrelated.

group model, has very desirable properties for the construction of large software systems because it eliminates the single global namespace and its associated potential for non-modular name conflicts.

CM should be viewed as a modest extension of SML. SML and CM are tightly coupled, and this tight coupling makes it possible to keep the configuration language very simple. CM is convenient to use because automatic dependency analysis avoids the need for extensive hand-crafted specifications.

Of course, all this comes at a cost. CM inherently depends on SML, although it would be possible to adapt the same ideas to other languages as well. Therefore, the benefits of hierarchical modularity and automatic dependency analysis do not directly carry over to other languages. CM's extensible toolbox can accomodate a variety of other language processors, but those are mere add-ons that are not integrated with the sophisticated management of SML code.

CM is predated by the "Incremental Recompilation Manager" [HLPR94b], which later became known as SC [HLPR94a]. IRM and SC were first to take advantage of SML/NJ's visible compiler interface with its support for *cutoff recompilation*. But the group model implemented by SC lacked explicit export filters and provided only rudimentary support for libraries and hierarchical modularity. A solution to the problem with global namespaces, based on link-time renaming of identifiers, has been reported for Vulcan, an experimental Modula-2+ programming environmant that was developed as part of Vesta [BE93].

The CM software constitutes an important practical contribution of my work. Ever since it was first introduced, its users have often praised the advantages for program development and maintenance. Meanwhile, CM has become an integral part of SML/NJ [Blu95].

Cutoff recompilation. The main service offered by CM (or any other compilation manager) is a mechanism to establish consistency between sources and *derived objects*. CM's

most important derived object is the *binfile*. Binfiles are the result of compiling SML compilation units. Such compilation units almost always depend on several other compilation units. A binfile consists of two parts: executable code and a *static environment*. The static environment plays the role of a symbol table that records type information for definitions exported by the compilation unit.

If `b.sml` depends on `a.sml`, then the compiler must take into account the static environment recorded in `a.sml.bin` to be able to produce the binfile `b.sml.bin`. Therefore, whenever the static environment in `a.sml.bin` changes, `b.sml` must also be recompiled.

Make safely approximates this by recompiling `b.sml` every time `a.sml` gets recompiled. However, such an approach can be overly pessimistic. Deep dependency graphs, as occur frequently in SML programs, lead to many unnecessary recompilations. As long as the static environment in `a.sml.bin` stays unaltered, recompiling `a.sml` does not require subsequent recompilation of `b.sml`. Cutoff recompilation [ATW94], which is the strategy used by CM, takes advantage of this observation. For efficiency, instead of comparing entire static environments, CM only compares relatively small fingerprints. (The fingerprints are based on CRC polynomials [Bro93].) Gunter explains how to deal with the extremely unlikely case of error due to imperfections of this technique [Gun96].

1.3 Automatic dependency analysis

The order of compilation matters for ML programs. CM provides automatic dependency analysis, thereby maintaining the illusion of unordered source collections. This significantly simplifies the task of writing group descriptions. Automatic dependency analysis makes CM easier to use and therefore more attractive as a tool.

Most other compilation and configuration management tools do not provide automatic

dependency analysis but require the programmer to specify dependencies explicitly. Since dependency information is usually coded in some specification language, one can imagine adding dependency analysis using an auxiliary program that calculates and generates specifications. Examples of this are the **imake** and **makedepend** tools that generate input for **make** from C source code [DuB96].

In the case of **make**, this idea works most of the time, but only because programmers usually disregard the fact that the entire recompilation process depends on the specification of dependencies. Had this been spelled out, then every derived object would technically depend on the makefile, which in turn depends on all sources. Any modification at all would then require the entire system to be rebuilt from scratch.

At the heart of the problem is the bundling of information; the makefile serves as a singular bottleneck in the dependency graph. A similar effect occurs when certain operations only use part of the information available to them. We have already seen an example of this when cutoff recompilation was discussed.

The same problem has been identified in the context of Vesta [LM93]. Vesta is based on immutable objects; every modification to a source produces a new source. This model works well with the Vesta configuration language [HL93], which describes dependencies in declarative style. A sophisticated caching system avoids recompilation where possible. The cache persistently remembers results of function calls. If the same function is later called with the same arguments, they can be found and re-used. However, not every change to an argument of an operation causes a corresponding change in the result. This can lead to false dependencies and spurious recompilations. Abadi, Lampson and Lévy developed a mechanism for finding those parts of an expression that contribute to its value [ALL96]. Such a mechanism can be used to alleviate the problem.

Dependency analysis for ML. In ML, each top-level definition implicitly starts a new nested scope. Thus, an existing definition does not preclude subsequent redefinitions of the same name. And yet, it is guaranteed that later definitions cannot affect earlier uses. Consider the following code:

```
val a = 1
val x = a
val f = fn () => a
```

Later, in a different section of the program one can "recycle" the name `a` by giving it a new definition for a new purpose:

```
val a = "hello, world"
```

Normally, we expect `x` not to be affected by this; `x` still evaluates to `1` as before. Function `f` was also defined in terms of `a`, and certainly it should still refer to the old definition, as is the case in SML. Other languages, notably Scheme [Ce91], behave differently.

If the body of `f` would refer to the new definition of `a`, then the variable `f` would also represent a new value, in this case even with a different type. Previous uses of `f` might be incompatible with (or at least affected by) such a change.

SML's behavior avoids these complications. But it means that rearranging the order of source files amounts to rearranging name visibility and scopes. The following code ultimately binds `x` to `1`:

```
(* 1 *) val a = 1
(* 2 *) val x = a
(* 3 *) val a = 2
```

But if one rearranges the definitions by exchanging lines 2 and 3, then `x`, like `a`, will be bound to 2. In ML it does not matter whether bindings appear on consecutive lines or in separate sources files.

Dependency analysis must find an ordering that is *feasible* and *unique*. By definition, a feasible ordering allows the program to be compiled successfully. The uniqueness require-

ment means that no other ordering should be feasible. Otherwise there would be a danger of several different meanings for the same program.

Moreover, dependency analysis should also identify sources that do not depend on each other. This can help minimizing the work that is necessary when source code was modified; only files affected by a change will have to be recompiled.

In chapter 4 I show that some features of SML make dependency analysis hard. More than one source file can define the same symbol, and a named structure can be "opened," thus introducing bindings for all its members into the current scope, even though none of the newly bound names are lexically apparent. Dependency analysis is tractable if multiple top-level definitions for the same symbol are ruled out. But if one were to enforce this rule globally, then a definition in one part of the program would prevent definitions for the same name in other, unrelated parts.

It is important to CM's group model that such restrictions be only applied locally, because otherwise they would inhibit modularity (→ Chapter 3). Therefore, redefinitions are tolerated if the original definition was imported from a subgroup or a library. But such leniency causes new problems when naïvely combined with SML's **open** construct, which makes all definitions of a named module directly available in the current scope. In section 4.2, I show that certain uses of **open** can make dependency analysis NP-hard.

Other languages, for example C [KR88] and Ada [Ada80], do not exhibit such problems because they use a different model for separate compilation and require globally unique names.

A C object file (`.o`) depends on its source file (`.c`) and on other files that are included via preprocessor directives (`#include`). Therefore, one must calculate the transitive closure of the "#include" relation to find dependencies for each object. In this case it is not even relevant that C, unlike ML, does not permit multiple top-level definitions for the same name.

An Ada compilation unit depends on other compilation units only if they are named explicitly. Furthermore, names for compilation units are globally unique. Consequently, there is no potential for ambiguities.

In the case of ML, problems arise because one can freely refer to identifiers declared in other compilation units without having to say explicitly where the name is imported from. When combined with **open**, it can become difficult to determine the definition corresponding to a given use of an identifier. Consider:

```
(* 1 *) val x = 1
(* 2 *) structure A = struct
(* 3 *)    val x = 2
(* 4 *) end
(* 5 *) open B
(* 6 *) open A
(* 7 *) val y = x
```

Depending on the contents of structure `B`, variable `x` that is mentioned on line 7 could refer to line 1, to line 3, a member of structure `B`, or even a member of some structure `B.A`.

C lacks a language construct like **open**, and Ada's **use** is unable to override earlier definitions. Under these rules, regardless of `B`'s contents, line 7 would always refer to the definition of `x` on line 1.

ML's approach to separate compilation is more flexible, hence harder to analyze. However, I proposed and implemented two simple restrictions that solve problems with the complexity of dependency analysis. These restrictions fit well into CM's group model and have been found to be entirely reasonable in practice.

1.4 Cross-module optimization

Parnas [Par72] describes the importance of good modularization. He was the first to recognize that programs should be divided along lines of abstraction and not at phase boundaries.

The timeline of a program's execution does not serve well as a guide for determining how it should be structured.

But if we do not divide at phase boundaries, then at runtime there will be many more cross-module function calls. Therefore, when paired with abstraction, separate compilation often incurs large performance penalties. Information hiding helps in the process of software design, but at the same time it can inhibit useful optimization because certain optimizations only work when coordinated across compilation-unit boundaries.

Cross-module optimization is the attempt to circumvent barriers erected for the purpose of modular design, in order to produce more efficient code. Abstract interfaces are "bottlenecks," which reduce the amount of reasoning needed to understand the meaning of a program and to convince ourselves of its correctness. But eventually we want the compiler to "widen" such bottlenecks in order to be able to generate faster executables.

As an extreme measure, one could verify each of the individual components of a program separately, but later remove abstraction boundaries and compile everything as one single large piece. The result is separate type checking, but no separate code generation. In many cases that would not be a viable option. Slightly less aggressive approaches defer important parts of optimization and code-generation until link-time [Fer95b, Fer95a]. In effect, this is a very similar approach, but at least it performs some of the compilation steps early: parsing, type-checking, and intramodular optimizations.

I developed λ-splitting as a technique for automatic cross-module optimizations that is able to treat free variables, nested scopes, higher-order functions, and link-time side effects from module-level initialization code. The automatic cross-module inlining schemes used to date [DH88, CHT91, CMCH92] were unable to do so, or they failed to preserve efficient separate compilation [Sch77, CHT91].

Example: Abstract data types. The symbol type of a compiler is a good showcase example for abstract data types. Appel [App97] uses a definition close to the following:

```
signature SYMBOL =
sig
  type symbol
  val name :  symbol -> name
  val eq :  symbol * symbol -> bool
  val symbol :  string -> symbol
  ⋮
end
```

The idea is that symbols act as representatives of their corresponding name string. However, implementations of type `symbol` may decide to provide additional features, such as very fast comparison for the purpose of speeding up frequent lookup operations in symbol tables.

Therefore, a realization of type `symbol` could be:

```
structure Symbol :> SYMBOL =
struct
  type symbol = string * int
  fun name (s, n) = s
  fun eq ((s1, n1), (s2, n2)) = (n1 = n2)
  fun symbol n = ...
  ⋮
end
```

Under this implementation the function called `name` is equivalent to a simple field selection operator. If we expect that `name` will be invoked frequently, then inline-expanding such function calls can save a lot by removing the associated overhead.

Function `eq` is another example where inline-expanding function invocations can significantly improve running time. In general, we want to remove the overhead of frequent calls to small functions. On average it is less important to do the same for large functions because the relative overhead will be smaller and it is likely that they are called less often. (See Appel's discussion of *leaf procedures* [App97, page 121].)

But abstraction, modularization, and separate compilation usually means that all functions must be called via a generic function call protocol. Furthermore, the abstract interface does not reveal which functions are large and which ones are small. The solution to these problems is cross-module inlining, which is able to automatically bypass abstraction and modularization during code generation without sacrificing their benefits for software engineering.

Example: Datatype representation. The following problem prompted my work on cross-module optimization [App93]:

In SML one can define algebraic types in *signatures* without being fully specific about their components. For example, one can say:

```
signature S = sig
  type t
  datatype l = NIL | CONS of t
end
```

Such an interface specification must then match corresponding structures with arbitrary choices of type `t`. In particular, it should be compatible with structure `A` defined as:

```
structure A = struct
  datatype l = NIL | CONS of int * l
  type t = int * l
  ...
end
```

Note that this type `l` is very similar to SML's predefined `list` type. Figure 1.1 shows how one would like to represent such lists in memory. The compiler has significantly less knowledge about type `l` when it compiles the signature. But it must make decisions about how to represent values of type `l`. These decisions can be based only on information available in every part of the program that is sensitive to `l`'s representation.

One implementation strategy for representing values whose type is unknown is to fully "box" them—to make them fit into one single machine word. Cartesian products, for ex-

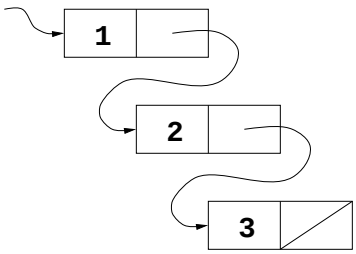


Figure 1.1: **Efficient representation of lists.** One would like to represent each element of an integer list as a box containing the integer and a pointer to the next box. Some immediate value distinguishable from all pointers serves as empty list `NIL`. This figure shows the pointer diagram for `CONS(1,CONS(2,CONS(3,NIL)))`.

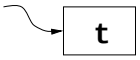


Figure 1.2: **Representing values of abstract type.** If the type of the value carried by a datatype constructor is left abstract, then it becomes necessary to make conservative assumptions about its runtime representation. One commonly used strategy is to "fully box" values of abstract types. They must fit into one single machine word. This figure shows a pointer diagram for `CONS(x)`, where the type of `x` is left abstract.

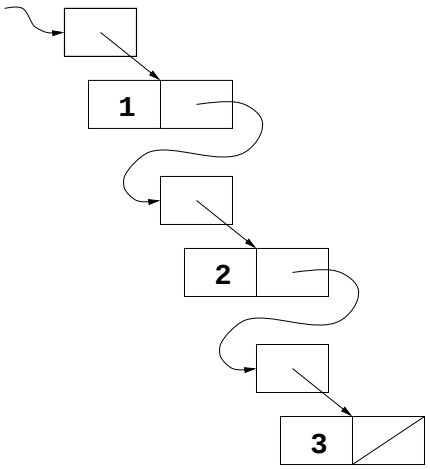


Figure 1.3: **Inefficient list representation.** Actual representations must be compatible with every conservative assumption that the compiler might make about them. Therefore, in order to match the representation shown in figure 1.2, one would have to implement `CONS(1,CONS(2,CONS(3,NIL)))` using double indirections.

ample, would have to be heap-allocated, so a pointer to the heap-record can serve as their boxed representation. If `S` is part of the interface of a generic module (a "functor"), then the code in such a module would expect values of type `t` to be boxed, because one needs to know the representation of constructor `X.CONS`, but type `t` in signature `S` is unknown (→ Figure 1.2):

```
functor F (X: S) = struct
  fun f (X.NIL) = NONE
    | f (X.CONS x) = SOME x
  ...
end
```

Since the functor could potentially be applied to structure `A`, the representation of `A.CONS` must match that of `X.CONS`. This means that there will have to be an extra indirection in its representation. It cannot be implemented as a flat record containing just an integer and a pointer to the next element, because `int * l` must fit in a single machine word (→ Figure 1.3).

But this argument also applies to one of the most pervasive types in ML programs: `'a list`. The formal functor parameter in:

```
functor L (type 'a t
           datatype 'a list = nil
                            | ::  of 'a t) =
struct
  ...
end
```

must match the built-in `list` type, as available from structure `List`:

```
structure X = L (open List type 'a t = 'a * 'a list);
```

Even if no one ever writes the functor, the mere potential for such a functor to be written implies that *all* lists must use an extra indirection. But this is not tolerable, because it incurs at least 50% space overhead and doubles the number of memory accesses when traversing a list.

Therefore, all implementors of ML have used the efficient representation for lists anyway, thus compromising the completeness of their implementation, as they were not able to compile code that contained and applied a functor like `L`. It has only very rarely been the case that users of the compiler were negatively affected by this choice, but these cases did occur.

However, there are many more tricks of how to cleverly optimize the representation of datatypes. The more such tricks are employed, the more likely it becomes for real programs to run into problems with representations that do not match.

These difficulties can be solved using even more abstraction, but that solution can only be practical if efficient cross-module optimizations can eliminate the associated run-time overhead.

More abstraction: Abstractions, such as type `t` in the examples above, make code generation difficult. Surprisingly, the situation can be improved by being even more abstract. Type `l` is represented using three functions: two for construction and one for a combination of pattern matching and deconstruction [HS97]. Signature `S` will then be treated as if it looked like this:

```
signature S = sig
  type l
  type t
  val NIL: unit -> l
  val CONS: t -> l
  val NIL'CONS'match:
        l * (unit -> 'r) * (t -> 'r) -> 'r
end
```

Furthermore, functor `F` will be compiled as:

```
functor F (X: S) = struct
  fun f z =
    X.NIL'CONS'match (z,
      fn () => NONE,
      fn x => SOME x)
  ...
end
```

The representation of values of type `l` is now completely encapsulated in `NIL`, `CONS`, and `NIL'CONS'match`; no other part of the program needs to know. Therefore, it can be chosen arbitrarily at the time the datatype is defined.

But the resulting code seems to be even less efficient. Every constructor and every pattern match has been replaced by a function call. Therefore, such a solution must rely on techniques to eliminate function calls wherever possible. Inlining is such a technique. *And since it is common for datatypes to be defined in separate modules it is necessary to have efficient cross-module inlining.*

Of course, abstract datatypes and type representation are not the only examples that justify the need for cross-module optimizations. A technique that is capable of inline-expanding small functions defined in one module when they are used in another can remove many inefficiencies incurred by abstract design and modularization. The advantages abstraction can have for software engineering are clear; one must eliminate the disadvantages.

λ-splitting. Chapter 5 of this dissertation presents λ-splitting as a technique for automatic cross-module inlining. The approach does not rely on some new kind of optimizer that is capable of reaching beyond module boundaries. Instead, it leverages the power of an existing *intra*modular optimizer.

The λ-splitting transformation transparently moves the boundaries between compilation units—hoping they will then no longer play the role of the narrow "bottleneck" that,

for the purpose of improving separate design and testing, they were designed to be.

The intermediate representation underlying λ-splitting is a λ-calculus with records. λ-calculus [Chu41] serves as a powerful language for expressing program transformations. It is a well-studied formal system with known properties. That was of great help when we developed our technique.

The SML/NJ compiler translates the code of each compilation unit into a closed λ-expression [AM94]. Being able to work with closed expressions allows for a highly modular design of λ-splitting itself. There is no need to consult external symbol tables. Subsequent compiler passes are easily interfaced by passing well-formed, closed λ-expressions to them. Thus, extensive modifications to other phases of the existing compiler were unnecessary.

1.5 Macro systems

A macro system can be used as a programmer-controlled tool for forced inline-expansion. As such, it represents an alternative approach to optimization. But macros can also be used as a meta-programming tool. The application of macros in meta-programming has been most popular in languages of the Lisp-family and renewed its appeal when Scheme's "hygienic" macro systems were developed [Ce91, Koh86, BR88, Han91, CR91, Dyb92].

Traditional macros create problems by violating the rules of the language they are extending. This can happen at many different levels, ranging from failure to respect the integrity of tokens or expressions up to name clashes due to inadvertent name captures.

Hygienic macro systems systematically rename identifiers throughout the entire program. They maintain information about the syntactic role (free, bound, or binding occurrence) of each name introduced by a macro expansion. This enables them to correctly relate variables to their respective bindings and to avoid name clashes.

But existing solutions did not address the problem of separate compilation and linking. Moreover, they were not developed for statically typed languages or for languages that have nested modules whose components are accessed using long identifiers. Long identifiers are sequences of ordinary identifiers corresponding to names of nested modules. In many programming languages they are written using a dot-notation. (For example, `A.B.x` might refer to the definition of `x` in module `B`, where module `B` itself is defined in module `A`.)

The most serious deficiency of macro systems, however, is that they can be used to create obscure binding constructs or to alter the behavior of existing ones. The impact on scoping rules is confusing both to automated analyses and to the human reader who is trying to make sense of some piece of code.

Chapter 6 starts with an overview of problems associated with macro systems and reviews existing solutions to some of them. It then presents a refinement of an algorithm for hygienic macro expansion that lets macros and modules coexist. The same algorithm can be adapted to facilitate correct name resolution during separate compilation and linking.

Nevertheless, I conclude that macro systems as powerful as the ones in existence should not be recommended for use in large-scale programming because they continue to have serious problems in essential areas:

- Macro systems are untyped. Type errors will not be detected until macros are instantiated. But then it can be too late to fix the problem.
- Hygienic macro systems facilitate recursive macros that do not guarantee termination of the expansion process. But macros must be expanded by compilers and analysis tools where potential non-termination is not acceptable.
- Dependency analysis in the style of CM is complicated for languages with macros. Even if all macros could be expanded in constant time, dependency analysis would still be NP-complete.

1.6 Contributions

The contributions of this dissertation are the following:

- Design and implementation of CM, the compilation manager of SML/NJ, as a practical tool for managing large software projects (→ Chapter 2).

- An investigation of problems with modularity caused by the use of global namespaces. I develop the *group* model—a model that does not rely on a global namespace while providing the facilities for controlling name visibility in a hierarchical fashion. The implementation of the group model in CM results in a system where non-modular restrictions on definability of names or availability of definitions can be eliminated effectively. Such restrictions used to be typical for other separate compilation and linking mechanisms (→ Chapter 3).

- A discussion of automatic dependency analyses for SML and their asymptotic complexities. I prove the general problem NP-complete and give two simple language restrictions that avoid the problem with intractability. Automatic dependency analysis has been implemented as part of CM (→ Chapter 4).

- λ-splitting as a technique for automatic cross-module inline expansion and optimization. The technique was designed and implemented as manipulations of programs expressed in λ-calculus. It is capable of correctly dealing with free variables, nested scopes, higher-order functions, and link-time side effects (→ Chapter 5).

- A discussion of hygienic macros and their relationship with module systems and separate compilation. I show that hygienic macro systems are not capable of expressing ML-style modules and long identifiers. But a modification to an existing algorithm allows macros and modules to coexist. Unlike its predecessors, my algorithm also

supports separate compilation. The presentation is concluded by pointing out areas where problems for macro systems continue to persist: types, dependency analysis, and termination (→ Chapter 6).

Chapter 2

The design of CM

CM is the compilation manager for Standard ML of New Jersey [MTH90, AM91]. It is loosely based on its precursor SC [HLPR94b, HLPR94a] and provides functionality similar to the well-known UNIX program **make** [Fel79].

CM simplifies the maintenance of large software systems by subdividing them into a hierarchy of groups. Groups can play the role of software libraries. They consist of individual SML source files and can refer to other groups or other libraries.

CM is tailored to the language SML. It provides automatic dependency analysis and cutoff recompilation. This contrasts with general-purpose compilation management tools such as **make** and **Odin** [Cle94], which target a broader variety of languages and systems, and which, therefore, can only offer services that are equally applicable to all of them.

By using the hooks provided by SML/NJ's "visible compiler" [AM94] CM translates SML sources to produce binary object files called binfiles. But SML source files themselves can be the result of executing other programs. This is accounted for by CM's "tools" mechanism. The version control system RCS, the parser-generator ML-Yacc, or the literate programming tool noweb are examples of such tools [Tic85, TA90, Ram94].

Figure 2.1 shows the dependency graph for CML's sources. CML is John Reppy's

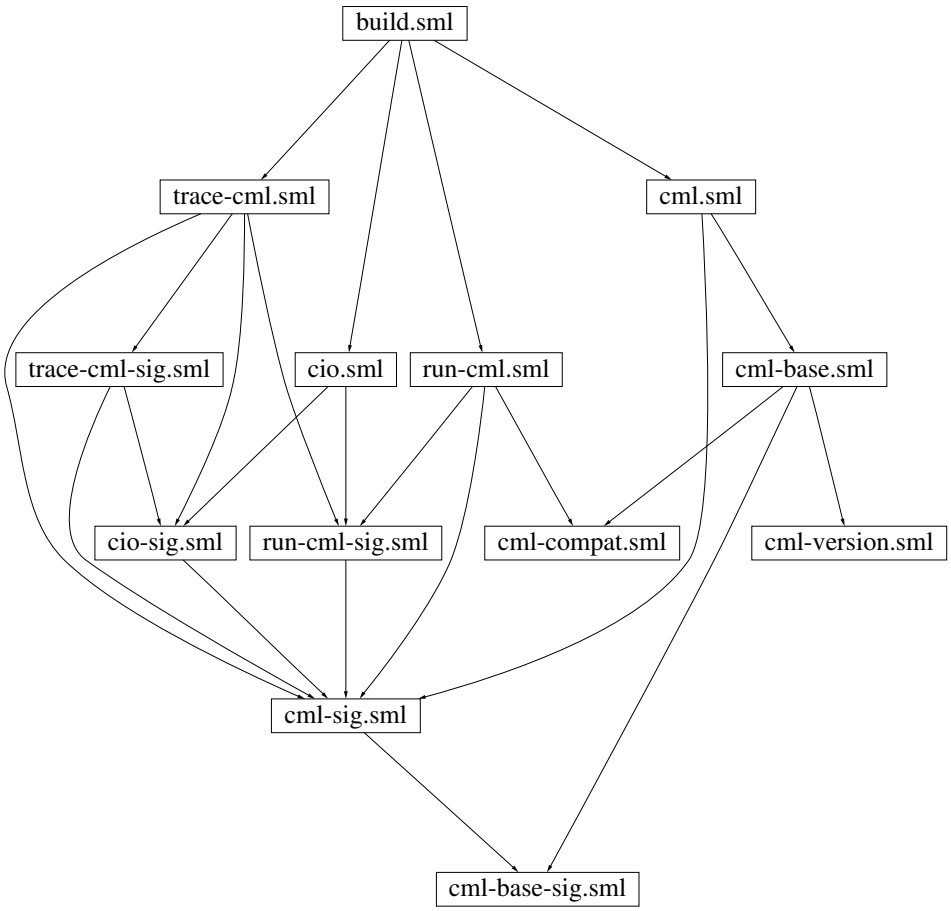


Figure 2.1: **A dependency graph.** Separate compilation can be used to break large programs into smaller pieces and to compile these smaller compilation units one at a time. But in most cases one cannot compile a compilation unit in complete isolation. The result depends on both the source file itself and the result of compiling other sources. Here `trace-cml.sml` depends on `cml-sig.sml`, which is indicated by the presence of directed paths from the former to the latter.

Concurrent ML system [Rep91]. It is implemented as an ordinary ML program on top of SML/NJ. Here I use it as an example for a medium-scale programming project that is managed using CM.

Without the help from a compilation manager one would have to invoke a number of `use` commands, one per source file, in order to load CML into SML/NJ. Of course, the order in which the commands are issued is important, since identifiers can only be used after they have been defined. Therefore, the loading of source code has to proceed in topological order with respect to the dependency graph. Unfortunately, the task of manually maintaining sources in topological order is tedious and error-prone.

But not every source depends on every other source. I hope that in a well-structured project there are relatively few dependencies between source files. Therefore, when some (but not all) source files have been changed, it is usually possible to compile only a small subset of all compilation units.

The UNIX program **make** recompiles all ancestors of a source that has been altered. But this strategy, while correct, can be too pessimistic because changes to one file might not have an actual influence on subsequent compilations of depending units. In contrast, CM first recompiles the source that was modified and then compares the outcome with the result of previous compilations. Recompilation propagates to ancestors in the dependency graph until the original change ceases to have an impact.

Suppose I modify `cml-version.sml` without changing its interface. CM will recompile this file—but no more than that; further work is unnecessary. Presented with the same problem, **make** would also process three other files: `cml-base.sml`, `cml.sml`, and `build.sml`.

It is often the case that modifications propagate for some time without eventually reaching the root of the hierarchy. For example, if I alter one of the declarations in `cml-compat.sml` (→ Figure 2.1), then CM must also recompile its direct predeces-

sors, `run-cml.sml` and `cml-base.sml`. At this point, if the original modification to `cml-compat.sml` is not reflected in definitions exported from either of the two, then it will not be necessary to recompile anything else. Otherwise, for example if one of the exports of `cml-base.sml` changes, then CM will also have to recompile `cml.sml` and consider possible changes to its exports.

2.1 Simple groups

The easiest way of getting started with CM is to treat the entire program as one single group. A group in its most basic form is just a collection of SML source files; the names of these files must be listed in a group description file.

The simplest description files start with two keywords, **Group** and **is**, which are followed by the list of source file names. For example, if the group is to consist of `main.sml`, `table.sig`, and `table.sml`, then one must create a file `sources.cm` containing:

```
Group is
    main.sml (* the application code *)
    table.sig (* interface to 'table' abstraction *)
    table.sml (* implementation of 'table' abstraction *)
```

As shown in this example, it is possible to use SML-style comments within description files.

Once `sources.cm` has been set up, one can start CM. This is usually done by running the command `sml-cm`. That begins an interactive SML/NJ session with CM pre-loaded.

CM is controlled by simple ML function calls. For example,

```
CM.make ();
```

will analyze the dependencies among components of the system, determine a feasible ordering of (re-)compilation steps, and carry them out as necessary.

2.2 Restrictions

Unfortunately, if one does not enforce certain restrictions, then dependency analysis is intractable (→ Chapter 4). Therefore, CM requires that all SML code adhere to the following additional rules concerning top-level definitions:

- No symbol can be defined in more than one source of the same group.
- The use of **open** is not permitted.

It must also be noted that only structures, signatures, functors, and functor signatures are tracked by CM's dependency analysis. Good programming style demands keeping all other definitions inside structures and signatures. This style guarantees that there will be no problem.

The analyzer makes no attempt to take into account the side-effects of running the code that initializes a compilation unit. For example, file `a.sml` might define:

```
(* file a.sml *)
structure A = struct
  val r = ref 0
end
```

Later, the execution of `b.sml` will modify `A`'s reference cell:

```
(* file b.sml *)
structure B = struct
  val _ = A.r := 1
end
```

Suppose `c.sml` also refers to `A.r`. Which value should it see? Both `0` and `1` are legal answers because the relative order of `b.sml` and `c.sml` is not constrained. Therefore, either result would be "correct" in some sense. Without additional information neither one can be preferred over the other:

```
(* file c.sml *)
structure C = struct
  val x = !A.r
end
```

Care must be taken with top-level side effects because CM does not try to address this problem. It is up to the programmer to make sure that the ordering of side-effects is either unambiguous or unimportant.

2.3 Hierarchies

Large programs can be broken into more than one group. These groups can then be arranged into a group hierarchy. This makes it easier to manage large systems because related sources are grouped together. Software reuse is promoted by consolidating generally useful components into libraries. Multiple definitions for the same name are not allowed within the same group, but no such restriction exists for definitions in different groups.

Cycles. Definitions in SML cannot form cycles across module boundaries. In particular, structure `A` from `a.sml` cannot refer to structure `B` in `b.sml`, if at the same time structure `B` refers to structure `A`. This rule is checked and enforced by CM.

A definition of the form `structure A = A` is *not* considered a cycle. CM will treat the use of `A` as a reference to a definition imported from a subgroup.

Groups and subgroups. A group A that is mentioned in the description of some other group B is called a *subgroup* of B. We say B itself is a *client* of A because it *imports* A.

Sources of the client can refer freely to any of the symbols defined within and exported by the subgroup. However, the client can also provide new definitions for any of the subgroup's symbols, thereby masking the original one.

Imported symbols are available for re-export by the client. The exact rules governing re-export will be described when libraries and export filters have been introduced. So far I have only introduced ordinary groups and subgroups that do not have export filters. In this simple scenario all imported definitions, with the exception of those that have been masked, will always be re-exported.

Hierarchical group descriptions. The *members* of a group consist of its constituent sources and of its imported subgroups. Usually, CM will automatically identify description files in its member list by their names. Those that end in `.cm` are treated as names of CM-style group description files. Relative filenames are resolved with respect to the directory that contains the description file.

Suppose `a.sml` and `b.sml` are sources of a group that needs to refer to a subgroup containing `util/c.sml` and `util/d.sml`. In this case one could create a description file `util/sources.cm` containing:

```
Group is
    c.sml
    d.sml
```

and another one, called `sources.cm`, specifying:

```
Group is
    a.sml
    b.sml
    util/sources.cm
```

Tools and member classes. Some members are neither SML sources nor group descriptions. They require special processing before CM's analyzer can understand them. This is done by built-in "tools." For example, an ML-Yacc source file `parser.grm` will be fed to `ml-yacc`, which produces two SML files: `parser.grm.sig` and `parser.grm.sml`.

CM applies tools in cascades where necessary. For example, in place of the grammar

file `parser.grm`, one can use the corresponding RCS archive `parser.grm,v`. CM's RCS tool will first run the `co` command to check out a copy of `parser.grm`, then the ML-Yacc tool will take over to produce `parser.grm.sig` and `parser.grm.sml`, which are finally processed by CM directly.

Ordinarily, tools are picked according to the name of the source file. However, the decision can also be guided by explicit annotations in the description file. Each member name can be followed by a colon and the name of a member class. The default classes that CM knows about are: `Sml`, `CMFile`, `MLLex`, `MLYacc`, `MLBurg`, `RCS`, and `Noweb`. Class names are case-insensitive.

Some of the predefined classes do not refer to tools but to CM's basic functionality. In particular, the classes `Sml` and `CMFile` identify SML source code and group descriptions, respectively. Those are subject to automatic dependency analysis.

Since CM tries to guess the member class based upon the member's name, it will rarely be necessary to specify class names explicitly. The appendix shows the list of recognized file name suffixes and their corresponding class names (→ Table A.4).

The built-in tool box of CM is extensible. It allows for seamless addition of new tools by writing a few lines of Standard ML (→ Appendix A.10). A small number of tools is already predefined (→ Appendix A.5).

Export filters. Groups can have *export filters*. A filter, which is simply a list of symbols, restricts the set of definitions that are exported. Groups with export filters will export definitions for precisely the symbols listed, regardless of whether they are defined in the group itself or in one of the subgroups.

Export filters are useful for adding an interface to an entire set of source files. The interface governs what outside clients can see; the members of the group themselves can still freely refer to each other's exports. Chapter 3 discusses how such summary interfaces

can improve separation between software components and how large-scale software development can benefit from that. Libraries are a special kind of group where export filters are mandatory.

Export filters are specified as lists of symbols. They appear between the keyword **Group** (or **Library**) and the keyword **is**. Since in SML we distinguish between symbols of different name spaces, we must write:

- `structure` *struct-sym* for a structure symbol,
- `signature` *sig-sym* for a signature symbol,
- `functor` *fct-sym* for a functor symbol, and
- `funsig` *fsig-sym* for a functor signature symbol.

Therefore, a group description with export filter could look like this:

```
Group
    structure Table
    signature TABLE
    structure Main
    functor A
    funsig A
is
    main.sml
    a/fct.sml
    a/fsig.sml
    table/sources.cm
    RCS/parser.grm,v
```

Libraries. Groups are either ordinary groups or libraries. In many ways libraries are very similar to groups. They have a list of members, they contain SML sources and members that must be processed by tools, and they can be clients to subgroups. The major difference between ordinary groups and libraries becomes apparent when we look at the rules for name visibility.

If a group imports a symbol from an ordinary group, then the corresponding definition (besides being allowed to be masked) has the same status as any definition that appears directly in one of the sources of the client. In particular, it will be exported to clients of the client unless hidden by the client's export filter.

Symbols imported from libraries can be referred to from within sources of the client and they can be masked, but they will not automatically be re-exported to clients of the client.

But symbols from libraries are also subject to re-export provided they have been mentioned in the client's export filter explicitly.

Here is the precise rule which governs symbol export (→ Chapter 3, Equation 3.7).

A group with export filter exports all symbols listed by its filter regardless of their origin.

A group without export filter exports all symbols defined by sources of the group and all non-masked symbols exported by ordinary subgroups of the group.

If a symbol's definition is not used by any of the clients of a library, then it cannot be accessed at all. CM will ignore the members of libraries whose exported definitions are not accessed; they will not be loaded and they will not be linked with the rest of the system. Therefore, they behave in a way similar to members of libraries for other languages.

Syntax. A library description looks almost exactly like a group description. The only difference is that the initial keyword **Group** must be replaced with **Library**.

Aliases. Description files can also act like a symbolic link and "point to" another description. A file containing

```
Alias name
```

behaves exactly like the description file identified by *name*. To locate the aliased description, CM applies the same rules that are also used to find subgroups of groups. If *name* is a relative pathname, then CM will first try to find it in the directory that contains the alias. Upon failure it then consults an internal search path (→ Section A.8).

Syntax. The full syntax of description files in BNF is:

description-file	→ *group-description* | *library-description* | *alias*
group-description	→ **Group** [*export-filter*] **is** *member-list*
library-description	→ **Library** *export-filter* **is** *member-list*
export-filter	→ *export-symbol* { *export-symbol* }
export-symbol	→ *name-space* `Identifier`
name-space	→ **structure** | **signature** | **functor** | **funsig**
member-list	→ { *member* }
member	→ `Pathname` [**:** `Class`]
alias	→ **Alias** `Pathname`

`Identifier`, `Pathname`, and `Class` are lexical classes consisting of non-empty strings without white space, colons, parentheses, or semicolons. Comments in the style of Standard ML (text between balanced pairs of `(*` and `*)`) or in the style of Scheme (text extending from a semicolon to the end of the line) are permitted. They count as delimiters like white space.

Preprocessor. At the time it reads a description file, CM applies a simple, C-like preprocessor that allows for conditional compilation. However, the syntax does not provide for "definitions." If necessary, symbols must be defined using CM commands. These commands modify an internal preprocessor environment that is maintained by CM.

CM's preprocessor syntax is very similar to that used by the C-preprocessor: Lines starting with # are treated specially:

line → *nonpreprocline*
→ *preproc*
nonpreprocline → line not starting with #
preproc → *if* { *line* } *elif-opt else-opt endif*
→ *error*
if → *beginning-of-line* # **if** *expression end-of-line*
elif-opt → { *elif* { *line* } }
elif → *beginning-of-line* # **elif** *expression end-of-line*
else-opt → [*else* { *line* }]
else → *beginning-of-line* # **else** *end-of-line*
error → *beginning-of-line* # **error** *text end-of-line*

Example:

```
Group is
 a.sml
 b.sml
# if (SMLNJ_VERSION >= 109 || defined(structure SMLofNJ))
 util.sml
# elif (SMLNJ_VERSION < 108)
# error The version of SML/NJ is too old.
# else
 util-workaround.sml
# endif
```

Expressions denote integer quantities; 0 is used for for false and non-zero values for true.

There are four forms of atomic expressions:

1. Integer literals evaluate to the corresponding integer.

2. A symbol evaluates to the value bound to that symbol or to 0 if the symbol is not defined.

3. The expression `defined` (*symbol*) evaluates to 1 if *symbol* is defined, or to 0 if it is not defined.

4. The forms:

   - `defined (signature` *sigid*`)`,

- `defined (structure` *strid*`)`,
- `defined (functor` *fctid*`)`,
- and `defined (funsig` *fsigid*`)`

test to see if the given ML module is defined in the base environment.

Depending on architecture, operating system, and configuration some symbols are predefined (→ Appendix A, Section A.9).

Expressions are formed using a variety of binary operators, all of which are left-associative. Operators are listed with increasing precedence. Those that appear on the same line have equal binding strength:

```
||
&&
== !=
< <= > >=
+ -
/ *
```

Logical disjunction `||` and conjunction `&&` are short-circuiting operations. The unary operators for logical and numerical negation are `!` and −, respectively. Parentheses can be used for grouping.

2.4 Caches, missing sources, and stable groups

CM uses a variety of caches to speed up its analysis and recompilation steps. In-core caches provide fast access to information as long as the CM session is kept running. The ambient file system provides a second level of caches. It is used to remember the results of expensive operations from one session to the next. All cache files are stored in subdirectories of a directory called `CM`, which itself appears in the directory where the corresponding source files are located.

Binfiles. The most important kind of cache is the binfile. It plays the role of the binary object file and enables CM to avoid compiling sources over and over. Aside from storing them in binfiles, compilation results are also kept in main memory. This often reduces file system traffic in ongoing sessions because data is readily available in main memory and does not need to be re-loaded from auxiliary storage.

Binfiles are located in a subdirectory whose name depends on target machine architecture and operating system. For example, the binfile for `u/a.sml` on a Sparc running some form of UNIX is `u/CM/sparc-unix/a.sml.bin`.

Dependency files. Dependency analysis is much less expensive than compilation, but it still has its cost because the dependency analyzer must parse all SML source files. However, only very little of the information from each source file is actually necessary to drive the analysis process (→ Section 4.4). Therefore, CM extracts the important small part and caches it (both in main memory and in the file system). Files used for this purpose are called dependency files. They are stored in a subdirectory `CM/DEPEND`. For example, the dependency file for `u/a.sml` is `u/CM/DEPEND/a.sml`. Write errors on dependency files are ignored. If CM encounters an error while writing such a cache file, then it only keeps the corresponding in-core information.

When a source is missing. Even if some source files are not available, caches often make it possible to proceed with analysis, compilation, and linking. If both the dependency file and the binfile are present, then CM might not need to consult the corresponding source file.

However, there are situations where the source must be consulted regardless of whether the binfile exists or not because the binfile also depends on the environment that was in effect when the source was compiled. This environment is the result of compiling the

source's successors in the dependency graph. Modifications to any of them may require the source to be recompiled as well.

Stable groups. If one does not expect a group to change in the near future, one may decide to *stabilize* it. Examples for stable groups are central libraries, which are installed and maintained by the system administrator.

After a group has been stabilized, one can be sure that binfiles will always be valid. The process of stabilization creates a special version of a dependency file. It is called the stablefile. Stablefiles contain summaries of what would have been stored in per-source dependency files.

The presence of the stablefile indicates that a group is indeed stable. Sources in stable groups do not need to be present. Furthermore, individual dependency files are also no longer necessary, because all dependency information has been summarized in the stablefile.

Much less file system activity is required when dealing with stable groups because fewer files must be opened and read. On computers with comparatively slow access to the file system, this can improve performance considerably. On many systems it will make no difference.

The name of the stablefile is derived from the name of the description file. For example, on a Sparc-based UNIX system the stablefile for a group described by `u/sources.cm` will be `u/CM/sparc-unix/sources.cm.stable`.

Chapter 3

Hierarchical Modularity

We have seen that abstraction and modular design are important for large-scale program development. The previous chapter gave a description of CM, the compilation manager for Standard ML of New Jersey, from the programmer's point of view. The following discussion will explain the problems that led to its design and show a formal development of the group model.

3.1 Division into groups

The reason for dividing large sets of compilation units into groups is the same reason that prompted us to divide large programs into individual compilation units. By subdividing the problem we hope to control its complexity.

Big programs should be structured into groups, subgroups, and libraries, the way operating systems structure their file system into directories with subdirectories. But one can do more. As I will show, groups and libraries can be powerful tools for increasing separation of unrelated parts of the program and for controlling interfaces between parts that are related. This will provide solutions to the problems shown in figures 3.1, 3.2, 3.5, and 3.6.

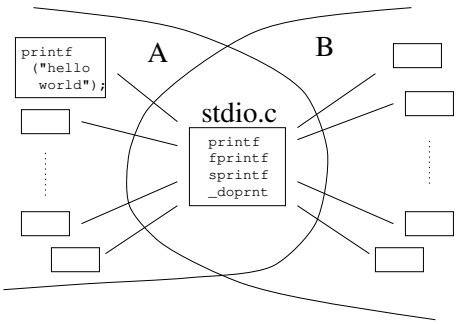


Figure 3.1: **Sharing one source file.** Only a few programming languages let the programmer control the interface of one source file to guarantee that it can be shared between several clients. In C one would declare `_doprnt` as *static*, because local functions should not be exposed. But the C programmer has no way to prohibit `stdio.c` from referring to symbols in system A; these references might prevent it from being used in system B.

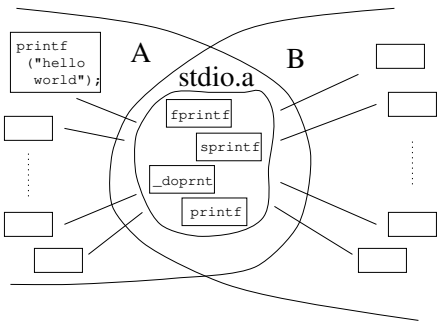


Figure 3.2: **Sharing groups of source files.** Often we want a group of sources to act like one module. An explicit interface placed on `stdio.a` as a whole (as opposed to its constituent components) guarantees the absence of undesired dependencies on either A or B and does not expose local objects like `_doprnt`. The C language has no mechanisms for expressing either of these requirements.

I define *compilation management* in such a system to comprise:

- calculating the dependency DAG between individual source files;
- determining which subset of sources need to be compiled or recompiled;
- determining the order of compilation;
- defining the relationship between source files and program libraries;
- controlling the visibility of globally defined names.

The last two items—involving the management of the namespace—are the important new topics addressed by this work; but name visibility also affects dependency calculation, as I shall show in chapter 4.

Figure 3.1 depicts a situation where one module (`stdio`) is shared by two different projects (A and B). One would like to administer its interface so that the compiler can guarantee the absence of unwanted dependencies on either A or B. Even that is not fully supported by most programming languages. But for large-scale programming one needs to take this further. It is important that `stdio` itself can be a group of source files, and that there can be a summary interface that controls such a group's interface to its various clients (→ Figure 3.2).

The global namespace, which is inhabited by "external" symbols, and which is commonly used by the program linker, poses a problem for modularity. Since there is only one such namespace, any given identifier can only have one definition at a time. Thus, there is a potential for conflicting definitions, which is especially troublesome when the interfering parts of the program are unaware of each other's existence. This is often the case with libraries.

Definitions are said to *interfere* if they cause restrictions on parts of the program that are neither directly nor indirectly related (→ Figure 3.3 and 3.4). A restriction is called *modular* if it only affects related parts of the program.

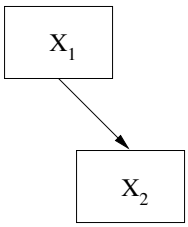


Figure 3.3: **Direct relation.** Two program fragments X_1 and X_2 are directly related if X_2 explicitly refers to a definition that is exported from X_1.

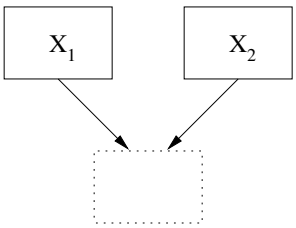


Figure 3.4: **Indirect relation.** Two program fragments X_1 and X_2 are indirectly related if there is a third fragment that explicitly imports from both.

This work's primary concern is to present a framework that avoids all non-modular restrictions on *definability* of identifiers and *availability* of definitions. I will also show how the restrictions that are modular can be resolved by simple local modifications to only the part of the program where they occur.

Definability: A program in a language like C [KR88] cannot use certain identifiers because they are potentially taken by libraries or other parts of the program, even if those are conceptually unrelated. Thus, these identifiers are no longer definable (→ Figure 3.5).

Availability: Programs in other languages, for example SML [MTHM97], can have arbitrarily many definitions for the same name. Although this eliminates restrictions on definability, it creates new restrictions on availability. Parts of a larger program may not be able to see an early definition for variable x because there is a different, intervening definition for x inhibiting access to the one that was intended (→ Figure 3.6).

Because of the global namespace's lack of structure, definitions that are conceptually

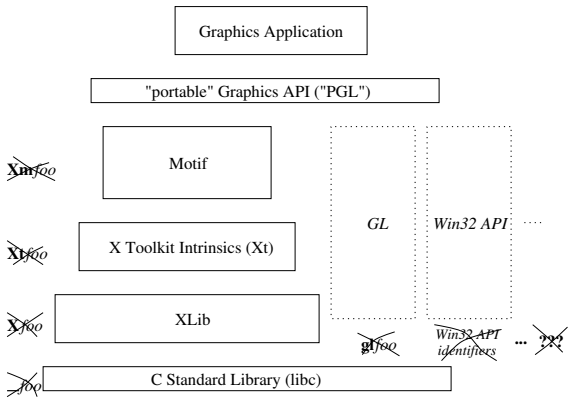


Figure 3.5: **Restrictions on definability inhibit modularity.** The programmer of this graphics application in C, who ideally would like to use the "portable" PGL library abstractly, as a black box, must be careful not to interfere with any of the various libraries upon which PGL might be implemented. Entire classes of identifiers must be avoided: **Xm***xxx*, **Xt***xxx*, and so forth because of X Windows, **gl***xxx* to avoid conflicts with GL, scores of symbols to account for Win32, but also many names that the creators of new library designs *might* choose for their purpose in the future.

local to a small group of sources are often promoted to be globally visible. For example, many implementations of the C standard library export a function `_doprnt`, but the only purpose of this function is to be called by other functions (`printf`, `sprintf`, ...) that are exported from the same library. The application program is not supposed to refer to `_doprnt` directly, which is indicated by the presence of the leading underscore in its name. But "magic" names like this are nothing more than a poor man's solution to the more general problem. It cannot give guarantees of non-abuse, but such guarantees are sometimes necessary when the programming environment is trying to promote safety and security. Of course, one could make `_doprnt` static, but this would require `printf`, `sprintf`, and so forth, all be implemented in the same compilation unit. Neither this approach nor the use of magic names scale well.

Modularity suffers if the programmer who uses a "portable" API of graphics routines, like the fictitious PGL in figure 3.5, has to worry about how it is implemented. With only one single global namespace the programmer must be careful not to use any of the symbols taken by, for example, the X Windows libraries [SGN88], if the program later has to be

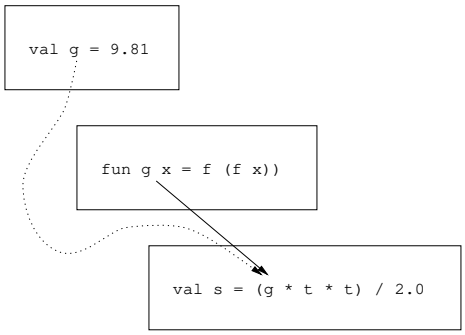


Figure 3.6: **Restrictions on availability inhibit modularity.** In this SML program a value for `s` was meant to be calculated in terms of the gravitational constant `g` that was defined earlier. But a third, conceptually unrelated compilation unit redefines `g`, so the original definition becomes unavailable. In this case the resulting program does not even type-check, and the problem can be detected at compile time. In other cases the program might produce an unintended result.

linked with those. That alone excludes hundreds of identifiers, but the argument extends to all other basis libraries that may also serve as an implementation platform for PGL. To write completely portable code, one would even have to foresee any future development that leads to alternative implementations. This, of course, is impossible.

3.2 The compilation manager CM

CM provides hierarchical modularity through its group model. The description of a group consists of three elements: a set of source files, a set of imported groups, and a list of exported symbols. The minimal design of CM's configuration language ($\rightarrow$ example in figure 3.7) makes it easy to use, especially because built-in automatic dependency analysis eliminates the need for being explicit about order of compilation within each group.

Sources of a group should conceptually belong together. They may correspond to a subcomponent of a larger program or they may act as a program library. If group A needs to refer to definitions exported from another group B, then B must be mentioned in A's description. Names exported by the modules of a group are visible in other modules of the same group, but only those that appear in the list of exported symbols can also be seen from

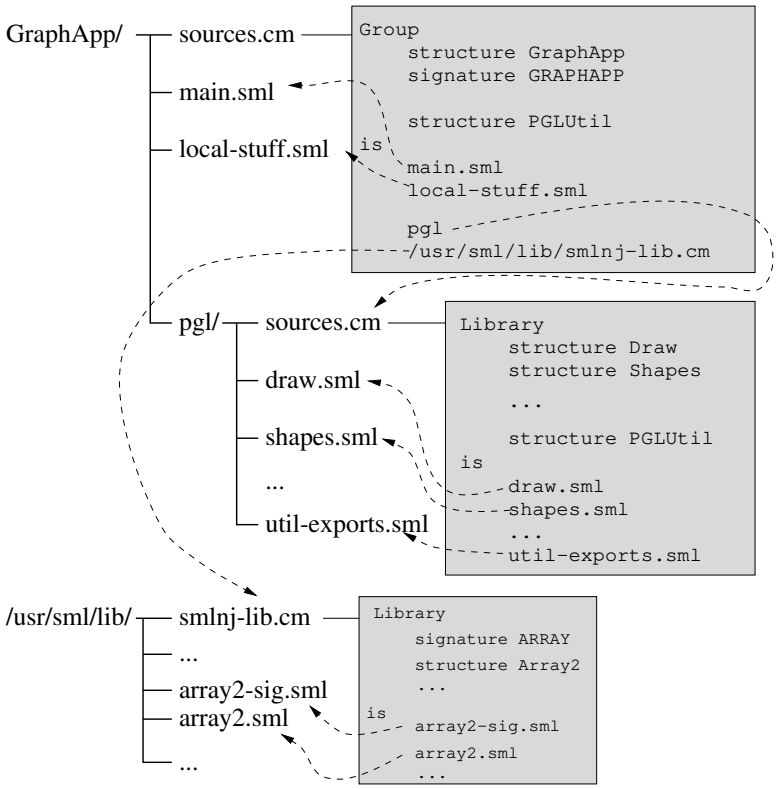


Figure 3.7: **CM group description files.** This figure shows a sketch of a program, its group descriptions, and a sample directory hierarchy to hold the associated files. Directory `GraphApp` contains source files (`main.sml`, `local-stuff.sml`) and the description (`sources.cm`). The application exports symbols `structure GraphApp`, `signature GRAPHAPP`, and `structure PGLUtil`. `PGLUtil` itself is imported from the graphics library "PGL" described by `pgl/sources.cm`. Furthermore, there are imports from the SML/NJ library, which was installed in a central location by the system administrator. (Relative pathnames in description files refer to files in the directory that contains the description.)

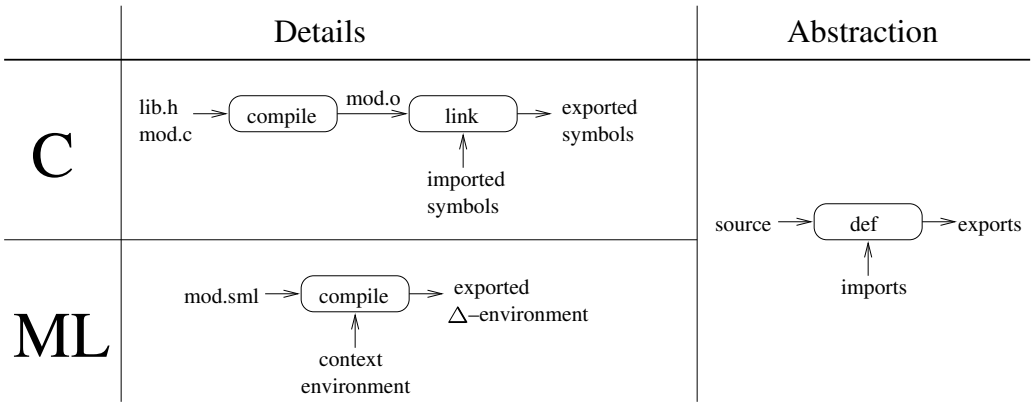


Figure 3.8: **Compiling, linking, context.** A C source file is first compiled and subsequently linked with respect to a table of imported definitions. In the case of SML compiling and linking are combined into one step, but, again, there is a context environment representing definitions that are imported from other compilation units. I abstract from language-specific differences and uniformly use the *def* operator as a model for compiling (and linking) a source with respect to some context.

the outside.

Without CM's groups, the client of PGL in figure 3.5 could not use certain identifiers because they are potentially taken by the libraries that implement PGL. To cope with this, one would implement PGL as a group whose export filter only lists those identifiers that are supposed to be accessible by its clients. Consequently, the application programmer cannot even tell how PGL is implemented; the application code will be truly independent of the libraries underlying PGL's concrete realization.

A group implementing the equivalent of the C standard library would not list `_doprnt` in its export list. No client of that library could then accidentally or voluntarily access the corresponding routine. This matches one's intention of `_doprnt` being local to the library's implementation.

CM's group model is able to eliminate modularity-inhibiting restrictions on definability and availability in a general way. The following section will discuss this more formally.

3.3 Environments and linking

Cardelli [Car97] presents an excellent discussion of the problems that arise with modules and separate compilation. His notion of a linkset is used as a framework for describing and reasoning about consistent, type-safe linking. Type-safe linking, for example provided by SML/NJ's "visible compiler" [AM94], is a prerequisite of my work, but not the focus. In place of Cardelli's linksets my notation uses functions to express operations on environments and equations for describing their properties. This also reflects the actual implementation of CM more closely.

Environments. During separate compilation, individual sources are always compiled with respect to some environment that represents the definitions exported from other compilation units.

Formally, an environment $\rho \in U$ is a partial mapping from long identifiers Ide^+ to denotations D. Long identifiers are non-empty sequences of simple identifiers. They are used to express access to members of a structure by means of a "dot notation" that is found in many programming languages. The notation $Y.z$ stands for a long identifier $\langle id_1, \ldots, id_k, z\rangle$ where the last component $z \in \text{Ide}$ is a simple identifier and where $Y = \langle id_1, \ldots, id_k\rangle$.

$\text{Hd}(I)$ is the head component of a long identifier:

$$\begin{aligned} \text{Hd} &\in \text{Ide}^+ \to \text{Ide} \\ \text{Hd}(\langle id_1, \ldots\rangle) &= id_1 \end{aligned}$$

D depends on the programming language. In SML it would correspond to the compilation unit's static and dynamic semantics. In C, on the other hand, external identifiers stand for machine addresses. In this case D would be a domain of locations. To abstract from such language-dependent issues, I instead use a domain of labels $l \in \text{Lab}$. These labels uniquely

identify each definition of a given program. Thus, I can always distinguish between different bindings for the same identifier without having to consider the meaning of an identifier according to the semantics of the language.

$$D = \text{Lab}$$

The domain $\mathit{dom}(\rho) \subset \text{Ide}^+$ of an environment ρ is the set of identifiers that are bound there. I call $\{\text{Hd}(x) \mid x \in \mathit{dom}(\rho)\}$ the head domain $\mathit{dom}_H(\rho) \subset \text{Ide}$ of ρ. $\emptyset_U$ is the environment with an empty domain.

All prefixes of long identifiers that are bound by an environment must also be bound by the same environment:

$$Y.z \in \mathit{dom}(\rho) \Rightarrow Y \in \mathit{dom}(\rho)$$

Environments can be combined using the $+$ operator:

$$\begin{aligned} + &\in U \times U \to U \\ \rho_1 + \rho_2 &= \lambda x. \begin{cases} \rho_1(x);\ \text{Hd}(x) \in \mathit{dom}_H(\rho_1) \\ \rho_2(x);\ \text{otherwise} \end{cases} \end{aligned} \tag{3.1}$$

The operator $+$ is associative, but is not commutative if $\mathit{dom}(\rho_1) \cap \mathit{dom}(\rho_2) \neq \emptyset$.

Compiling and linking. I use the *def* operator (→ Figure 3.8) as an abstraction of compiler and linker. It calculates an incremental delta environment containing just the bindings corresponding to definitions that are explicit in the compilation unit:

$$\mathit{def} \in \text{Source} \times U \to U$$

Programming languages differ in how they calculate the input environment for *def*. Con-

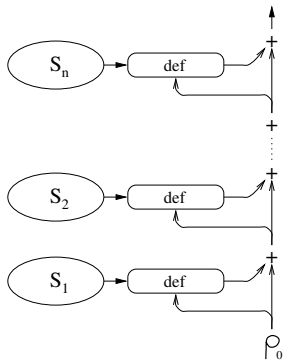


Figure 3.9: **Compilation environments for SML.** In SML the context environment that is used for compiling a source is built incrementally by layering the exports of sources that were compiled earlier on top of the initial basis environment ρ_0.

sider a program consisting of sources $S_1, \ldots, S_n$ written in SML. The export environment ρ_i of source S_i is

$$\rho_i = \mathit{def}(S_i, \rho_{i-1} + \cdots + \rho_1 + \rho_0)$$

where ρ_0 is the initial basis environment. This situation is depicted in figure 3.9.

In C, on the other hand, every source file is linked in the context of the same global environment. The global environment is constructed by combining the exports of all sources ($\rightarrow$ Figure 3.10):

$$\begin{aligned} \rho_i &= \mathit{def}(S_i, \rho); \qquad \forall i \in \{1, \ldots, n\} \\ \rho &= \rho_n + \cdots \rho_1 + \rho_0 \end{aligned}$$

The process of linking corresponds to solving the system of simultaneous equations. In C none of the identifiers can be bound by more than one environment ρ_i:

$$i \neq j \Rightarrow \mathit{dom}(\rho_i) \cap \mathit{dom}(\rho_j) = \emptyset$$

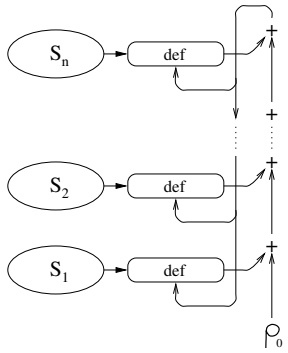


Figure 3.10: **Compilation environments for C.** Conceptually, every source of a C program is compiled with respect to the same global environment, which itself is constructed by layering the exports from all sources on top of some initial basis environment ρ_0. Implementations resolve the circularity at link-time; the compiler takes information from header files and uses it as an approximation of the global environment.

Because of the resulting commutativity of $+$, linking is order-independent and becomes straightforward. However, with each source the programmer must provide header files containing the information necessary to construct an incomplete version of ρ that is suitable for compiling the source, because the full ρ only becomes available after all sources have been compiled. This is sometimes rather cumbersome, but unlike SML it allows for mutual recursion across compilation units.

Linking subsets of sources. I define:

$$\mathit{multidef}_{\mathit{lang}} \in 2^{\text{Source}} \times U \to U$$

to be the language-dependent extension of *def* to sets of sources.

C: The $\mathit{multidef}_{\text{C}}$ operator passes its context argument to individual calls to *def* for each of the constituent sources. The resulting delta environments are combined using $+$, thus yielding a delta environment for the entire subset ($\rightarrow$ Figure 3.11):

$$\mathit{multidef}_{\mathbf{C}}(\{S_1, \ldots, S_n\}, \rho) \quad = \quad \mathit{def}(S_1, \rho) + \cdots + \mathit{def}(S_n, \rho)$$

SML: $\mathit{multidef}_{\text{SML}}$ must first use a dependency analyzer $\mathcal{A}$ ($\rightarrow$ Chapter 4) to turn the set of sources into a sequence:

$$\mathcal{A} \in 2^{\text{Source}} \times U \rightarrow \text{Source}^*$$

The *step* function then incrementally builds the resulting export environment ρ'. Intermediate values for ρ' are layered on top of the initial context ρ to serve as the context needed for compiling individual sources of the set:

$$\begin{aligned}
\mathit{step}(\langle\rangle, \rho, \rho') &= \rho' \\
\mathit{step}(\langle S_i, S_{i+1}, \ldots\rangle, \rho, \rho') &= \mathit{step}(\langle S_{i+1}, \ldots\rangle, \rho, \mathit{def}(S_i, \rho' + \rho) + \rho') \\
\mathit{multidef}_{\text{SML}}(s, \rho) &= \mathit{step}(\mathcal{A}(s, \rho), \rho, \emptyset_U) \qquad (3.2)
\end{aligned}$$

Exports from each source are combined to eventually form the exports of the entire sequence; a second layering operation per source is necessary to form the corresponding compilation context ($\rightarrow$ Figure 3.12). This explains why there are two occurrences of $+$ in the definition for *step*.

3.4 Definability and availability

To show how external definitions in different sources of a program interfere with each other, I investigate the notions of definability of identifiers and availability of definitions.

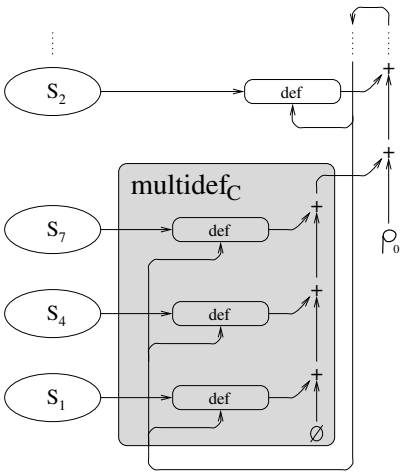


Figure 3.11: **Linking subsets of C sources.** The *multidef*$_{\mathrm{C}}$ operator is the C-specific extension of *def*. It calculates the export environment for a given subset of sources. *multidef*$_{\mathrm{C}}$ returns an incremental delta environment that only binds symbols which are explicitly defined in that subset of sources. This is consistent with the way *def* works for single sources.

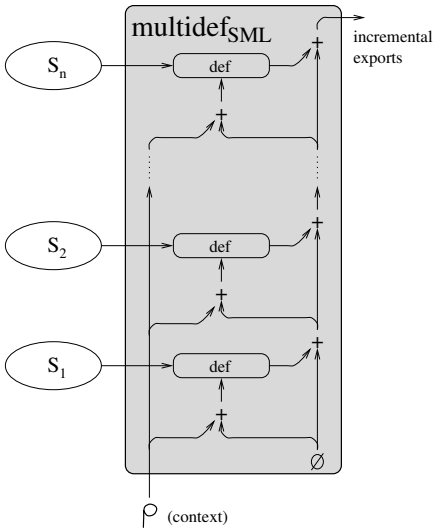


Figure 3.12: **Compiling (ordered) sets of SML sources.** The *multidef*$_{\mathrm{SML}}$ operator extends *def* to sets of SML sources. Individual exports are incrementally layered on top of an initially empty environment. The result is a delta environment only containing bindings for identifiers defined in $\{S_1, \ldots, S_n\}$, which is analogous to the behavior of *def*.

An identifier x is said to be *definable* in source S if it is legal for S to export a binding for x. Furthermore, as a result of how I have chosen the domain of denotations D it becomes straightforward to define availability. The definition that binds identifier x at label l is *available* in environment ρ if $\rho(x) = l$.

Restrictions on definability or availability are not modular. The purpose of CM's group model is to curb undesired long-range effects of global definitions in order to eliminate such modularity-inhibiting restrictions.

Example: C. All external definitions of a C program are available everywhere, but an identifier x is definable in S_i only if it is not defined anywhere else:

$$x \text{ definable in } S_i \text{ iff}$$
$$\forall j : j \neq i \Rightarrow x \notin \mathit{dom}(\rho_j)$$

Thus, in C a definition in one source file affects definability and availability in the entire program. This creates an unfortunate implicit coupling between compilation units that is not modular because it means that the programmer has to be aware of every identifier in every part of the entire program, including all of the program libraries that it could be linked with.

Example: SML. The treatment of SML code requires a well-defined ordering among the sources; S_1 will be compiled before S_2, and so on. Unlike in C, every identifier is definable everywhere,[1] but a definition in source S_i that binds x is available in S_j only if $i < j$ and there is no other definition for x in one of the sources "between" S_i and S_j:

[1]This is a slight oversimplification, because the declaration `val y = x` does not provide a new definition for `y` if `y` is currently a constructor tag for some datatype.

$$\text{definition for } x \text{ from } S_i \text{ available in } S_j \text{ iff}$$
$$\forall k : (i < k) \wedge (k < j) \Rightarrow \text{Hd}(x) \notin \mathit{dom}_H(\rho_k)$$

Other than in C, in SML definitions do not interfere with the definability of names. They still have a non-local impact on availability because a definition for x makes previous definitions for x unavailable. The programmer has to be aware of all the definitions exported from *earlier* sources. Again, the resulting implicit coupling inhibits modularity.

Availability is sensitive to the order of compilation, which is especially troublesome when the order is chosen automatically and not by the programmer. But problems only arise because a module can redefine symbols that are already defined. That not only makes the code difficult to read for a human—it also makes dependency analysis NP-hard (→ Chapter 4).

3.5 Groups

Naturally, the part of a program that sees variable x bound to l_1 cannot simultaneously see another definition that binds x to l_2. But at the time when the code mentioning x was written, the programmer certainly had a specific idea about which binding x is supposed to refer. To make sure the right thing happens, it is necessary to state one's intentions explicitly. Compilation management can enforce them.

Problems arise when different, unrelated definitions for x in different portions of the program interfere with each other. As discussed earlier, this is caused by the fact that a single global environment represents a universal container that is inhabited by all top-level definitions. The SML compiler builds this container incrementally, but that does not really help because no matter what the order of two definitions binding the same identifier may

be, one of them will make the other unavailable.

Groups localize the effects of definitions. If a source in group G_2 refers to a definition exported from another group G_1, then the description of G_2 must name G_1 explicitly as one of its imports. The resulting explicit export-import relation between groups can be used to establish rules for removing unwanted interference between definitions in "unrelated" groups.

Two groups are directly related if one explicitly imports from the other. They are indirectly related if a third group imports definitions from both. In any other case I call them "unrelated" ($\rightarrow$ Figure 3.3 and 3.4).

Formally, a group $G \in \text{Grp}$ is a triplet (s, i, e). Here, s is the set of sources, i is the set of imported groups, and e is a set of identifiers that is used for thinning the group's export interface.

$$\text{Grp} \subset 2^{\text{Source}} \times 2^{\text{Grp}} \times 2^{\text{Ide}}$$

Given a set of groups $g \subset \text{Grp}$ let $\mathcal{I}(g)$ be the *import set* of g.

$$\mathcal{I}(g) = \{G \mid \exists (s, i, e) \in g : G \in i\}$$

The *cumulative import set* $\mathcal{I}^*(g)$ is the transitive closure of the import set. The graph of direct dependencies must be acyclic. Therefore, no group can be in its own cumulative import set.

Thinning can be understood as a filter operation $\mathcal{F}$ applied to an environment. The filter retains bindings to only those long names that start with a simple identifier in e; the filtered environment $\mathcal{F}(\rho, e)$ is ρ with its head domain restricted to e.

$$\mathcal{F} \quad \in \quad U \times 2^{\text{Ide}} \rightarrow U$$

$$
\begin{aligned}
\mathit{dom}_H(\mathcal{F}(\rho, e)) &= \mathit{dom}_H(\rho) \cap e \\
\mathrm{Hd}(x) \in e &\Rightarrow \mathcal{F}(\rho, e)(x) = \rho(x)
\end{aligned}
$$

For example, the C standard library would list `printf`, `sprintf`, and so forth in its export list, but `_doprnt` would be omitted.

Let ρ_0 be the initial basis. $\mathcal{C}(i)$ is the context environment that is used when compiling the set of sources s of a group (s, i, e). It is defined in terms of the group's imports i.

$$
\begin{aligned}
\mathcal{C} &\in 2^{\mathrm{Grp}} \rightarrow U \\
\mathcal{C}(i) &= (\sum_{I \in i} \mathcal{E}(I)) + \rho_0
\end{aligned} \tag{3.3}
$$

$\mathcal{E}(G)$ is the export environment of group G:

$$
\begin{aligned}
\mathcal{E} &\in \mathrm{Grp} \rightarrow U \\
\mathcal{E}(s, i, e) &= \mathcal{F}(\mathit{multidef}_{\mathrm{SML}}(s, \mathcal{C}(i)) + \mathcal{C}(i), e)
\end{aligned}
$$

A group (s, i, e) can re-export part of its own context $\mathcal{C}(i)$. For this, the definition to be re-exported must be named in e and cannot be redefined by any of the sources $S \in s$.

I have chosen to consider unordered sets of imported groups. But the summation in equation 3.3 is order-independent only if the domains of the imported environments are disjoint. Therefore, I require that indirectly related groups do not export definitions for the same identifier.

$$\forall(s, i, e) \in \text{Grp} : (\{I_1, I_2\} \subset i \Rightarrow$$
$$dom(\mathcal{E}(I_1)) \cap dom(\mathcal{E}(I_2)) = \emptyset \tag{3.4}$$

3.6 How interferences can be resolved

An implementation of the group model immediately removes the vast majority of unwanted interferences between definitions in large programs because definitions can only interfere when their respective environments are joined using $+$. With groups, most environments never meet each other that way.

A very important property of the group model is that any remaining interferences can always be resolved locally by modifying only the group where they cause a problem. Three situations can arise:

1. The export environments for two sources in the same group contain definitions for the same identifier. These environments are joined using $+$ according to equation 3.2, which defines $multidef_{\text{SML}}$.
2. A source provides a definition for an identifier that is also defined by one of the imported groups.
3. In violation of equation 3.4 two imported groups independently provide definitions for the same identifier.

Interferences of the first kind can always be removed by locally changing one of the offending sources.

The second kind, a clash between a definition obtained from one of the imported groups and a definition in one of the sources, is legal and has a well-defined meaning. Definition

from the group's sources override imported definitions (→ Equation 3.3). Sometimes, when this is not what was intended and the situation still cannot be resolved by a simple change to one of the group's sources, it becomes necessary to rename at the point of import the identifier that is used to access a particular binding from an imported group. This technique does not require changes to the exporting group, and it can also be used to resolve clashes of the third kind.

Renaming can be expressed as yet another operation on environments. First, I show how to rename a long identifier:

$$
\begin{aligned}
R &\in \text{Ide}^+ \times \text{Ide} \times \text{Ide} \to \text{Ide}^+ \\
R(Y.z, x, y) &= R(Y, x, y).z \\
R(\langle x \rangle, x, y) &= \langle y \rangle \\
R(\langle z \rangle, x, y) &= \langle z \rangle; \qquad x \neq z
\end{aligned}
$$

Renaming in environments can then be defined in terms of identifier renaming.

$$
\begin{aligned}
\mathcal{R} &\in U \times \text{Ide} \times \text{Ide} \to U \\
\mathcal{R}(\rho, x, y)(z) &= \rho(R(z, y, x))
\end{aligned}
$$

In general, to formally describe renaming one must extend the notion of groups. Imported groups (→ domain equation 3.5) are described by regular groups and an arbitrary number of identifier pairs. The pairs specify renaming operations (→ Equation 3.6).

$$
\begin{aligned}
\text{Imp} &= \text{Grp} + \text{Imp} \times \text{Ide} \times \text{Ide} && (3.5)\\
\text{Grp} &= 2^{\text{Source}} \times 2^{\text{Imp}} \times 2^{\text{Ide}} \\
\mathcal{C}(i) &= (\sum_{I \in i} \mathcal{E}'(I)) + \rho_0 \\
\mathcal{E}(s,i,e) &= \mathcal{F}(\mathit{multidef}_{\text{SML}}(s, \mathcal{C}(i)) + \mathcal{C}(i), e) \\
\mathcal{E}'(G) &= \mathcal{E}(G) \\
\mathcal{E}'(I,x,y) &= \mathcal{R}(\mathcal{E}'(I), x, y) && (3.6)
\end{aligned}
$$

However, CM currently only implements the original model without renaming because in the case of SML one can get the same effect by using "administrative" groups where necessary. The administrative group imports a binding under one name and exports it under a different one. This is possible because in SML renaming can be expressed in source language (→ Figure 3.13). A definition of the form `val y = x` or `structure B = A` establishes `y` to be an alias of `x` and `B` to mean the same as `A`. (Unfortunately, this is not properly reflected in my formalism because of my choice of labels as placeholders for denotations.)

Other languages, for example C and Scheme, do not provide a general way of defining one name to be an alias for another. A variable definition in these languages creates a new, unique meaning. In these cases a compilation manager like CM would have to implement renaming directly. It should also be noted that even in SML one cannot always create aliases that are truly indistinguishable from their original. For example, variable `y` in `val y = x` will not have the status of a constructor even if `x` was a constructor. However, at the moment this is not an issue because CM does not deal with type or value definitions at top level. Only structures, signatures, and functors are tracked.

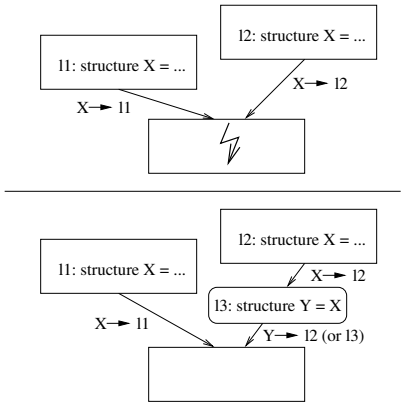


Figure 3.13: **Administrative groups.** In SML one can achieve the effect of renaming upon import by administrative groups because the source language provides facilities of defining one identifier to be a proper alias of another. Many other languages, including C and Scheme, lack such a language feature. For those languages it is necessary to build renaming into the group model.

3.7 Groups vs. modules, namespaces, and packages

Some programming languages offer facilities to help fight the pollution of the global namespace. SML has structures and functors, Modula-3 [CDG+89] has modules, C++ [ES90] has namespaces, and Java [AG96] offers packages.

The main difference in comparison to CM's groups is that these facilities are expressed in source language; modules, structures, namespaces, and packages have names themselves! Consequently, although less likely, there is always the potential for these names to cause the same problems they were supposed to solve by removing other names from the global namespace.

SML structure names can clash, C++ namespace names can interfere with one another, and Modula-3 modules may be in conflict due to an unfortunate choice of module names. Java package identifiers are designed to be globally unique, but since clients of a package must name it in their sources, it now becomes impossible to re-link such a client with a different version of the same conceptual package (perhaps for the purpose of debugging or profiling) without first modifying and recompiling the client's source.

Of course, CM groups must also be named. The current implementation uses the name of the description file for this purpose. The crucial distinction is that the group name never appears in source code. It only appears in other group description files where it can be adjusted without causing recompilation. This way naming remains flexible. Groups can be moved, renamed, or replaced freely without having to touch the program's source code.

3.8 Realization in CM

CM implements the group model without built-in renaming. Therefore, conflicts involving definitions imported from subgroups must be resolved using administrative groups. Since such cases are relatively rare, this has not been a burden so far.

The specification for a group must be written by the programmer. Group description files consist of two main parts: the list of exported identifiers and the list of members. Each member is either a source filename or the name of another description file, in which case the corresponding group will become an imported group.

There are two types of groups: "ordinary" groups and libraries. Ordinary groups do not require an explicit export list. If it is omitted, then CM will provide a default. The group then exports everything defined by its own sources and everything that is imported from sub-groups (but not from libraries). Formally, let $(s, i, \cdot)$ be a group and $i_G \subseteq i$ be the set of imported groups that are ordinary groups themselves. The export list $\hat{e}$ provided by CM is calculated as:

$$\hat{e} = \mathit{dom}(\mathit{multidef}_{\mathrm{SML}}(s, \mathcal{C}(i)) + \sum_{I \in i_G} \mathcal{E}(I)) \tag{3.7}$$

As described earlier, CM always compares the context environment (→ Equation 3.3) to the one used when the same source was previously compiled. If the source itself has not

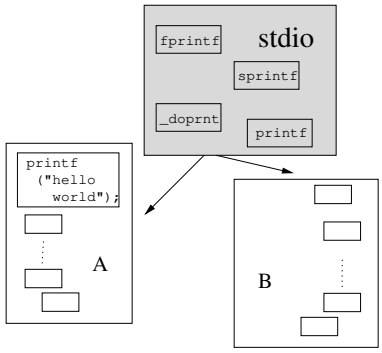


Figure 3.14: **Controlling sharing using groups.** With a compilation manager like CM one can structure the project shown in figure 3.2 into a collection of groups. The export interface on group `stdio` will make sure that implementation details, such as the local function `_doprnt`, are not exposed to its clients, A and B. At the same time, since `stdio` is specific in not listing either A or B in its imports, it cannot depend on either one of them.

been modified, then recompilation can be avoided in the case that the compilation context is unchanged. This is an important optimization.

Figure 3.14 illustrates how CM's groups can be used to solve modularity problems of the type discussed at the beginning of this chapter. Recall that the `stdio` component (shown in figure 3.2) was supposed to get a summary interface. Furthermore, there was concern about potential dependencies of this component on implementiation details in client projects A and B. With CM one can implement `stdio` as a group. To be able to use its services, clients like A and B must explicitly list that group as an import. No client can depend on implementation details that are not explicitly advertised by the group's export filter.

Moreover, the subgroup relationship is unidirectional. The `stdio` group does not (and cannot) mention either A or B as part of *its* imports. Of course, the design of a group can cater to specific needs of selected clients. Less preferred customers must put up with this. But no subsequent modification to any one client can cause incompatibilities with others if they were compatible before.

Chapter 4

Automatic dependency analysis

The minimal design of CM's configuration language (→ example in Figure 3.7) makes it easy to use, especially because built-in automatic dependency analysis eliminates the need for being explicit about order of compilation within each group.

The problem of dependency analysis for SML is the following: For a given set of sources $s = \{S_1, \ldots, S_n\}$ and a context $\rho_0 \in U$ we are looking for a sequential arrangement $\vec{s} = S_{p_1}, \ldots, S_{p_n}$ of the same sources, such that all variables are defined before they are used:

$$\begin{aligned} x \quad &\in \quad \mathit{dom}(\rho_{k-1} + \cdots + \rho_0) \\ &\qquad \text{where } \rho_i = \mathit{def}(S_{p_i}, \rho_{i-1} + \cdots + \rho_0) \end{aligned} \tag{4.1}$$

The sequence $\vec{s}$ implies a total order $\prec$ on s. We call this order *feasible*.

Thus, in a context ρ the dependency analyzer $\mathcal{A}$ takes a set of sources s and forms a

sequence $\vec{s}$ of these sources:

$$\mathcal{A} \in 2^{\text{Source}} \times U \to \text{Source}^*$$

Equation 3.2 shows how dependency analysis is incorporated into our group model.

By equation 4.1 the definition for x that is referred to in S_{p_k} can be provided by either the context ρ_0 or by one of the other sources: $x \in \mathit{dom}(\rho_0) \vee \exists j : (j < k \wedge x \in \mathit{dom}(\mathit{def}(S_{p_j}, \rho_{j-1} + \cdots + \rho_0)))$. However, this is not sufficient: Let x be a long identifier of the form $Y.z$, referring to a definition in structure Y. Another source S' could define structure Y without defining $Y.z$. But according to equation 3.1, definitions for Y that do not define $Y.z$ make previous definitions for $Y.z$ unavailable.

Without structure definitions it is relatively straightforward to implement dependency analysis. Unfortunately, this is of no help if one wants to deal with languages that have module systems similar to SML.

If structures are permitted, then there are two aspects of SML that complicate dependency analysis: symbols can be defined at top level in more than one source and structures can be "opened" at top level. We will see that either feature independently makes the problem intractable. Therefore, I will address them separately.

4.1 Multiple definitions

Multiple definitions of the same symbol at top level in separate sources can introduce ambiguities into the association between uses of a variable and the corresponding definition.

For example, consider three sources S_1, S_2, and S_3:

S_1	S_2	S_3
`val x = 1`	`val x = 2`	`val y = x`

In this simple example it is not hard to spot the problem. If we prove that ambiguities can always be detected easily when they exist, then the detection algorithm can be built into the dependency analyzer, and ambiguous specifications could be rejected gracefully. However, the situation is more complicated:

Claim 1 *With multiple top-level definitions for the same identifier the problem of finding a feasible ordering is NP-complete.*

Proof. The problem is in NP because one can simply pick some ordering and check its feasibility. This can obviously be done in polynomial time.

To prove the problem NP-hard, I use a reduction from the satisfiability problem (SAT), more specifically 3-SAT: For any formula in conjunctive normal form with n variables $v_1, \ldots, v_n$ and m clauses $c_1, \ldots, c_m$, where each clause contains exactly three literals, there is a set of $2n + 2$ sources, for which a feasible ordering corresponds to a satisfying truth assignment. These sources are named: $\tilde{S}, S_1, S'_1, \ldots, S_n, S'_n, \hat{S}$.

Let me first illustrate the main idea of the construction. Suppose $\{v_1, v_2\}$ is a clause with two variables. Each variable will be represented by two structures in two corresponding sources. For v_1 there are sources S_1 and S'_1 with:

S_1	S'_1

`structure A =`
`struct`
`  val x = 1`
`end`
	`structure A =`
	`struct`
	`  (* empty *)`
	`end`

S_2 and S'_2 are constructed according to similar principles, but instead of declaring an empty structure `B`, S'_2 now contains a reference to structure `A`.

S_2	S_2'
`structure B = struct val x = 1 end`	`structure B = A`

Now consider the variable `B.x`. It can be defined either because S_2 was compiled after S_2' or because S_2' was compiled after S_2 and, at the same time, S_1 was compiled after S_1'. Thus, the relative orders of S_i and S_i' plays the role of boolean switches. They model the behavior of the boolean variables v_i. The availability of `B.x` corresponds to the truth value of the entire clause.

This construction can be extended easily to handle negative literals. Suppose the clause was $\{v_1, \bar{v_2}\}$. In this case the contents of S_2 and S_2' must be exchanged. Three or more literals can be accounted for by adding more sources and structures. For a three-literal clause $\{v_1, v_2, v_3\}$ one would add S_3 and S_3':

S_3	S_3'
`structure C = struct val x = 1 end`	`structure C = B`

The complete construction uses multiple versions of structures `A`, `B`, and `C` to handle more than one clause. Additional definitions, a "header" source $\tilde{S}$, and a sentinel $\hat{S}$ constrain the system so that only the relative order of S_i and S_i' (for each i) remains unrestricted.

Construction: $\tilde{S}$ contains:

```
val z0 = 0
val z'0 = 0
structure X1 = struct end
⋮
structure Xn = struct end
```

Note that subscripted names like $\texttt{z}_0$ are used as a meta-notation to specify the SML symbols that need to be generated and inserted. $\hat{S}$ is given as:

```
val z = z_n + z'_n
val c_1 = C_1.x
⋮
val c_m = C_m.x
val y = Y_1.x + ··· + Y_n.x
```

No clause may contain both v and $\bar{v}$ for any variable v, but that is not a restriction because such clauses are always satisfied and may therefore be ignored.

Let $\{c_{k_1}, \ldots, c_{k_j}\}$ with $k_1 < \cdots < k_j$ be the set of clauses where the variable v_i occurs (either directly or in negated form $\bar{v}_i$). The corresponding sources S_i and S'_i will then have the following general form:

S_i	S'_i
`val z_i =`	`val z'_i =`
`  z_{i-1} + z'_{i-1}`	`  z_{i-1} + z'_{i-1}`
`structure Y_i = X_i`	`structure Y_i = X_i`
`structure X_i =`	`structure X_i =`
`struct`	`struct`
`  val x = 1`	`  val x = 1`
`end`	`end`
X_{k_1}	X'_{k_1}
⋮	⋮
X_{k_j}	X'_{k_j}

The X_k are chunks associated with the clauses in which v_i appears. In particular, a clause $c_k = \{x_{l_1}, x_{l_2}, x_{l_3}\}$ with $x_l \in \{v_l, \bar{v}_l\}$ and $l_1 < l_2 < l_3$ is represented by a combination of chunks of SML constructs in sources S_{l_1}, S'_{l_1}, S_{l_2}, S'_{l_2}, S_{l_3}, and S'_{l_3}. Suppose all three literals in c_k are positive. The following table gives the chunks of SML code for each of the respective files. However, if literal x_l is negative, then the corresponding chunks for S_l and S'_l must be exchanged:

Source	Chunk

S_{l_1}	`structure A`$_k$ `= struct`
	`  val x = 1`
	`end`

S'_{l_1}	`structure A`$_k$ `= struct` `end`

S_{l_2}	`structure B`$_k$ `= struct`
	`  val x = 1`
	`end`

S'_{l_2}	`structure B`$_k$ `= A`$_k$

S_{l_3}	`structure C`$_k$ `= struct`
	`  val x = 1`
	`end`

S'_{l_3}	`structure C`$_k$ `= B`$_k$

The constraints imposed by definitions for $\texttt{z}_0, \ldots, \texttt{z}_n, \texttt{z'}_0, \ldots, \texttt{z'}_n$, and z restrict any feasible ordering to one where S_i and S'_i precede S_j and S'_j whenever $i < j$, so S_k and S'_k are adjacent for all k. $\tilde{S}$ is the least and $\hat{S}$ is the largest element:

$$\tilde{S} \prec \left\{ \begin{array}{c} S_1 \\ S'_1 \end{array} \right\} \prec \cdots \prec \left\{ \begin{array}{c} S_n \\ S'_n \end{array} \right\} \prec \hat{S}$$

The definitions for the structures $\texttt{X}_1, \ldots, \texttt{X}_n, \texttt{Y}_1, \ldots, \texttt{Y}_n$ and the variable y guarantee that for each $i \in \{1, \ldots, n\}$ either $S_i \prec S'_i$ or $S'_i \prec S_i$. In other words, every feasible ordering must be total.

A total ordering under which S'_i precedes S_i corresponds to an assignment where v_i is true. On the other hand, if $S_i \prec S'_i$, then v_i is false under the corresponding truth assignment. A definition for $\texttt{C}_k\texttt{.x}$ exists if and only if clause c_k is satisfied under this interpretation. Therefore, a feasible ordering defines a satisfying assignment and vice versa. This concludes the reduction of 3-SAT to the ordering problem.□

The correspondence between feasible orderings and satisfying assignments gives rise

to the following corollary:

Corollary 2 *Proving a feasible ordering unique is co-NP-complete.*

Unfortunately, in general it does not help to rely on SML types to solve situations that otherwise look ambiguous. The programs constructed for the NP-completeness proof would not benefit from such additional information. Also, note that SML type inference itself is a hard problem; it has been shown to be DEXPTIME-complete [Mai90, KTU94].

I felt that it is reasonable to require having at most one definition for each top-level symbol per group. Although there are circumstances when one would want to override a given definition with a new one, CM addresses this issue adequately by introducing the notions of libraries and sub-groups. In this discussion of dependency analysis, imported groups of a group are represented abstractly as part of the context environment. Top-level definitions for symbols that were already defined by the context are permitted in this model.

Restriction 1 *In each group there can be at most one source that provides a top-level definition for any given symbol. A source can define a name that was already defined by the context, but uses of that name in any of the sources will then refer to the new definition.*

I implemented this restriction in CM. To my knowledge there has never been an instance where it caused difficulties to users. It provides a well-defined association of defined symbols with the sources in which they are defined. Therefore, dependency analysis can be performed by a depth-first search on the resulting use-definition graph, which can be done in time linear in the number of edges. Those edges correspond to the free occurrences of symbols in SML sources.

Partial orders

Intuitively, $S_1 \prec S_2$ means that S_2 "depends" on S_1. If we want to avoid unnecessary recompilation, then we must capture the idea that two sources do *not* depend on each other. Total orders contain "too many" relations, so we will consider partial orders instead. For the sake of semantic predictability we desire a feasible partial order $\prec_{\min}$ that contains the fewest relations and is unique.[1]

One can represent $\prec_{\min}$ as a DAG of sources given the "predecessor" function $\mathcal{P} \in \text{Source} \to 2^{\text{Source}}$ that is calculated by dependency analysis:

$$\mathcal{A} \in 2^{\text{Source}} \times U \to \text{Source} \to 2^{\text{Source}}$$

Now $\mathit{multidef}_{\text{SML}}$ ($\to$ Equation 3.2) must be revised accordingly; the compilation environment $\tilde{\rho}(S, \rho, \mathcal{P})$ for source S is given by the exports of the predecessors $S' \in \mathcal{P}(S)$:

$$\begin{aligned} \tilde{\rho}(S, \rho, \mathcal{P}) &= (\sum_{S' \in \mathcal{P}(S)} \mathit{def}(S', \tilde{\rho}(S', \rho, \mathcal{P})) + \rho \\ \mathit{multidef}_{\text{SML}}(s, \rho) &= \sum_{S \in s} \tilde{\rho}(S, \rho, \mathcal{A}(s, \rho)) \end{aligned}$$

In general, the summations in these equations still require a total order imposed on the set of sources. But $+$ is commutative under restriction 1, summations become order-independent, and $\mathit{multidef}_{\text{SML}}$ is well-defined.

[1] CM ignores the issue of link-time side effects during module initialization.

Uniqueness and use-def mappings

Partial orders can be extended to become total, but in general there will be more than one way of doing so. However, in some sense one would like to think of all total orders that are compatible with a given partial order as being equivalent. Therefore, it is necessary to clarify the uniqueness requirement.

During compilation every *use* of an identifier will be resolved by the compiler by looking it up in the current compile-time environment. Thus, it will associate each use with a corresponding definition. For the individual uses of identifiers to be distinguishable we will label them in a fashion similar to how definitions were labeled:

$$\text{Use} = \text{Lab}$$

The ordering, partial or total, of sources induces a particular *use-def* mapping M:

$$M \in \text{Use} \to D$$

Note that restriction 1 guarantees that the use-def mapping induced by a feasible partial order is well-defined. Moreover, a feasible use-def mapping reveals the underlying partial order on sources if one collapses the uses of all free variables of each source.

Dependency analysis must reveal a unique use-def mapping.

4.2 Opening structures

SML programs that do not make use of the **open** syntax have a convenient property because both the set of free variables of a source and the set of exported top-level definitions can be determined by scanning only the source itself. It is not necessary to know the definitions

for any of the free variables.

The ability to open a structure, thereby making its constituent definitions directly available without need to use long identifiers, comes at the cost of losing this property. The problem is that **open** introduces a number of definitions, but the names so defined are not lexically apparent. In the scope of such a set of "indirect" definitions it may be that what looks like a free variable is actually bound, and what looks bound under superficial inspection may in certain cases actually be a free occurrence. Here is an example of the latter:

```
structure S = struct
  val x = 1
end
open X
open S
val y = x + 1
```

Opening structure `S` seems to bind variable `x`, but if structure `X`, about which nothing is known, contains a sub-structure `S` without a variable `x`, then `x` is actually free in this code.

Language constructs that, like Pascal's **with** [JW78], bind identifiers implicitly have been criticized before [Ten81, section 6.2.3.]. A feature that confuses dependency analysis tools will not be easy to understand by the human reader.

Similar constructs, for example **import** in Java or **using** in C++, can behave like **open** in SML—with all the same implications for dependency analysis.

Ada's **use** [Ada80] is not capable of masking existing bindings. However, a subtle problem still remains there as well:

```
use A;
use B;
... use of x ...
```

If `x` is supposed to be taken from package `B`, then a bug is introduced into this program if a modification to package `A` causes `A` to define `x` as well, because in this case it is impossible for the second **use**-clause to override the existing definition.

Modula-2 [Wir82] and Modula-3, on the other hand, do not have these problems. In Modula-2 one must write:

```
FROM M IMPORT a, b, c;
```

in order for `a`, `b`, and `c`—and only those—to become directly accessible without having to prefix them by `M`. Therefore, the identifiers defined are lexically apparent even without knowing the definition of module `M`. The design of Oberon takes this much further. Oberon discards both **with** and **from-import**, leaving the language without any facility for circumventing the qualification of identifiers [Wir88a, Wir88b].

In the context of dependency analysis it is especially troublesome that **open** at top level takes away the analyzer's ability to determine the set of exported names by simply scanning the source code. Instead, it will have to process **open** as it goes, which is complicated by the fact that in general yet-to-be-determined knowledge about the dependencies would be required for this. Even after banning multiple definitions for the same name, dependency analysis is NP-complete if the use of **open** is not restricted.

Claim 3 *Dependency analysis is NP-complete for programs with* **open** *where multiple definitions for top-level names are prohibited, but where a top-level definition can override a binding in the context.*

Proof. By the same argument used in the proof for claim 1 the problem is in NP. To prove it NP-hard, I will reduce SAT to the dependency analysis problem.

Consider a formula in conjunctive normal form with n variables v_i and m clauses c_k. Here is the heart of the construction. Suppose there are structures `A` and `A'` defined by the context as follows:

```
structure A = struct
  structure A' = struct end
  val c1 = 1
end


structure A' = struct
  structure A = struct end
  val c2 = 1
  val c3 = 1
end
```

Depending on whether `A` or `A'` is opened first, there will be a definition for either `c1` or both `c2` and `c3`. We can think of these variables as "satisfied clauses"; the order of opening `A` and `A'` corresponds to the truth assignment for a variable.

However, this construction is slightly flawed because a clause can be satisfied by more than one variable, but due to restriction 1 one cannot define the corresponding identifier more than once in different sources. To fix this technical problem, one can delay the definition(s) of variables $\mathtt{c}_k$ until the last source $\hat{S}$ by wrapping them into structures $\mathtt{C}_i^k$. There is one such structure per v_i and clause c_k. $\mathtt{C}_i^k$ contains a definition for $\mathtt{c}_k$ if clause c_k is satisfied by the value of v_i. The sentinel source $\hat{S}$ eventually opens all structures $\mathtt{C}_i^k$ in a local scope, thus adhering to restriction 1. This always works because the context provides a dummy definition for each.

Suppose v_7 appears in clause c_1 and $\bar{v_7}$ in c_2 as well as c_3. The revised version of a "gadget" for v_7 would then be:

```
structure A7 = struct
  structure A'7 = struct end
  structure C1'7 = struct
    val c1 = 1
  end
end
```

```
structure A'7 = struct
  structure A7 = struct end
  structure C2'7 = struct
    val c2 = 1
  end
  structure C3'7 = struct
    val c3 = 1
  end
end
```

Construction: A variable v_i is encoded as a pair of sources S_i and S'_i:

S_i	S'_i
`val` $\mathtt{z}_i$ = $\mathtt{z}_{i-1}$ + $\mathtt{z'}_{i-1}$ `open` $\mathtt{A}_i$	`val` $\mathtt{z'}_i$ = $\mathtt{z}_{i-1}$ + $\mathtt{z'}_{i-1}$ `open` $\mathtt{A'}_i$

Let $\{c_{k_1^+}, \ldots, c_{k_a^+}\}$ be the set of clauses that contain the literal v_i. Then $\mathtt{A'}_i$ is be defined by the context as follows:

```
structure A'_i = struct
  structure A_i = struct
    val y_i = 1
  end
  structure C_i^{k_1^+} = struct
    val c_{k_1^+} = 1
  end
  ⋮
  structure C_i^{k_a^+} = struct
    val c_{k_a^+} = 1
  end
end
```

Likewise, let $\{c_{k_1^-}, \ldots, c_{k_b^-}\}$ be the set of clauses containing $\bar{v}_i$. $\mathtt{A}_i$ becomes:

```
structure A_i = struct
  structure A'_i = struct
    val y_i = 1
  end
  structure C_i^{k_1^-} = struct
    val c_{k_1^-} = 1
  end
  ⋮
  structure C_i^{k_b^-} = struct
    val c_{k_b^-} = 1
  end
end
```

Furthermore, the context also defines `val z`$_0$ `= 0`, `val z'`$_0$ `= 0`, and empty structures $\mathtt{C}_i^k$ for $i = 1, \ldots, n; k = 1, \ldots, m$. An additional source $\hat{S}$ has the form:

```
val z = z_n + z'_n
val y = y_1 + ··· + y_n
local
  open C_1^1
  ⋮
  open C_n^m
in
  val c = c_0 + ··· + c_m
end
```

The definitions for `z`$_0$`, ..., z`$_n$`, z'`$_0$`, ..., z'`$_n$, and `z` restrict any feasible ordering to one where only the relative order of S_k and S'_k for any k is not yet determined. Similarly, `y`$_0$`, ..., y`$_n$, and `y` guarantee that any feasible ordering will be total, because either $S_i \prec S'_i$ or $S'_i \prec S_i$ must be true.

$S_i \prec S'_i$ corresponds to v_i being false, because structure $\mathtt{A}_i$ will be opened first, providing definitions for the "clauses" that contain $\bar{v}_i$, thereby satisfying the corresponding requirements imposed by the code in $\hat{S}$. It also overrides the existing definition for $\mathtt{A}'_i$, so the subsequent opening of that structure will not be able to introduce definitions for any $\mathtt{c}_k$.

In a completely symmetrical fashion, one can argue that $S_i' \prec S_i$ corresponds to an assignment under which v_i is true. A definition for $\mathtt{c}_k$ is available if at least one of the structures containing such a definition is opened. By construction, that will be the case precisely when the corresponding literal becomes true.

Therefore, I have created a set of sources for which a feasible ordering exists if and only if there is a satisfying assignment for the given satisfiability problem. This reduces SAT to the dependency analysis problem and, thus, renders the latter NP-complete.□

To make dependency analysis tractable, one must impose a restriction that, at least, prevents the construction of the program that was used by the proof. The heart of the problem is the ability to open certain structures at top level. If **open** is banned from the top level, then the definitions exported by a source can be determined by looking at just that source. Formally, there is a computable function $\mathcal{E}_H \in \text{Source} \to 2^{\text{Ide}}$ that calculates the head domain of the source's exports independently from the compilation environment. Thus, for every ρ that is a suitable environment for compiling S, we have $\mathcal{E}_H(S) = \mathit{dom}_H(\mathit{def}(S, \rho))$. Whenever the dependency analyzer has to process an internal **open** it will already know where to find the definition of the structure that is being opened. The problem is tractable again.

Restriction 2a *The* **open** *syntax cannot be used at top level.*

Claim 4 *Under restrictions 1 and 2a any feasible use-def mapping is unique if it exists.*

Proof. I refer to the proof for the *stronger* claim 5.□

The current implementation of CM enforces restriction 2a. I believe in a programming style that uses SML's module language extensively, so there is no need for **open** at top level. However, in some instances such a complete ban was prohibitive. These cases have been rare, but occasionally it is important to support them. For example, someone who is using

Concurrent ML [Rep91] extensively, as a programming language in its own right, might want to open the `CML` structure to have more convenient access to its components.

There are several ways of restricting the use of top-level **open** in a more relaxed way. The drawback is that it becomes increasingly difficult to specify the rules precisely and to explain them to the user. The latter has an impact on, for example, the quality of error messages and therefore on overall acceptance of the dependency analyzer as a tool.

Here is an alternative to restriction 2a, which also leads to a tractable dependency analysis problem:

Restriction 2b *Instances of the* **open** *syntax at top level are not permitted to introduce definitions for names that are already defined by the context.*

This restriction can be weakened some more by limiting its scope to structure definitions only. In fact, it can be relaxed even further by only considering definitions for those structures that are also used (as opposed to just being re-exported):

Restriction 2c *Instances of the* **open** *syntax at top level are not permitted to introduce definitions for structure names that are used somewhere within the group if the context already provides a definition for them.*

Restriction 2a is strictly stronger than restriction 2b, and restriction 2c is a further relaxation of the latter.

Claim 5 *Under restrictions 1 and 2c any feasible use-def mapping is unique if it exists.*

Proof. Suppose there are two feasible use-def mappings M and M'. To be different, there must be at least one use of an identifier where they disagree. I shall show that this is not possible.

Consider the partial order $\prec_M$ on sources that is induced by M. I use the notation $x \in S$ for uses x of identifiers that occur in S and $M(x) \in S'$ for the corresponding definitions $M(x)$ that occur in S'. Some definitions are given by the context environment ρ_0. In this case I write $M(x) \in \rho_0$.

Let there be at least one use where M and M' disagree. Then there must be a source $\hat{S}$ and a use $\hat{x} \in \hat{S}$ such that:

1. $M(\hat{x}) \neq M'(\hat{x})$
2. no predecessor of $\hat{S}$ reveals discrepancies between M and M':

$$\forall S, x : S \prec_M \hat{S} \wedge x \in S \Rightarrow M(x) = M'(x)$$

3. $\hat{x}$ is the (textually) earliest use of a name in $\hat{S}$ for which M and M' disagree

Let us find the location of $M(\hat{x})$. There are three possible cases:

1. $M(\hat{x}) \in \hat{S}$
2. $\exists S : M(\hat{x}) \in S \wedge S \prec_M \hat{S}$
3. $M(\hat{x}) \in \rho_0$

But none of these cases can actually occur:

1. If $M(\hat{x}) \in \hat{S}$, then M' must induce a different scope for $\hat{x}$ in $\hat{S}$. The only language construct that is capable of inducing different scoping for different use-def mappings is **open**, and such an **open** must textually precede the use x. But we picked x to be the textually earliest use of a name where M and M' disagree.
2. If $\exists S : M(\hat{x}) \in S \wedge S \prec_M \hat{S}$, then S exports different definitions under M than it does under M'. This can only happen if S opens a structure Y, and $M(Y) \neq M'(Y)$.

But $\hat{S}$ was picked to be minimal; no predecessor of $\hat{S}$ can contain a use of such a Y for which M and M' disagree.

3. If $M(\hat{x}) \in \rho_0$, then $M'(\hat{x}) \notin \rho_0$. Therefore, there must be a source S' exporting $M'(\hat{x})$ under M': $\exists S' : M'(\hat{x}) \in S'$. Because of restriction 2c, no top-level **open** can provide the definition $M'(\hat{x})$. Therefore, there must be an explicit definition for (the head-component of) $\hat{x}$ in S'. There is no language construct capable of wiping out such an explicit definition, even under different use-def mappings. This implies that S' exports $M'(\hat{x})$ under *any* mapping, including M. But this is impossible because restriction 1 would then demand $\hat{x}$ to refer to $M'(\hat{x})$, which I assumed it does not.

Thus, if both M and M' are feasible use-def mappings, then they must coincide.□

4.3 The analysis algorithm

The previous discussion has established that if a use-def mapping exists, then it must be unique under restrictions 1 and 2c. Suppose the mapping is already known. One could then *verify* it by processing individual sources in topological order. From this idea one can derive a quadratic-time algorithm for *discovering* the correct partial order.

To present the algorithm formally, let us consider a simplified language that only contains structure declarations, sequences of declarations, and opening of structures. Each source of the group to be analyzed is represented by a declaration (*decl*). The definition of *decl* is shown in figure 4.1. Omitted from this language are signatures, signature constraints on structures, functors, and functor applications. They do not complicate matters further and would only add bulk to the exposition.

Structures can be defined to contain any number of other declarations (possibly zero), or they can be equal to previously defined structures. Structure declarations assign a struc-

$$
\begin{array}{rcl}
\mathit{lid} & = & \mathit{id} \times \mathit{id}^* \\
 & & \\
\mathit{decl} & \rightarrow & \textbf{structure}\ \mathit{id}\ \mathit{strexp} \\
 & | & \textbf{seq}\ \mathit{decl}\ \mathit{decl} \\
 & | & \textbf{open}\ \mathit{strexp} \\
 & | & \textbf{empty} \\
\mathit{strexp} & \rightarrow & \textbf{name}\ \mathit{lid} \\
 & | & \textbf{struct}\ \mathit{decl}
\end{array}
$$

Figure 4.1: **Simple module language.** This figure shows the abstract syntax of a module language that has been simplified for expository purposes. However, the language still has nested structures and the ability to open them. Therefore, it exhibits the same intrinsic problems with respect to dependency analysis that are present in Standard ML.

Direct-Decl($\textbf{structure}\ (v, d)$) = $\{v\}$
Direct-Decl($\textbf{seq}\ (d_1, d_2)$) = Direct-Decl(d_1) $\cup$ Direct-Decl(d_2)
Direct-Decl($\textbf{open}\ (s)$) = $\{\}$
Direct-Decl($\textbf{empty}$) = $\{\}$

Lookup-Rest($\rho, \langle\rangle$) = ρ
Lookup-Rest($\rho, \langle v_1, v_2, \ldots\rangle$) =
    if $v_1 \in$ *dom*(ρ) **then**
        Lookup-Rest($\rho(v_1), \langle v_2, \ldots\rangle$)
    else abort "member not found in structure"

Figure 4.2: **Auxiliary functions for dependency analysis.** *Direct-Decl* calculates the set of names bound by "direct" definitions. A direct definition is one that is not introduced via **open**. Given the environment for the head component of a long identifier, we use *Lookup-Rest* to complete the lookup operation for the entire name. Note that in correct programs this operation must always succeed.

ture expression (*strexp*) to a simple identifier (*id*). A structure expression is either a long identifier (*lid*) that refers to some previously defined structure or a declaration (*decl*) that provides definitions for the members of the structure.

Environments $\rho \in U$ map simple identifiers to other environments: $\rho(v)$ represents the definitions for members of the structure that is named v in environment ρ. A name that is mapped to an empty environment is different from a name that is not mapped at all. The notation $\rho[v \mapsto \rho']$ refers to the environment ρ augmented with a new binding that maps v to ρ'—possibly overriding an existing binding for the same variable v. The already familiar operator $+$ denotes environment layering.

Figure 4.2 shows two auxiliary functions. *Direct-Decl* calculates the set of simple identifiers for which there is a definition that was provided directly and not by opening some structure, while *Lookup-Rest* resolves the remaining components of a long identifier once the environment representing its head component is known.

The input to the algorithm is a set $\{d_1, \ldots, d_n\}$ of sources (represented by *decl*s) and the context environment ρ_0. The objective is to calculate the partial order *Depend*, where *Depend*$[i]$ gives the indices of those sources that d_i depends on. The set denoted by D is used to remember all simple names for which there is a direct definition in one of the sources. The variable *Analyzed* keeps track of sources that have already been analyzed successfully. Each element $(i, \rho_i) \in$ *Analyzed* contains the environment ρ_i representing definitions exported from d_i.

Function *Analyze-Source* is implemented in terms of two mutually recursive functions *Analyze-Decl* and *Analyze-Strexp*, which are used to process *decl*s and *strexp*s, respectively. These functions are shown in figure 4.3. The result obtained from a call to *Analyze-Decl* represents the definitions exported from a *decl*, while the value returned from *Analyze-Strexp* corresponds to the members of a given structure. The environment argument implements scope rules by keeping track of local definitions.

```
Analyze-Source(D,                                  set of names that have an explicit definition
               Analyzed,                           results from successful analyses
               ρ_0,                                context environment
               d) =                                current source
    P ← {}                                         initialize dependencies for this source

    Analyze-Decl(structure (v, s), ρ) =            explicit definition
        return ρ[v ↦ Analyze-Strexp(s, ρ)]
    Analyze-Decl(seq (s_1, s_2), ρ) =              sequential definitions
        return Analyze-Decl(s_2, Analyze-Decl(s_1, ρ))
    Analyze-Decl(open s, ρ) =                      opening a structure
        return Analyze-Strexp(s, ρ) + ρ
    Analyze-Decl(empty, ρ) =                       empty definition
        return ρ

    Analyze-Strexp(struct d, ρ) =                  new structure body
        return Analyze-Decl(d, ρ)                  analyze body of the structure
    Analyze-Strexp(name (v, v*), ρ) =              name of existing structure
        if v ∈ dom(ρ) then                         is defined in same source
            return Lookup-Rest(ρ(v), v*)
        else if v ∉ D ∧ v ∈ dom(ρ_0) then          has no direct definition but is defined in context
            return Lookup-Rest(ρ_0(v), v*)
        else if ∃(j, ρ_j) ∈ Analyzed : v ∈ dom(ρ_j) then   defined in other source (already analyzed)
            P ← P ∪ {j}                            register dependency
            return Lookup-Rest(ρ_j(v), v*)
        else                                       so far, no definition is known
            return "abandon"                       defer current analysis

    ρ ← Analyze-Decl(d, ∅_U)                       run analysis, gather dependencies
    return (ρ, P)                                  return export environment and dependencies
```

Figure 4.3: **Syntax-directed traversal as performed by the dependency analyzer.**

```
Analyze(⟨d_1, ..., d_n⟩,                          representation of n sources
        ρ_0) =                                    context environment
    D ← ⋃_{i=1}^{n} Direct-Decl(d_i)               calculate set of explicitly defined names

    FindNext(Analyzed, {}) =                      all sources have been analyzed
        return Depend                             return final dependency graph
    FindNext(Analyzed, R) =                       more sources to be analyzed
        return Try(Analyzed, R)                   find source where analysis succeeds

    Try(Analyzed, {}) =                           search was unsuccessful
        abort "undefined variable or cyclic reference"
    Try(Analyzed, {(i, d)} ∪ R') =                pick arbitrary element
        case Analyze-Source(D, Analyzed, ρ_0, d)  try analyzing this source
        of (ρ, Dep) ⇒                             analysis was successful
            Depend[i] ← Dep                       remember dependencies
            return FindNext(Analyzed ∪ {(i, ρ)}, R')   analyze rest
        | "abandon" ⇒                             analysis was not successful
            return Try(Analyzed, R')              keep searching

    return FindNext({}, {(1, d_1), ..., (n, d_n)})   start analysis for all sources
```

Figure 4.4: **Dependency analysis.** Dependency analysis consists of two nested loops. Function FindNext loops over the set of sources that are yet to be analyzed. The inner loop, represented by function Try, repeatedly invokes Analyze-Source until it finds a source where it succeeds. The algorithm calls Try $O(n^2)$ times in the worst case.

The important aspect of the algorithm is the way it handles names that are not found in the local environment. First it checks D and ρ_0. Restriction 2c guarantees that a binding in ρ_0 is the correct one to be used if the variable is not in D. Otherwise the definition must be exported from one of the other sources. *Analyzed* is checked to see if a previously analyzed source has already revealed it. If this is not the case, then the current source was processed prematurely; analysis must be repeated later.

The remainder of the algorithm, shown in figure 4.4, consists of two nested loops represented by FindNext and Try. R holds pairs (i, d_i) of indices and sources that still need to be processed. The inner loop repeatedly calls *Analyze-Source* until it finds a source for which this analysis succeeds.

The restrictions guarantee that names will be resolved correctly if they are resolved at all. Therefore, the algorithm will indeed discover the desired partial order if it exists. In the worst case it will take $O(n^2)$ calls to *Analyze-Source* to do so.

It is possible to reduce running time to $O(n)$ by avoiding repeated invocations of the analyzer for the same source. The trick is to run the analysis algorithm on all sources simultaneously. Instead of abandoning a computation and later duplicating work that already had been accomplished, the algorithm will simply wait until definitions for previously unknown names become available.

A version of such an algorithm had been implemented in SC, which was CM's precursor [HLPR94a]. But the authors of SC were not aware of the general problem's complexity class, so they only enforced restriction 1 and made no attempt to restrict the use of top-level **open**. As a result, the analysis performed by SC was unsound in the sense that for certain programs it would fail to find an existing feasible ordering. Furthermore, for some programs with ambiguous dependencies, it would silently pick one of the choices without warning the user about the existence of others that differ semantically.

To present the algorithm, I rely on a small number of primitives for non-preemtive

```
global Analyzed                                   results of successful analyses
global Depend                                     dependency graph to be constructed

Analyze-Source′(d,                                current source
                i,                                index of current source
                D,                                set of names that have an explicit definition
                ρ0) =                             context environment
    Analyze-Decl(structure (v, s), ρ) =           explicit definition
        return ρ[v ↦ Analyze-Strexp(s, ρ)]
    Analyze-Decl(seq (s1, s2), ρ) =               sequential definitions
        return Analyze-Decl(s2, Analyze-Decl(s1, ρ))
    Analyze-Decl(open s, ρ) =                     opening a structure
        return Analyze-Strexp(s, ρ) + ρ
    Analyze-Decl(empty, ρ) =                      empty definition
        return ρ

    Analyze-Strexp(struct d, ρ) =                 new structure body
        return Analyze-Decl(d, ρ)                 analyze body of the structure
    Analyze-Strexp(name (v, v*), ρ) =             name of existing structure
        if v ∈ dom(ρ) then                        is defined in same source
            return Lookup-Rest(ρ(v), v*)
        else if v ∉ D ∧ v ∈ dom(ρ0) then          has no direct definition but is defined in context
            return Lookup-Rest(ρ0(v), v*)
        else
            wait v                                block if no other thread has revealed a definition yet
            ∃!(j, ρj) ∈ Analyzed : v ∈ dom(ρj)    definition provided by source dj
            Depend[i] ← Depend[i] ∪ {j}           register dependency
            return Lookup-Rest(ρj(v), v*)

    ρ ← Analyze-Decl(d, ∅U)                       run analysis, gather dependencies
    Analyzed ← Analyzed ∪ {(i, ρ)}                register successful analysis
    resume dom(ρ)                                 restart waiting threads; no future wait on these events will block
    return                                        terminate thread
```

Figure 4.5: **Syntax-directed traversal modified for concurrent analysis.**

```
Analyze(⟨d_1, ..., d_n⟩,                          representation of n sources
        ρ_0) =                                    context environment
    D ← ⋃_{i=1}^{n} Direct-Decl(d_i)               calculate set of explicitly defined names
    for i ← 1 to n do Depend[i] ← {}               initialize dependency graph
    Analyzed ← {}                                 initialize analysis results
    Threads ← {}
    for i ← 1 to n do                             start all analysis threads
        T ← fork Analyze-Source'(d_i, i, D, ρ_0)
        Threads ← Threads ∪ {T}
    join Threads                                  collect threads after termination
    if deadlock then                              deadlock indicates error in sources
        abort "undefined variable or cyclic reference"
    return Depend                                 return dependency graph
```

Figure 4.6: **Concurrent dependency analysis.** The concurrent version of dependency analysis creates one thread per source. Therefore, it only incurs $O(n)$ calls to *Analyze-Source*.

concurrency. Threads are created using **fork** and collected using **join**. A thread can wait on an event using **wait**. The **resume** operation unblocks all threads that wait on one of the events in the specified set. It also prevents from blocking any future **wait** operations on these events.

Figure 4.5 shows a concurrent version of *Analyze-Source*. As before, *Analyze-Strexp* goes through a case analysis for resolving structure names. However, in the absence of a definition for a name it does not abandon the computation but waits for such a definition to arrive. Identifiers play the role of events in this algorithm.

The new main loop simply spawns one thread for each source, waits for their completion, and returns the result. This is shown in figure 4.6.

4.4 Optimizations

It is relatively expensive to parse an entire source file every time the dependency analyzer needs it. Therefore, CM calculates a condensed version of the source, which sheds all

parts of the abstract syntax tree that are not necessary for dependency analysis. The much smaller result is kept in a cache.

Without **open** this would be very easy. All the dependency analyzer needs to know is the set of names that occur free in a source and the set of names defined and exported by the source. In the absence of **open** (and the absence of datatype replication as introduced by the revised definition of SML [MTH90, MTHM97]) it is straightforward to calculate these sets for each given source.

With **open** this becomes more involved because, as we have seen before, without prior knowledge of the structure being opened the analyzer will potentially lose track of what is currently bound or free. Furthermore, the condensed version of the source must still maintain knowledge about the constituent parts of a structure in order to be able to handle cases where dependency analysis eventually reveals that it is being opened somewhere.

Even worse, the fact that **open** may be used locally (e.g. inside a **let** expression) means that information about nested scopes in ordinary program code must also be retained. For example, in the expression

```
let open X
    val b = a + 1
in
    b + c
end
```

it is not clear whether `a` or `c` are actually free until `X` becomes available.

Summary information about definitions and uses can still be obtained for the code between separate occurrences of **open**. For instance, in the previous example we know that `b` is not free because there is no **open** separating its definition from its use. Fortunately, the savings will normally be substantial because programmers tend not to use **open** locally very often.

The current implementation of CM uses a different strategy of avoiding this problem. It

simply ignores all names that are not structures, signatures, functors, or functor signatures. Again, for this to be useful, one must rely on a programming style where everything is defined in modules, but it avoids the need to keep track of anything but module definitions and uses.

The size of typical dependency files is only 1–4% of the corresponding SML source code size.

4.5 Implications for language design

The use of a compilation management tool like CM is considerably simplified by automatic dependency analysis. Unfortunately, some features of the SML language pose as a problem for tractability. I have shown two simple restrictions that fix such problems and that in my experience are entirely reasonable in practice.

As we have seen, the language feature most troublesome for the dependency analyzer is **open**. It is a pity that SML'97 [MTHM97] has made things worse with the introduction of datatype replication, which also has the property of introducing bindings for identifiers that are not lexically apparent. However, datatype replication is no more difficult to deal with than **open**. The algorithms presented in section 4.3 would work just as well under a suitable (but straightforward) extension of restriction 2c.

Chapter 5

Cross-Module Optimizations

Abstraction and modular design of software promote clarity and provide clear lines along which large projects can be subdivided. But one often pays a large performance penalty for using abstraction. Cross-module inlining, by which I mean "inline-expansion across compilation unit boundaries," can bridge the gap between abstract design and high performance by transparently moving the border between compilation units.

I need not explain in detail the disadvantages of making the programmer inline-expand "by hand": burdening the programmer, blurring modular design by exposing implementation details, and making it difficult to adjust the amount of inlining dynamically for development (recompiling) or "shrink-wrapping" mode.

But the automatic cross-module inlining schemes used to date have not treated free variables, nested scopes, higher-order functions, or link-time side effects from module-level initializers [DH88, CHT91, CMCH92]. They cannot move a function-body from compilation unit A to compilation unit B if the function has a free variable that is not exported from A and cannot be copied into B. This limits the generality of existing approaches, especially when applied to higher-order functional languages.

One might think of inlining functions after closure-conversion. Closure-conversion is

a phase found in many compilers for functional languages. Its purpose is to eliminate free variables from all functions by turning those variables into additional parameters. But this does not solve the problem because the free variables have become function arguments, and the callers in other scopes have no access to the corresponding actual-parameter values.

λ-splitting is a new, fully automatic technique for cross-module inlining. It exposes implementation details on an as-needed basis only to the compiler of client code; abstraction and modularity are never compromised at the source level. Furthermore, by tuning a few compile-time parameters, one can adjust the aggressiveness of cross-module inlining or turn it off completely.

In contrast to previous experiments [Sch77, CHT91] that did not explain how to preserve efficient separate compilation while inlining, λ-splitting is fully integrated with the separate compilation system of SML/NJ. It cleanly exports inlinable portions of one source file through the binary object file into the importing compilation unit.

This approach should be applicable to a wide range of languages and compilers with intermediate languages based on λ-calculi [KKR$^+$86, Pey87]. λ-calculus [Chu41, Bar81] has the advantage over competing intermediate representations of being a well-studied logical system. It also proves to be very convenient for expressing the necessary transformations in my prototype. Although the emphasis on λ-calculus favors functional programming languages as a target for my technique, the ideas could be adapted to other languages and implementations.

The compiler makes cross-module inlining decisions automatically. This frees the programmer from the burden of having to worry about many "micro-optimizations." The relationship between source-level constructs and functions to be inlined is not always obvious to the programmer, especially in implementations that pass type information at runtime, and which therefore may encode polymorphism as abstraction and type specialization as function application [Oho92, Tol94, HM95, Sha97a, Sha97c].

Resource-conscious programmers often like to say explicitly that they believe certain procedure calls should be inline-expanded. Performance hints from a profiling feedback system [CMCH92] can play a similarly important role. The framework that I explain here can easily be adapted to take account of external hints.

λ-splitting moves function bodies from one source to another, but ensuring that these functions are then inlined *within* the importing compilation unit is up to the existing intramodule inliner [App92], which does a good but not perfect job. Since the intermodule inlining algorithm preserves separate compilation, it must make decisions without seeing the client modules. Therefore, it relies on simple syntactic cues instead of powerful dataflow analyses [JW96] that could aid inlining decisions.

I have previously reported on results of this work during the 1997 ACM SIGPLAN International Conference on Functional Programming [BA97].

5.1 Example

Consider a simple procedure, which maintains the maximum of the values it has been presented with so far. The implementation is parameterized, so it can be instantiated with different comparison predicates `lt`.

```
fun extremeFun (lt, x0) =
let val e = ref x0
    fun get () = !e
    fun check0 x =
        if lt (!e, x) then e := x
        else ()
    fun check [] = ()
        | check (h ::  t) = (check0 h; check t)
in (get, check, lt) end
```

It is often desirable to inline-expand function calls, especially if the functions are small and calls occur frequently. This is true regardless of whether the function in question is

defined within the same or a different compilation unit. Let the example above be one compilation unit, while another unit instantiates `extremeFun` in order to be able to use the resulting procedures `check` and `get`.

```
val (get, check, lt) =
  extremeFun (op < , 10000);
...
```

Calls to `get` are really just accesses to reference cell `e`; one would like to inline-expand `get`, but `e` is not exported—presumably because it was not supposed to be mutable from the outside other than via `check`. But after type-checking, the compiler should be allowed to inline the function.

The transformation presented here achieves this effect even though `e` is originally not exported *and* everything is parameterized and therefore hidden within the body of `extremeFun`. λ-splitting takes one compilation unit and rewrites it as two parts. The first part contains those sections of the code that cannot or should not be inlined, while the other will be made available for being inline-expanded into client modules.

It is common for a compilation unit to have more than one client, which means that the inlinable portion of that unit will be duplicated. For correctness it is necessary to avoid the duplication of side effects. To prevent excessive growth in total code size, one must keep the inlinable portion small. Section 5.5 shows the measure that I use to estimate code size.

Of course, at compile time the presence of side effects cannot be decided precisely. Therefore, the algorithm relies on a safe approximation that can easily be characterized syntactically. It is very similar to the notion of *expansiveness*. Accordingly, I call the first part of a split the *expansive portion* (even though in general it will also contain a lot of non-expansive code) and use the subscript e as a label; code examples are marked with the letter `E`.

The second, inlinable part of a split—marked with subscript i or a letter `I`—contains no

expansive code, only values (variables, constants, λ-abstractions) and other effect-free code (records, immutable vectors, datatype constructors). It will be compiled together with its client code. A formal definition of what the algorithm considers "permitted" to be inlined is given in section 5.5. In effect, λ-splitting redraws the lines between compilation units and lets an existing local optimizer look beyond the boundaries of the original compilation units.

In the example, assuming the size of `extremeFun` to exceed the limit that is imposed on inlinable code, the transformation will divide it into `extremeE` and `extremeI`:

```
fun extremeE (x0, check0Fun) =
let val e = ref x0
    val check0 = check0Fun e
    fun check [] = ()
      | check (h ::  t) = (check0 h; check t)
in (e, check) end
```

The expansive portion `extremeE` is compiled to machine code; the inlinable portion `extremeI` is kept in symbolic form and will be copied verbatim into the client's code:

```
local
fun extremeI (lt, x0) =
let fun check0Fun e =
    let fun check0 x =
          if lt (!e, x) then e := x
          else ()
    in check0 end
    val (e, check) =
        extremeE (x0, check0Fun)
    fun get () = !e
in (get, check, lt) end
in
val (get, check, lt) =
    extremeI (op < , 10000)
...
end
```

The intramodule inliner reduces this client into:

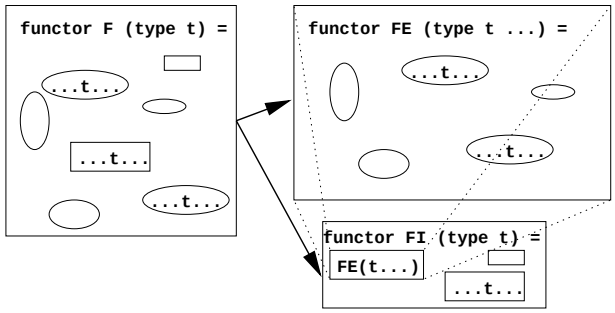


Figure 5.1: **Functor splitting.** Type `t` is propagated through the functor into the client by copying `FI` into each client and β-reducing it. But `F` contains an expansive portion (ovals) that cannot be copied; this is abstracted as `FE`. `F`, and `FE`, may be very large, but only the inlinable portion `FI`, which the algorithm keeps small, gets copied into clients.

```
val (e, check) =
    extremeE (10000,
      fn e => fn (x:  int) =>
          if !e < x then e := x
          else ())
fun get () = !e
val lt = (op <):  int * int -> bool
...
```

Thus, λ-splitting achieves the following effects:

- `extremeE` exports reference cell `e`.
- The compiler can now rewrite invocations of `get` in the client as accesses to `e`.
- Information about the fact that `lt` is the same as `op <` has not been lost on its way through the body of `extremeFun`.
- The function `check0` is lifted out of `extremeE` and put into `extremeI` so the actual comparison < can be inlined.

The example shows what can be gained by a higher-order inlining algorithm. Previous approaches would not inline `get`, `check`, and `lt` because they do not appear at top level.

A function like `extremeFun` really plays the role of a parameterized module. In ML terminology these are called *functors*, and there is special syntax (which I did not use here

to avoid complicating the presentation) together with ramifications for the static semantics of the language.

Standard ML provides only first-order functors, but this has been taken a step further in SML/NJ by offering a higher-order module system [Mac90, MT94]. Functors can be defined within functors, they can be passed as functor arguments, and they may be returned as part of functor instantiations. Therefore, in the intermediate representation there is really not much difference between ordinary functions and functors because they are represented the same.

Functions are non-expansive themselves. One could, without sacrificing correctness, place them into the inlinable portion as a whole, thereby effectively delaying the compilation of parameterized modules until they are instantiated. This would maximize the compiler's chances of generating optimized code tailored to each individual instantiation. It also duplicates all of the functor's body every time it is used, potentially leading to intolerable increases in size. But large functors sometimes have a small "core" that should be delayed and inlined for efficiency, while the remainder can be compiled where it is declared. λ-splitting is able to accomplish that (almost—see the discussion of $\underline{1}$-decomposition).

5.2 Separate compilation and linking

The meaning of any compilation unit can be understood as a function that maps meanings for its imports to meanings for its exports. This allows us to view cross-module inlining as a problem of manipulating such functions.

Imports correspond to the free identifiers in the source program; new definitions that other compilation units can refer to are considered exports. One can assign meaning to a system of several compilation units by solving the corresponding system of equations, which describe the import-export relations. In languages that permit circular dependencies

among source files this will in general involve calculating a fixed point; without circularities it becomes much simpler.

Standard ML has no circular dependencies among compilation units. As discussed in chapter 3, separate compilation and linking can be modeled using nonrecursive higher-order functions. Type checking already imposes an ordering on compilation steps, guaranteeing that whenever a source B refers to something exported from source A, then A must be compiled before B, thereby ensuring that inlining information from A will be available at the time B is compiled.

Appel and MacQueen [AM94] describe how Standard ML of New Jersey implements separate compilation. Since this is the basis for this work I will summarize it here:

A compilation unit consists of a number of definitions for types, values, signatures, structures, and functors. Types and signatures are relevant only for dealing with static semantics and can be ignored as far as linking, a dynamic concept, is concerned. The SML/NJ intermediate representation (λ-language [App92]) represents structures as records, and functors as functions. It does not distinguish between the core and module languages. Each compilation unit is turned into a λ-expression, where references to identifiers from other compilation units appear as free variables. By abstracting over those variables one obtains a closed expression, which can now be compiled to machine code without further need to refer to any context information.

Thus, a compilation unit becomes a function taking imported values to exported values. Exported values are collected in a record that is the result of executing the unit. Another unit, which imports those values, simply takes that record as a function argument.
For example, a unit consisting of definition:

```
val a = 1 and b = 2
```

would be represented as `fun A () = (1, 2)`. The compiler remembers that `a` sits in the

first position and `b` in the second. Similarly, a unit consisting of the definitions

```
val c = 3 and d = 4
```

becomes `fun B () = (3, 4)`. Now, if one compiles a third unit containing

```
val e = a * b + c - d
and f = a + c
```

the compiler will construct a λ-expression of the form

```
fun C [A, B] =
let val (a, b) = A
    val (c, d) = B
in (a * b + c - d, a + c) end
```

5.3 λ-splitting

For simplicity let us ignore the issue of importing from multiple sources, so we need not deal with vectors of imports. The code for a unit U is a closed function mapping imports to exports: $U : I \rightarrow E$. The objective of λ-splitting is to choose a triplet $(T,\ U_i : T \rightarrow E,\ U_e : I \rightarrow T)$, such that the composition $U_i \circ U_e$ yields U again. T denotes the domain of possible values that are passed from U_e to U_i and corresponds to a type if the intermediate language is typed. The triplet is said to be a *B-decomposition*[1] or simply a *split*. For example, a trivial split for any $U : I \rightarrow E$ is (E, id_E, U), where $\mathrm{id}_E : E \rightarrow E$ is the identity function on E.

Now consider two units, where the second imports the exports of the first: $U^1 : I^1 \rightarrow E^1$ and $U^2 : I^2 \rightarrow E^2$, $E^1 = I^2$. The meaning of the program comprised of U^1 and U^2 can be understood in terms of their composition $U^2 \circ U^1$. For any split (T, U_i^1, U_e^1) this is the same as $U^2 \circ (U_i^1 \circ U_e^1)$, and by associativity $(U^2 \circ U_i^1) \circ U_e^1$. In other words, one can compile U_e^1 as one unit, $U^2 \circ U_i^1$ as the other, and still obtain the correct result. Cross-

[1] *B* is the composition combinator $\lambda f \lambda g \lambda x.f(gx)$.

module inlining can therefore be achieved by cleverly placing the things to be inlined by U^2 into U_i^1.

In general, U^1 will be referred to by several other units. The code that goes into U_i^1 must be chosen carefully because it will be duplicated in each one. Therefore, the algorithm never places expansive code and code that is deemed too big into the i-portion of a split. The idea of splitting functions into pieces can also be used to facilitate intramodular optimizations like "call forwarding" [BDGK94] or partial inlining [Gou94].

The basic algorithm

In SML/NJ a compilation unit translates into a sequence of nested variable bindings (`let`-bindings), some of which can be recursive. The rightmost body of the rightmost `let`-expression then builds a record of exported values. The entire construction is finally wrapped into a λ-abstraction together with some code for selecting values for the source code's free variables from the argument.

The code corresponding to any compilation unit U will be of the following general form, where the set $\{w_1, \ldots, w_k\}$ is a subset of $\{v_1, \ldots, v_n\}$:

```
fn v0 =>
let val v1 = E in
  let val v2 = E2 in
   ...
     let val vn = En in
       (w1, w2, ..., wk)
     end
   ...
  end
end
```

The construction of the code for U_e and U_i starts with selecting all j such that the bindings `val vj = Ej` are guaranteed to be effect-free. This will ensure that no side effects will be duplicated by the algorithm. A formal definition of what it considers "guaranteed

to be effect-free" is given in section 5.5. In most cases the set of bindings chosen in this step will be rather large. A combination of several heuristics will further reduce it.

The remaining bindings are then used to build U_i's body, and finally one needs to find T and U_e to complete the split $(T,\ U_i : T \to E,\ U_e : I \to T)$. The elements of T must communicate values for all free variables of U_i's body. These free variables will always be a subset of $\{v_0, v_1, \ldots, v_n\}$, so they are readily available at the time when the original code for U would have constructed its export record. Furthermore, since only non-expansive code has been placed into U_i, there will be no harm in duplicating some or even all of it in U_e. Therefore, the task of constructing U_e becomes surprisingly simple. One can use U almost unchanged; only the export record has to be replaced with a record holding said free variables. Of course, some of the bindings in U_e may become unnecessary. But the intramodular optimizer will delete dead bindings.

A-normal form

To break large expressions into inlinable pieces, I use a variant of A-normal form [FSDF93] as the calculus for λ-splitting. The important property is that every intermediate result will be explicitly bound to a variable. This increases the number of bindings while, on average, reducing the size of the expressions being bound. Thus, splitting decisions can be made with finer granularity. The structure of the intermediate language automatically enforces this. Continuation-passing style [Ste78, App92] would do as well; but the SML/NJ compiler transforms to CPS at a late stage after type information is discarded. I would rather preserve the ability to do monomorphic instantiation of polymorphic functions [SA95] *after* cross-module inlining has been done.

As an example of how A-normal form helps inlining, consider that the expression bound to `x` in:

```
let val x = (1, ref 0)
    val y = 2
in (x, y) end
```

is expansive because `ref 0` is expansive. But in A-normal form this turns into:

```
let val tmp = ref 0
    val x = (1, tmp)
    val y = 2
in (x, y) end
```

In this program `x` is bound to a non-expansive expression, which can very well be moved into the i-section of the split, thus exposing the 1 in the first field of the tuple. Thus, one can move more code into U_i.

The intermediate language

The following calculus is used as the intermediate representation. It is reminiscent of A-normal form. The only difference is that one does not need to distinguish between ordinary function application and tail calls. Variable bindings are established by **fn**, **fix**, and **let**. Each variable is bound at most once; variables occurring in closed functions are bound exactly once. I distinguish between *pure* and *impure* built-in "primitive operations" (primops); the former are guaranteed not to incur effects upon application. Even "read" effects, such as accesses to reference variables, are considered impure here.

$$
\begin{array}{ll}
\textit{var} & \rightarrow \textbf{v0} \mid \textbf{v1} \mid \ldots \\
\textit{const} & \rightarrow \textit{int} \mid \textit{real} \mid \ldots \\
\textit{primop} & \rightarrow \textbf{pure}\ \textit{pure} \\
 & \rightarrow \textbf{impure}\ \textit{impure} \\
\textit{val} & \rightarrow \textbf{var}\ \textit{var} \\
 & \rightarrow \textbf{const}\ \textit{const} \\
 & \rightarrow \textit{primop} \\
\textit{exp} & \rightarrow \textbf{val}\ \textit{val} \\
 & \rightarrow \textbf{fn}\ (\textit{var}, \textit{exp}) \\
 & \rightarrow \textbf{fix}\ ((\textit{var}, \textit{var}, \textit{exp}) \ldots, \textit{exp}) \\
 & \rightarrow \textbf{let}\ (\textit{var}, \textit{exp}, \textit{exp}) \\
 & \rightarrow \textbf{app}\ (\textit{val}, \textit{val}) \\
 & \rightarrow \textbf{record}\ (\textit{val} \ldots) \\
 & \rightarrow \textbf{select}\ (\textit{int}, \textit{val}) \\
 & \rightarrow \textbf{if}\ (\textit{val}, \textit{exp}, \textit{exp})
\end{array}
$$

Expressions that only consist of a variable access, a constant, or a primitive operator use the **val** clause; **fn** introduces a λ-abstraction. Recursive function bindings are established using **fix**; an expression of the form

fix ((f1,v1,E1),(f2,v2,E2),. . .,B)

corresponds to the SML expression

```
let fun f1 v1 = E1
    fun f2 v2 = E2
in B end
```

A **let**-expression locally binds a variable; **app** denotes the application of one value to another. Records are constructed using **record** and selected from using **select**. The **if**-clause specifies a conditional expression.

$\mathcal{F}_{\mathrm{V}}(\ldots)$ and $\mathcal{F}(\ldots)$ calculate the sets of free variables for *exp* and *val*, respectively:

$$
\begin{aligned}
\mathcal{F}_{\mathrm{V}}(\textbf{var}\ v) &= \{v\} \\
\mathcal{F}_{\mathrm{V}}(\textbf{const}\ c) &= \emptyset
\end{aligned}
$$

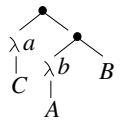
$$
\begin{aligned}
\mathcal{F}_{\mathrm{v}}(\textbf{pure}\ p) &= \emptyset \\
\mathcal{F}_{\mathrm{v}}(\textbf{impure}\ p) &= \emptyset \\
\\
\mathcal{F}(\textbf{val}\ x) &= \mathcal{F}_{\mathrm{v}}(x) \\
\mathcal{F}(\textbf{fn}\ (v, e)) &= \mathcal{F}(e) \setminus \{v\} \\
\mathcal{F}(\textbf{fix}\ (l, b)) &= (\mathcal{F}_{\mathrm{f}}(l) \cup \mathcal{F}(b)) \setminus \mathcal{B}_{\mathrm{f}}(l) \\
\mathcal{F}(\textbf{let}\ (v, e, b)) &= \mathcal{F}(e) \cup (\mathcal{F}(b) \setminus \{v\}) \\
\mathcal{F}(\textbf{app}\ (x_1, x_2)) &= \mathcal{F}_{\mathrm{v}}(x_1) \cup \mathcal{F}_{\mathrm{v}}(x_2) \\
\mathcal{F}(\textbf{record}\ (x_1, \ldots)) &= \bigcup_{i=1,\ldots} \mathcal{F}_{\mathrm{v}}(x_i) \\
\mathcal{F}(\textbf{select}\ (i, x)) &= \mathcal{F}_{\mathrm{v}}(x) \\
\mathcal{F}(\textbf{if}\ (x, t, e)) &= \mathcal{F}_{\mathrm{v}}(x) \cup \mathcal{F}(t) \cup \mathcal{F}(e) \\
\\
\mathcal{F}_{\mathrm{f}}((f_1, v_1, e_1), \ldots) &= \bigcup_{i=1,\ldots} \{\mathcal{F}(e_i) \setminus \{v_i\}\} \\
\mathcal{B}_{\mathrm{f}}((f_1, v_1, e_1), \ldots) &= \{f_1, \ldots\}
\end{aligned}
$$

λ-contract

I found it useful to run a simple optimizer, called λ-contract, over the code before attempting to split it. By un-nesting `let`-bindings and performing other code rearrangements, it brings the code into the general form expected by the splitting algorithm. It also performs value propagation, β-contractions, and some dead code elimination. After the code has been straightened out, one can rely on simple syntactic cues when looking for variable bindings to be exported in symbolic form as part of the inlinable portion of the split.

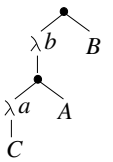
For example, if the right-hand side of a variable binding is another binding form itself, then the order of the two can be exchanged. This technique is known as *let-floating* [JPS96]:

```
let val a =
  let val b = B
  in A end
in C end
```



⇒

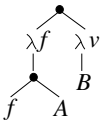
```
let val b = B
in let val a = A
   in C end
end
```



No α-conversion is necessary here because variable names are guaranteed to be distinct.

When abstraction and application are not adjacent, a β-reduction can make the `let`-binding apparent:

```
let val f =
  fn v => B
in f A end
```



⇒

```
let val v = A
in B end
```

Such transformations move more bindings to "top level," which improves the performance of λ-splitting.

It seems unfortunate that λ-contract duplicates some optimizations that are also performed by SML/NJ's CPS optimizer. The λ-splitter runs early to be able to take advantage of type information that is not present in later stages of the current SML/NJ compiler. Perhaps, in compilers where the intermediate language is typed at all times [TMC+96], λ-splitting can be moved to such a later stage, and the need for λ-contract and its associated redundancy can be eliminated.

Call counting

One major source of inefficiencies related to separate compilation is the need for a generic function call protocol, which has to be used whenever the compiler is unable to consistently identify all call sites of a given function or all functions callable at a given call site. In particular, this is always the case when the function is defined in one compilation unit and used in another.

Cross-module inlining, as proposed here, is an attempt to improve the situation by moving function definitions into the modules where they are used and vice versa. But unless one abandons the idea of separate compilation completely, there will always be some functions that are not defined where they are called. Therefore, there is a danger that cross-module inlining will make things worse than they were before! To illustrate this, consider a compilation unit B that contains a call to a function `f`; `f` is defined in compilation unit A and calls `g` as well as `h`, which are also defined in A.

```
(* compilation unit A *)
fun g () = ...
fun h () = ...
fun f () = (... g () ... h () ...)

(* compilation unit B *)
... f () ...
```

If "for efficiency" only `f`'s definition is moved into B, then for every call to the original `f` that had to use the generic, non-optimized protocol one now has to make two such calls to `g` and `h`:

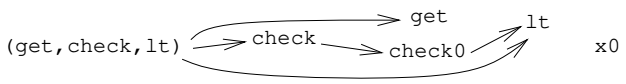
```
(* compilation unit A *)
fun g () = ...
fun h () = ...

(* compilation unit B *)
... (... g () ... h () ...) ...
```

To avoid this effect, one should keep track of the number of runtime function calls that, for the reason that caller and callee are separated and put into different source files, use the non-optimized protocol. This quantity cannot be calculated precisely, but one can use a static estimator $\mathcal{N}(\ldots)$, which is able to identify many of the bad cases. Moreover, inherent imprecisions do not have any bearing on correctness of the overall algorithm ($\rightarrow$ Section 5.5.)

B-decomposition algorithm

Some bindings need not appear in U_i because the variables they bind are not referenced. For a binding to be useful in U_i it must be reachable in the directed use-definition graph that contains an edge going from every use of a variable to its definition, and where the root node corresponds to the expression that constructs the export record. The use-def graph for `extremeFun` is



There is no node for `e` because `ref x0` is expansive. To avoid code blowup, the algorithm will put into U_i only a subset of the reachable portion of the graph.

The algorithm for constructing U_i proceeds as follows:

1. Calculate the initial set A of bindings that are available for inlining: all non-expansive bindings to non-functions, all bindings to small functions, and bindings derived from recursively splitting large nonrecursive functions. "Small" is defined by a tunable parameter to the algorithm. Code size is estimated using the measure $\mathcal{S}(\ldots)$ defined in section 5.5; non-expansiveness is judged using the predicate $\mathcal{P}(\ldots)$ from section 5.5. This part of the algorithm is the *Available* function of figure 5.2.

2. Construct the use-definition graph for this set (as in *UseDef* of figure 5.3).

B-decompose($w = \lambda(v_1, \ldots, v_n)$. let $bindings$ in $\vec{r}$) =
  $(E, A) \leftarrow$ Available($bindings$)
  $I \leftarrow$ ToBeInlined($A, \vec{r}$)
  $I_{\text{ord}} \leftarrow$ the ordered sublist of E whose elements are $\in I$
  $(x_1, \ldots, x_k) \leftarrow \mathcal{F}$(let I_{ord} in $\vec{r}$)
  return ($w_e = \lambda(v_1, \ldots, v_n)$.let E in $(x_1, \ldots, x_k)$,
    $w_i = \lambda(x_1, \ldots, x_k)$.let I_{ord} in $\vec{r}$,
    w=$\lambda(v'_1, \ldots, v'_n)$. $w_i(w_e(v'_1, \ldots, v'_n))$)

S-decompose($w = \lambda(v_1, \ldots, v_n)$. let $bindings$ in $\vec{r}$) =
  $(E, A) \leftarrow$ Available($bindings$)
  $I \leftarrow$ ToBeInlined($A, \vec{r}$)
  $I_{\text{ord}} \leftarrow$ the ordered sublist of E whose elements are $\in I$
  $(x_1, \ldots, x_k) \leftarrow \mathcal{F}$(let I_{ord} in $\vec{r}$) $\setminus \{v_1, \ldots, v_n\}$
  return ($w_e = \lambda(v_1, \ldots, v_n)$.let E in $(x_1, \ldots, x_k)$,
    $w_i = \lambda(v_1, \ldots, v_n).\lambda(x_1, \ldots, x_k)$.let I_{ord} in $\vec{r}$,
    $w = \lambda(v'_1, \ldots, v'_n).w_i(v'_1, \ldots, v'_n)(w_e(v'_1, \ldots, v'_n))$)

Available($b_1, \ldots, b_m$) =
  $E \leftarrow b_1, \ldots, b_m$
  for each $b \in E$
    if $\mathcal{S}(b) < K$
      $A \leftarrow A \cup \{b\}$
    else $(b_e, b_i, b') \leftarrow$ S-decompose(b)
      In the list E, replace b by b_e, b_i, b'
      $A \leftarrow A \cup \{b_i, b'\}$
    else if $\mathcal{P}(b)$
      $A \leftarrow A \cup \{b\}$
  return (E, A)

Figure 5.2: **λ-splitting algorithms.** Part 1.

UseDef(b) =
  The set of bindings $v = M$ such that $v \in \mathcal{F}(b)$

ToBeInlined($A, \vec{r}$) =
  $Q \leftarrow \{\vec{r}\}$
  $Q' \leftarrow \{\ \}$
  while $Q \neq \{\ \}\ \ \wedge\ \ \sum_{b \in B} \mathcal{S}(b)\ <\ K'$
    let b be the element of Q with smallest $\mathcal{S}(b)$
    $A \leftarrow A \setminus \{b\}$
    $Q \leftarrow Q \setminus \{b\}$
    $B \leftarrow B \cup \{b\}$
    $Q' \leftarrow Q' \cup (\text{UseDef}(b) \cup A)$
    if $\mathcal{N}(B) < \mathcal{N}(I)$
      $I \leftarrow B$
    if $Q = \{\ \}$
      $Q \leftarrow Q'$
      $Q' \leftarrow \{\ \}$
  return I

Figure 5.3: **λ-splitting algorithms.** Part 2.

3. Perform a breadth-first search on the graph starting at the root (the export record $\vec{r}$). Within each level of the graph, search is biased to favor smaller expressions. It stops when either the entire graph has been traversed or the total size of nodes searched exceeds a pre-set threshold. This step automatically eliminates all dead definitions. The greedy approach favors variables that are used by shorter use-def chains from the root node; this is called a *distance heuristic*. It also provides a cap for the total size of U_i, thereby preventing nonlinear increases in total code size. (See the *ToBeInlined* function of figure 5.3.)

4. From the space traversed by the greedy breadth-first search the algorithm picks the set of bindings for U_i that has the smallest call count $\mathcal{N}(\ldots)$.

5. The bindings in the set picked for constructing the body of U_i are used in the same

$\vec{r}$ is the expression that constructs the export record and serves as the starting point for the breadth-first search in function *ToBeInlined*.

E is the ordered list of bindings to be kept in U_e; the intramodular optimizer may later delete dead bindings, but this is not part of the splitting algorithm.

A is the set of bindings available for inlining.

B is the current candidate for the inlinable part U_i.

I is the best candidate seen so far.

Q, Q' are the two parts of the breadth-first search queue. The separation enables us to implement a bias in favor of smaller bindings within each level of the breadth-first search tree.

K is the maximum size of any function body that can be inlined without being recursively split.

K' is the maximum size of the inlineable code U_i.

$\mathcal{F}$ computes the free variables of an expression (see section sec:lang).

$\mathcal{P}$ tells whether an expression is nonexpansive (see section 5.5).

$\mathcal{S}$ tells the size of an expression (see section 5.5).

$\mathcal{N}$ approximates how many out-of-module calls a set of bindings makes.

Figure 5.4: **Legend to the algorithm in figures 5.2 and 5.3.**

order that they appeared in within U's code. This ensures correct scoping for bound variables.

6. Any remaining free variables will not be inlined. They will be calculated by U_e and passed at link-time (as function arguments) to the client (see *B-decompose* in figure 5.2).

5.4 Functions

Abstractions are syntactic values, so large functions will be large even in A-normal form. Inlining small functions is part of the overall goal, but big functions pose a problem. While it is possible to move the entire code of any function into U_i without sacrificing correctness, one cannot do this indiscriminately because it would severely inflate the code. On the other hand, when it is not feasible to move the entire function, then one would like to make some of its parts available for inlining ($\rightarrow$ Figure 5.1). After all, functors are encoded as functions, and highly functorized code could hardly benefit from cross-module inlining if this problem is not solved. As I will explain, the λ-splitting algorithm can be applied recursively to function bodies.

Recursive functions

The algorithm does not attempt to take recursion apart; it either moves entire clusters of mutually recursive functions into U_i when the cluster as a whole is deemed small enough, or it leaves them completely alone. To be able to do this at as fine a granularity as possible, it calculates strongly connected components of the use-definition graph for every recursive `let`. Each component is then considered separately.

In most cases, a strongly connected component of functions represents a genuine loop

or similar repetitive computation. This should not be spread across multiple compilation units because the overhead of cross-module function calls would be amplified considerably.

Functors and nonrecursive functions

B-decomposition works because compilation units are represented as functions. One could think of these compiler-constructed functions as implicit functors. From this observation one can derive the idea for dealing with actual functors and other large functions—they will be split as well. The resulting expansive and inlinable parts are placed into the corresponding expansive and inlinable portions of the surrounding function or compilation unit.

For correctness one must be careful that the splitting algorithm will never place expansive code into the inlinable portion of a compilation unit. When splitting functions this restriction does not exist, because side effects can only occur at the time the function is called. Therefore, side effects will be duplicated only if the function in question is explicitly invoked multiple times, which then makes this the correct behavior as mandated by the language definition.

Nevertheless, it is convenient to maintain the invariant that the inlinable portion of *any* split is nonexpansive. It frees the algorithm from having to pay attention to the ordering of effects within a function. Also, it guarantees that invocations of any inlinable portions themselves will be free of effects. Thus, it becomes possible to relax the notion of non-expansiveness (normally, function invocation is always considered expansive). Such invocations can also be moved into the inlinable portion of the enclosing function (→ Section 5.5).

Recursive decomposition. As a first attempt, the same technique, B-decomposition, that is used to split the functions representing entire compilation units can also be utilized when

splitting functions too large to be placed into the inlinable portion. This idea recursively extends to large functions encountered within large functions and so forth.

In the introductory example, if one assumes `check0` and `check` to be too big, then the "functor" `extremeFun` would be split like this:

```
fun extremeE (lt, x0) =
let val e = ref x0
    fun get () = !e (* dead *)
    fun check0 x =
        if lt (!e, x) then e := x
        else ()
    fun check [] = ()
      | check (h ::  t) = (check0 h; check t)
in (e, check, lt) end

fun extremeI (e, check, lt) =
let fun get () = !e
in (get, check, lt) end

fun extremeFun args =
    extremeI (extremeE args)
```

Notice that `extremeFun` is the *B-composition* of `extremeI` and `extremeE`. Definitions for `extremeI` and the reconstructed `extremeFun` will be exported as part of U_i, allowing the client to inline calls to `get` and rewrite them as accesses to reference cell `e`.

S-decomposition. Functor splitting should provide three benefits:

1. Parts of the functor's body become part of U_i and can, therefore, be inlined into client code.
2. Existing simple connections between argument and result remain visible in U_i. Since the functor argument is supplied where the functor is instantiated, this will enable client code to inline parts of a functor's result if they correspond to inlinable parts of the argument.

3. Part or all of the functor's argument are inlined into the functor's body.

Unfortunately, only the first point is addressed by B-decomposition. At first this may seem surprising because it works so nicely at the level of compilation units. What is it that distinguishes explicit functors from the ones the compiler constructs implicitly? In SML, if a compilation unit U^2 refers to U^1, then for reasons of type checking U^1 must be compiled before U^2. This means that the code for constructing U^2's input is already available when U^2 is compiled. In other words, the compiler really compiles $U^2 \circ U^1_i$ and not U^2.

With functors the situation is different. Functors are compiled separately from the code that constructs their arguments. But I want actual functor arguments to be propagated to functor bodies, so they can be inline-expanded there, and functor arguments to be propagated to functor results (for clients to use). B-decomposition propagates code only from functor bodies to functor results.

A variation on B-decomposition, called *S*-decomposition, addresses points 1 and 2 above. The functor argument is known at the time when function `extremeI` is invoked, so instead of first funneling it all the way through `extremeE` one can simply pass it directly to `extremeI`. Therefore, the S-decomposition for $F : I \rightarrow E$ is:

$$(T,\ F_i : I \rightarrow T \rightarrow E,\ F_e : I \rightarrow T)$$

such that $F = SF_iF_e$. S denotes the stronger version of composition, which distributes the argument to both functions: $Sfgx = fx(gx)$.

S-decomposition turns the example program into the following code, where F_i (here, `extremeI`) establishes an explicit link between the functor's argument and `lt`:

```
fun extremeE (lt, x0) =
let val e = ref x0
    fun get () = !e (* dead *)
    fun check0 x =
        if lt (!e, x) then e := x
        else ()
    fun check [] = ()
      | check (h ::  t) = (check0 h; check t)
in (e, check) end

fun extremeI ((lt, x0), (e, check)) =
let fun get () = !e
in (get, check, lt) end

fun extremeFun args =
    extremeI (args, extremeE args)
```

Differences between the algorithms for B- and S-decomposition are minor. S-decomposition can use exactly the same methodology and, in fact, share most of the implementation of B-decomposition as well. There is one small twist: At the point where the free variables of F_i are collected, S-decomposition explicitly excludes the original formal argument of F because it will be passed to F_i directly (see the *S-decompose* function of figure 5.2).

1-decomposition. Actual functor arguments can be inline-expanded in the functor body only if they are used in the inlinable portion of the functor split. Only then will both definition and use of the argument be available to the compiler at the same time. In the example, even S-decomposition does not promote the definition of `check0` into `extremeI`. Therefore, the comparison predicate will not be inlined there. The only mention of `check0` is in `check`, which itself is not in `extremeI`.

Still, `check0` is bound to a syntactic value. λ-splitting could factor it out and move it to `extremeI`. From there it will be passed *as an argument* to `extremeE`. The resulting code was shown in section 5.1.

When splitting a functor F one can generally leave it up to F_i how F_e is invoked. Even though F_e is compiled somewhere else and will be treated as a black box, its interface can still be chosen arbitrarily. This can be modeled by passing F_e itself, not its result, to F_i. Thus, a $\underline{1}$-decomposition of $F : I \to E$ is a triplet

$$(T,\ F_i : T \to I \to E,\ F_e : T)$$

such that $F = \underline{1} F_i F_e = F_i F_e$.[2] Normally T will itself be of the form $T_1 \to T_2$ for some T_1 and T_2.

The equality $F = F_i F_e$ is used to recover F from its pieces. Any occurrence of F in the client can be replaced with $F_i F_e$. But the code of F_i will then be available, so one can immediately β-reduce $F_i F_e$ and work with the result instead of F_i itself. I already did this when I presented the introductory example, which explains why `extremeE` is not being passed to `extremeI` as an argument but instead gets invoked directly.

Both B- and S-decompositions can be viewed as special cases of the more general approach of $\underline{1}$-decomposition. In particular, for any X-decomposition (T, F_i, F_e) of $F : I \to E$ (where X is either B or S) the triplet $(I \to T, X F_i, F_e)$ forms such a corresponding $\underline{1}$-decomposition.

I do not show an algorithm for $\underline{1}$-decomposition because I have not found a good set of heuristics to separate the "core" of a function (or functor) from the non-inlinable body (as illustrated in figure 5.1). Therefore, my implementation cannot quite transform `extremeFun` as shown in section 5.1, but can transform it via B- and S-decomposition.

[2] $\underline{1} = \lambda f.\lambda x.(f x)$ is the Church-numeral **1**.

5.5 Heuristics

The algorithms for B- and S-decomposition rely on my definition of expansiveness, on heuristics for measuring code size, and on cross-module call counts. I will now show the corresponding formal definitions.

Expansiveness

Expressions can be moved into the inlinable portion of a split if the predicate $\mathcal{P}(\ldots)$ ("permitted") holds for them:

$$\begin{aligned}
\mathcal{P}(\textbf{val}\ v) &= T \\
\mathcal{P}(\textbf{fn}\ (v, e)) &= T \\
\mathcal{P}(\textbf{fix}\ (l, e)) &= \mathcal{P}(e) \\
\mathcal{P}(\textbf{let}\ (v, e, b)) &= \mathcal{P}(e) \wedge \mathcal{P}(b) \\
\mathcal{P}(\textbf{app}\ (\textbf{pure}\ p, x)) &= T \\
\mathcal{P}(\textbf{app}\ (\textbf{impure}\ p, x) &= F \\
\mathcal{P}(\textbf{app}\ (\textbf{var}\ v, x)) &= \texttt{purefun}\ (v) \\
\mathcal{P}(\textbf{record}\ l) &= T \\
\mathcal{P}(\textbf{select}\ (i, x)) &= T \\
\mathcal{P}(\textbf{if}\ (x, t, e)) &= F
\end{aligned}$$

The use of the auxiliary function `purefun` deserves some explanation. Normally all function applications are deemed expansive and, thus, are not allowed to occur within the inlinable part. However, occasionally one can be sure that certain functions have no side effects, in which case there is no harm in moving a corresponding application. In particular, this happens when the cross-module inlining algorithm splits functions recursively, because it always maintains the invariant that the inlinable portion of a split is effect-free.

To illustrate this point, consider a compilation unit that defines a functor and then also instantiates it:

```
fun F x = ...
val m = F a
```

S-decomposition on `F` transforms this into:

```
fun Fe x = ...
fun Fi (fe, x) = ...
fun F x = Fi (Fe x, x)
val m = F a
```

It is now possible to export the definitions for `Fi` and `F`, but one still cannot also export `m`'s definition because its right-hand side is an application. This is unfortunate because `m` is constructed by applying `Fe` as well as `Fi`, and `Fi` itself was constructed in such a way that its applications should become inlinable. In other compilation units that instantiate `F`, this will not pose as a problem because λ-contract will turn such applications into explicit **let**-bindings. But within the same compilation unit λ-contract has already finished its work before `F` was ever split into `Fe` and `Fi`. Therefore, the implementation keeps track of such cases and β-reduces the call to `F` on the fly:

```
fun Fe x = ...
fun Fi (fe, x) = ...
fun F x = Fi (Fe x, x)
val tmp = Fe a
val m = Fi (tmp, a)
```

`Fi` refers to the inlinable portion of a split. Therefore, one can assert: $\texttt{purefun}(\texttt{Fi}) = T$. With this, the definition for `m` can be exported and inlined into other compilation units.

Currently, purefun$(\ldots)$ returns *false* in all other cases. It would be possible to further relax this by checking the bodies of other functions to see whether or not they are expansive. I have not done this because in many cases such function applications are already β-reduced during the λ-contract phase.

Size estimates

I use a simple, syntax-driven size estimate $\mathcal{S}(\ldots)$ to limit the amount of code that will be exported to be inlined into other compilation units. Necessarily, this estimate is imprecise because subsequent optimization can radically alter the code.

The values $A, B, C, D, E, F, G, H, I, K, L$ as well as the function $\mathcal{S}_{\mathrm{p}}(\ldots)$ are adjustable parameters to the algorithm and are chosen to approximately reflect the relative cost of implementing the respective language feature.

$$
\begin{aligned}
\mathcal{S}_{\mathrm{v}}(\textbf{var}\ v) &= A \\
\mathcal{S}_{\mathrm{v}}(\textbf{const}\ c) &= B \\
\mathcal{S}_{\mathrm{v}}(\textbf{pure}\ p) &= C \\
\mathcal{S}_{\mathrm{v}}(\textbf{impure}\ p) &= C \\
\\
\mathcal{S}(\textbf{val}\ x) &= \mathcal{S}_{\mathrm{v}}(x) \\
\mathcal{S}(\textbf{fn}\ (v, e)) &= \mathcal{S}(e) + D|\mathcal{F}(e) \setminus \{v\}| + E \\
\mathcal{S}(\textbf{fix}\ (l, b)) &= \mathcal{S}_{\mathrm{f}}(l) + D|\mathcal{F}_{\mathrm{f}}(l) \cup \mathcal{B}_{\mathrm{f}}(l)| + \\
&\quad \mathcal{S}(b) + F \\
\mathcal{S}(\textbf{let}\ (v, e, b)) &= \mathcal{S}(e) + \mathcal{S}(b) + G \\
\mathcal{S}(\textbf{app}\ (\textbf{pure}\ p, x)) &= \mathcal{S}_{\mathrm{p}}(p) + \mathcal{S}_{\mathrm{v}}(x) \\
\mathcal{S}(\textbf{app}\ (\textbf{impure}\ p, x)) &= \mathcal{S}_{\mathrm{p}}(p) + \mathcal{S}_{\mathrm{v}}(x) \\
\mathcal{S}(\textbf{app}\ (x_1, x_2)) &= \mathcal{S}_{\mathrm{v}}(x_1) + \mathcal{S}_{\mathrm{v}}(x_2) + H \\
\mathcal{S}(\textbf{record}\ (x_1, \ldots)) &= I + \sum_{i=1,\ldots} \mathcal{S}_{\mathrm{v}}(x_i) \\
\mathcal{S}(\textbf{select}\ (i, x)) &= \mathcal{S}_{\mathrm{v}}(x) + K \\
\mathcal{S}(\textbf{if}\ (x, t, e)) &= \mathcal{S}_{\mathrm{v}}(x) + \mathcal{S}(t) + \mathcal{S}(e) + L \\
\\
\mathcal{S}_{\mathrm{f}}((f_1, v_1, e_1), \ldots) &= \sum_{i=1,\ldots} \mathcal{S}(e_i)
\end{aligned}
$$

For Standard ML of New Jersey, I use the following coefficients:

A	B	C	D	E	F	G	H	I	K	L
0	0	0	2	1	2	0	5	1	1	4

$S_{\mathrm{p}}(p)$ is between 1 and 4, for most p.

Call counting

I want to estimate the number of function calls that cross between module boundaries. Of course, this number depends on the specifics of client code, which is not available at the time of conducting the analysis. But client code can only call the functions that were originally exported by the compilation unit. Therefore, one can "normalize" the estimate, for example, by setting the counts for these functions to 1.

In the absence of cross-module inlining, the compilation unit that only exports one function `f` would be assigned the estimate 1:

```
local
  fun g () = ...
  fun h () = ...
in
  fun f () = (... g () ... h () ...)
end
```

But when λ-splitting moves `f`'s definition into the client, then the estimate increases to 2 because now there are calls to `g` and `h` that cross compilation unit boundaries. Function `f` might also call other functions in a loop, it may pass them to higher-order functions, and so on. In these cases it becomes difficult to know the actual number of calls that will occur at run-time. I capture this by estimating such a number as ∞. Thus, the domain $\mathcal{C}$ of call counts is the set of elements $\{0, 1, 2, \ldots, \infty\}$.

Adding call counts:

$$\begin{aligned} y \oplus x &= x \oplus y \\ x \oplus y &= x + y; \qquad x, y \neq \infty \\ \infty \oplus x &= \infty \end{aligned}$$

Multiplication with ∞:

$$\begin{aligned} 0 \odot \infty &= 0 \\ x \odot \infty &= \infty \end{aligned}$$

Comparison:

$$\begin{aligned} x &\preceq y; \quad x, y \neq \infty; x \leq y \\ x &\preceq \infty \\ \infty &\not\preceq x; \quad x \neq \infty \end{aligned}$$

Least upper bound:

$$\begin{aligned} x \sqcap y &= x; \quad x \not\preceq y \\ x \sqcap y &= y; \quad x \preceq y \end{aligned}$$

Call count estimation: $\mathcal{N}_B(e)$ estimates the number of function invocations made via globally free variables while evaluating e. Included in the result is the number of such function invocations that will be suspended in e's value. Thus, the result of $\mathcal{N}_B(e)$ is in the domain $\mathcal{C} \times \mathcal{C}$. I use the notation $\nu \downarrow 1$ and $\nu \downarrow 2$ to extract first and second field from such a pair ν.

An environment B maintains information about variables bound by the "context expression" that surrounds e. It maps variables to the estimate for the number of calls suspended in the respective runtime values. If nothing else is known, then B binds a variable to $\diamond$.

$$\mathcal{N}(e) = \mathcal{N}_{\lambda x.\diamond}(e)$$

The estimate $\mathcal{N}_B(e)$ is given by a recursive definition directed by the syntactic structure of expression e:

$$\begin{array}{rcl}
\mathcal{N}_{\mathbf{V},B}(\textbf{var}\ v) & = & (0,1); \quad B(v) = \diamond \\
\mathcal{N}_{\mathbf{V},B}(\textbf{var}\ v) & = & (0,w); \quad B(v) = w \\
\mathcal{N}_{\mathbf{V},B}(x) & = & (0,0) \\
 & & \\
\mathcal{N}_B(\textbf{val}\ x) & = & \mathcal{N}_{\mathbf{V},B}(x) \\
\mathcal{N}_B(\textbf{fn}\ (v,e)) & = & (0, \mathcal{N}_{B[v\to 0]}(e) \downarrow 2) \\
\mathcal{N}_B(\textbf{fix}\ (l,b)) & = & \mathcal{N}_{B[f_1\to u,\ldots,f_n\to u]}(b); \\
 & & \quad l = (f_1,v_1,e_1),\ldots,(f_n,v_n,e_n), \\
 & & \quad B' = B[f_1 \to 0,\ldots,f_n \to 0], \\
 & & \quad u = (\bigoplus_{i=1}^{n} \mathcal{N}_{B'[v_i\to 0]}(e_i)) \odot \infty \\
\mathcal{N}_B(\textbf{let}\ (v,e,b)) & = & (n_1 \downarrow 1 \oplus n_2 \downarrow 1, n_2 \downarrow 2); \\
 & & \quad n_1 = \mathcal{N}_B(e), n_2 = \mathcal{N}_{B[v\to n_1\downarrow 2]}(b) \\
\mathcal{N}_B(\textbf{app}\ (\textbf{var}\ f,x)) & = & ((n_x \downarrow 2 \odot \infty) \oplus 1, n_x \downarrow 2); \\
 & & \quad B(f) = \diamond, n_x = \mathcal{N}_{\mathbf{V},B}(x) \\
\mathcal{N}_B(\textbf{app}\ (\textbf{var}\ f,x)) & = & (w, w \oplus x_s \downarrow 2); \\
 & & \quad B(f) = w, n_x = \mathcal{N}_{\mathbf{V},B}(x) \\
\mathcal{N}_B(\textbf{app}\ (x,y)) & = & \mathcal{N}_{\mathbf{V},B}(y) \\
\mathcal{N}_B(\textbf{record}\ (x)) & = & \mathcal{N}_{\mathbf{V},B}(x) \\
\mathcal{N}_B(\textbf{record}\ (x,y,\ldots)) & = & (n_x \downarrow 1 \oplus n_r \downarrow 1, n_x \downarrow 2 \sqcap n_r \downarrow 2); \\
 & & \quad n_x = \mathcal{N}_{\mathbf{V},B}(x), \\
 & & \quad n_r = \mathcal{N}_B(\textbf{record (y, \ldots)}\ ) \\
\mathcal{N}_B(\textbf{select}\ (i,x)) & = & \mathcal{N}_{\mathbf{V},B}(x) \\
\mathcal{N}_B(\textbf{if}\ (x,t,e)) & = & (n_t \downarrow 1 \oplus n_e \downarrow 1, n_t \downarrow 2 \sqcap n_e \downarrow 2); \\
 & & \quad n_t = \mathcal{N}_B(t), n_e = \mathcal{N}_B(e)
\end{array}$$

5.6 Implementation and results

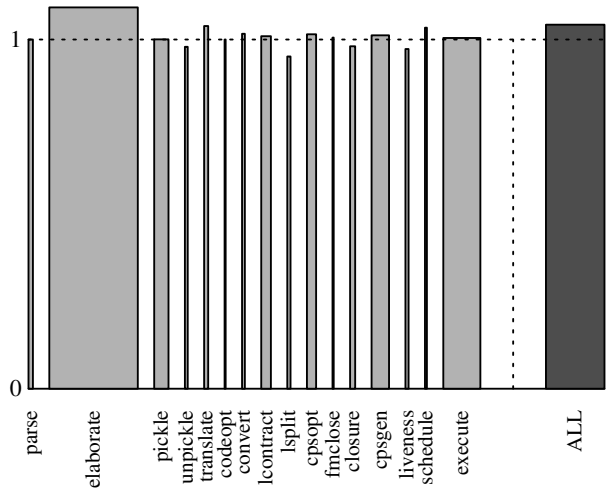
I implemented λ-splitting (B-decomposition for compilation units and recursive S-decomposition for functions) in SML/NJ. For the general framework only very small modifications to existing code in the compiler were necessary. In particular, I did not have to modify either the front-end (including symbol table management), the optimizer, or the various architecture-specific back-ends. λ-splitting fits neatly between existing compiler phases.

I changed the format of object files to accommodate symbolic λ-code and adjusted SML/NJ's compilation manager [Blu95] accordingly. The cutoff-recompilation heuristic must take into account that compilation units are connected via static environments (i.e., symbol tables) and inlinable code. This does not change the shape of the dependency graph, but it has an impact on how modifications to one unit propagate to other dependent units.

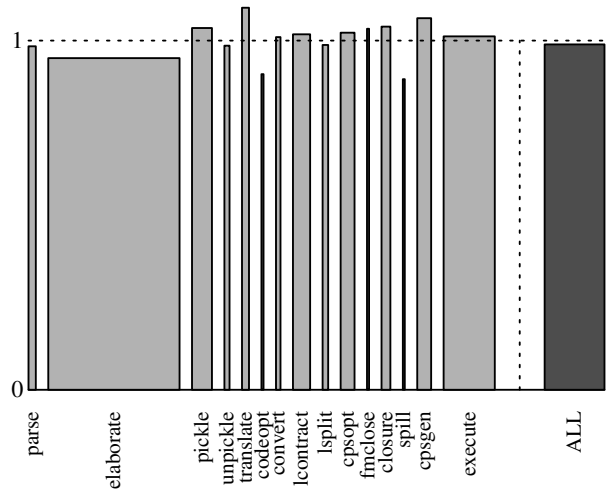
Many widely used SML benchmarks cannot be used to measure cross-module inlining because they contain only a single compilation unit. Therefore, I applied λ-splitting to SML/NJ's compiler itself and used several of its phases as my benchmarks. The compiler is a 100,000 line SML program comprised of more than 600 individual compilation units (source files). With a successful test of this magnitude, I am confident of the correctness and viability of the implementation.

Problems with λ-contract

In my prototype, the introduction of λ-contract had a disturbing effect. In some cases execution slows down ($\rightarrow$ Figure 5.5). The numbers indicate that whether or not λ-contract is advantageous on its own depends on architecture and benchmark. The overall result for my benchmarks is dominated by `elaborate`, which improves on the Pentium [Cor93] but slows down on the Alpha [Sit92, Dig92].



DEC Alpha 21064



Intel Pentium

Figure 5.5: **λ-contract on Alpha and Pentium.** Individual compiler passes of SML/NJ—light bars in the graphs—serve as benchmarks for my study. With the exception of the dark bar on the right, which shows overall performance of the whole compiler, width was chosen proportional to the time spent in each phase. The heights of the bars indicate speedup or slowdown caused by λ-contract (shorter is better). λ-contract is not the focus of this work; it is an auxiliary transformation that allows my splitting algorithm to find many more opportunities for cross-module inlining. The numbers here show that, depending on architecture and benchmark, λ-contract has sometimes positive, sometimes negative effects on running time. The *lcontract* and *lsplit* bars graphically illustrate that the algorithms presented in this paper do not make up a large portion of compilation time. → Tables C.3 and C.4

To explain this, I note that β-contractions can increase the size of certain closures, and if such closures are constructed repeatedly, for example in a loop, then execution time may go up.

```
let val f =
   fn x => g (a,b,c,d,e,x)
in let fun loop () =
       (...
         (fn y => f (z,y))
        ...loop())
   in
      loop ()
   end
end
```

In this example, let us replace the variable `f` in `(f (z, y))` with its definition:

```
let fun loop () =
    (...
      (fn y => g (a,b,c,d,e, (z, y)))
     ...loop())
in
   loop ()
end
```

The anonymous function inside `loop` now has seven free variables (those are: `g`, `a`, `b`, `c`, `d`, `e`, `z`) while before it had only two (`f` and `z`). This increases the size of the associated closure object. If, as is the case in my example, the function occurs inside a loop and, therefore, its closure must be constructed repeatedly, then the increase in closure size can become costly at runtime. I hope to improve the situation with a better closure-conversion algorithm, much later in the compiler after λ-contract, λ-splitting, and intramodular optimization.

I conducted an experiment where the compiler invoked λ-contract but not also λ-splitting. Compile-time switches were used to independently enable or disable β-contractions and let-floating [JPS96], which are part of λ-contract. I found that either "op-

timization" can sometimes lead to slightly diminished performance, while the remaining components of λ-contract never led to a slowdown and often improved execution speed. It is my belief that this supports the explanation given above. Unfortunately, the interactions between the various phases of the compiler are very complex, and I was not able to find a small code example that clearly isolates and demonstrates the effect.

Reordering `let`-bindings in itself does not change the size of any closure. It only moves bound variables closer to the expressions they are bound to. However, this may enable later phases of the compiler's CPS optimizer to recognize and reduce more β-redexes, which, again, may cause some closures to become bigger.

The charts in figure 5.5 show significant differences between different machine architectures. While let-floating and β-reductions as part of λ-contract almost uniformly lead to reduced performance an the Alpha, one often cannot find such behavior on the Pentium. But SML/NJ's code generators for these two architectures are also very different.

For example, since there are almost no general-purpose registers available on the Pentium, SML/NJ uses a number of "pseudo"-registers. Pseudo-registers are registers that are simulated in main memory. In this situation it may be that additional register spills are not as detrimental because the associated memory traffic only replaces other memory traffic (pseudo-register moves), while on machines with large register files it is expensive to substitute memory operations for operations that only involve (genuine) registers.

To obtain a better idea of how much cross-module inlining itself affects overall performance, I use SML/NJ with λ-contract enabled as the baseline for my comparisons. This compiler already does aggressive intramodular inlining [App92]; I measure only the increased performance from inlining across compilation unit boundaries and from pulling inlinable code out of higher-order functions.

Timing results from λ-splitting

Figure 5.6 shows timing results for running the benchmarks with λ-splitting enabled. The numbers reported correspond to compiler passes that took at least one second of user time. The net speedup of 8% for the Alpha is reduced to only 4% if one takes the effects of λ-contract into account.

On the Pentium I find a somewhat different picture. λ-contract seems to have a positive net impact here, although some phases of the compiler also suffer a significant slowdown. Overall speedup due to cross-module inlining is at 6% with and 5% without counting effects from λ-contract.

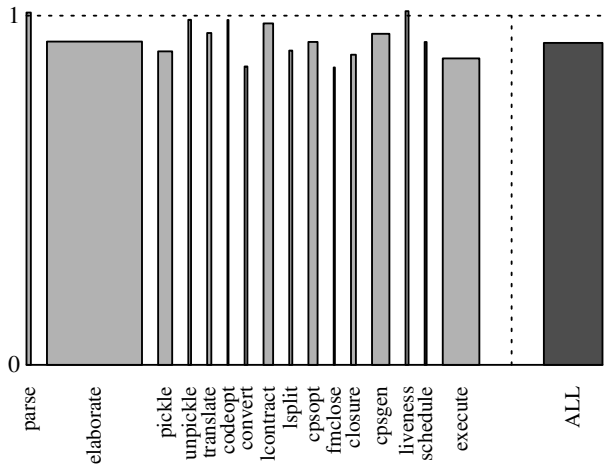
I believe my results are not due to "random" variations in cache-conflict performance. It has been shown that functional programs with properly tuned garbage collectors [Rep93] are not subject to many data-cache conflicts [Rei94, GA95], and an informal inspection of the results did not reveal the kind of instruction-cache conflict variability [App92, p. 194] to which SML/NJ was subject on previous-generation architectures.

SML/NJ without ad-hoc inlining

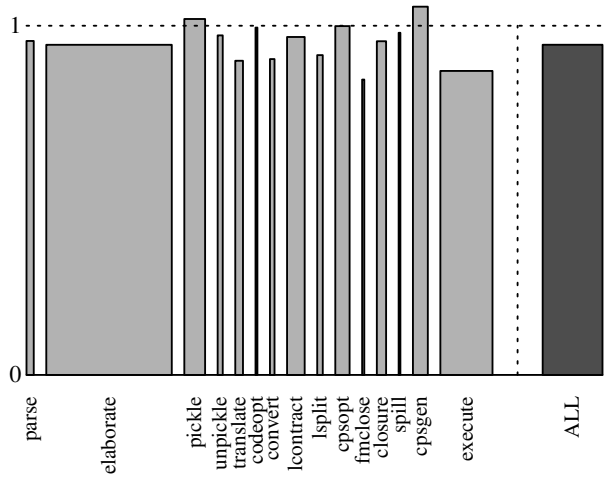
Cross-module inlining is so crucial for performance that the implementors of SML/NJ decided to compromise abstraction and modularity in some places. For a selection of the most important operations inline-expansion is absolutely necessary.

To achieve a limited form of cross-module inlining, the compiler treated certain variable bindings specially. If the right-hand side was known to be in a pre-selected set of primitive operators, then this fact was recorded as part of the static information available for the variable bound on the left-hand side. In effect, inlining information was treated in a fashion similar to types.

Unfortunately, the mechanism only applied to a fixed set of pre-selected values. Fur-



DEC Alpha 21064



Intel Pentium

Figure 5.6: **Improvement due to cross-module inlining.** λ-splitting yields improvement on almost all benchmarks, and improves total execution time by 8% on the Alpha, 5% on the Pentium. The baseline (dotted line at 1.0) shows a compiler with λ-contract but without cross-module inlining; the grey bars show runtime (smaller is better) with both λ-contract and inlining. $\rightarrow$ Tables C.1 and C.2

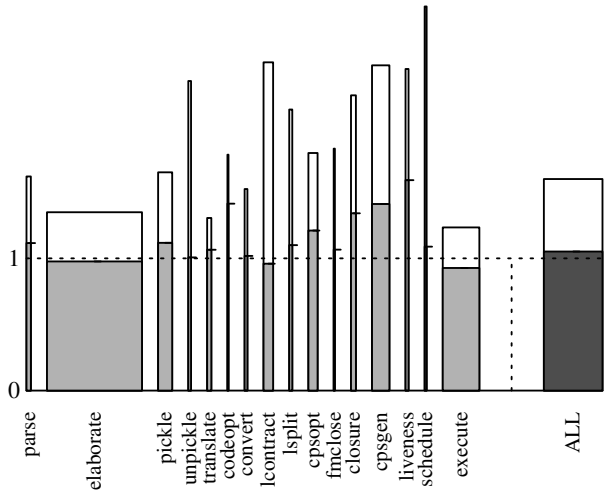


Figure 5.7: **Inlining applied to a cleanly modularized program.** I reinserted signature constraints and abstractions that had been removed by the efficiency-conscious software engineers. The white bars show how much this would have cost without cross-module inlining. The dark bars show that λ-splitting recovers essentially all of the performance cost of modular abstraction ($\rightarrow$ Table C.5).

thermore, it was not robust with respect to simple source code modifications. For example, it could be disabled by simply adding a few semantics-preserving signature constraints. The unsuspecting user might do this by accident. Even so, figure 5.6 shows that λ-splitting plus ad-hoc inlining is better than ad-hoc inlining alone.

As an experiment, I changed the compiler sources to eliminate old-style ad-hoc inlining. This elimination was done purely mechanically. I restored the signature constraints that one would normally write but that had been removed by the efficiency-conscious software engineers because they knew that such constraints would render ineffective the existing ad-hoc inlining mechanism in the compiler.

The results ($\rightarrow$ Figure 5.7) indicate that a fully automatic technique like λ-splitting can provide benefits close to those of previous ad-hoc approaches, while at the same time being more robust and less intrusive.

Comparing raw numbers shows that λ-splitting still fares a little worse than the special-

case solution (about 5%). However, the contest is not entirely fair. The ad-hoc approach implemented in SML/NJ not only propagates primitive operators from compilation unit to compilation unit, it also forces the *intra*modular optimizer to inline-expand every single use. In effect, it overrides decisions that the optimizer would have made on its own. λ-splitting does not do that. Instead, it relies entirely on the existing optimizer's heuristics. In my experiment, all the operators handled by the old approach were also propagated by λ-splitting. Therefore, a discrepancy in performance can be explained by shortcomings in intramodular optimization technology.

Complex numbers and FFT

Figure 5.8 shows the timing results for another test case. I implemented a type for representing complex numbers and their associated arithmetic operations. The resulting code (→ Appendix B.1) was compiled separately. I then used a simple recursive implementation of FFT, the Fast Fourier Transform [CLR90, chapter 32], as a client of this complex number package (→ Appendix B.2). To obtain timing measurements, I performed the following four experiments, each with and without λ-splitting turned on:

1. FFT of a 50,000-element list
2. FFT of a 50,000-element list followed by the inverse FFT of the result
3. FFT of a 100,000-element list
4. FFT of a 100,000-element list followed by the inverse FFT of the result

The entire experiment was conducted on a DEC Alpha machine. λ-splitting consistently improved running time by about 10%.

I also concatenated all sources related to my FFT benchmark and performed the same experiment with code that was compiled all at once. The differences in running time com-

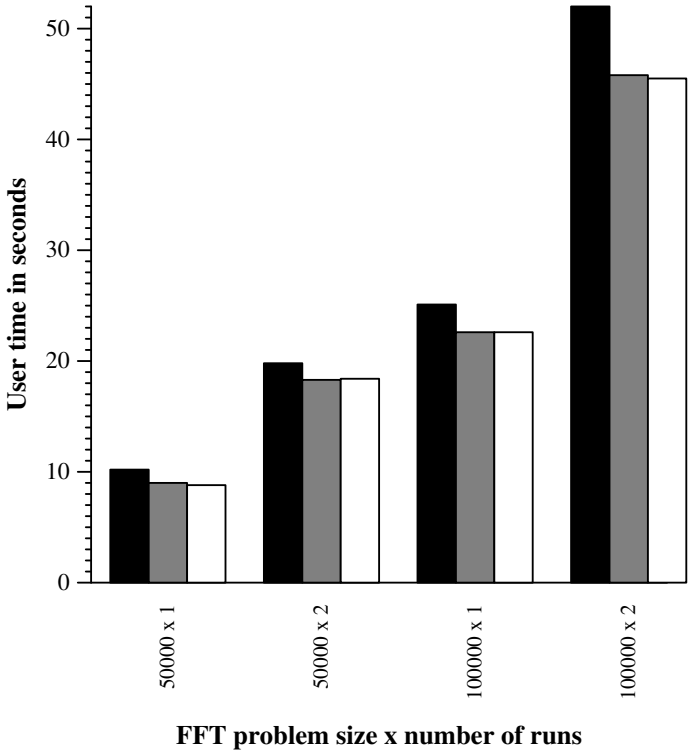


Figure 5.8: **Cross-module optimization for an implementation of FFT.** I have separated the code implementing complex numbers and their operations from a client implementation of FFT. The individual bars show the time spent on calculating the FFT of 50,000- and 100,000-element lists. In experiments labeled "$N \times 2$" I combined the FFT with an inverse FFT of the result. Black bars show the running time in seconds for the unoptimized version. Shorter grey bars indicate shorter running times for the program when compiled with λ-splitting enabled. For comparison, I also compiled the entire program as one single compilation unit. Its running times are shown in white. In this example λ-splitting achieves as much improvement as one can possibly hope for. The compiler was not able to do better, not even when it had access to the entire source code at once ($\rightarrow$ Table C.6).

pared with the cross-module optimized version are very small and well below the margin of error. Therefore, at least for small programs like the one shown, one gets as much improvement from λ-splitting as one could have hoped for.

Compile-time cost of inlining

In my preliminary implementation, the λ-contract phase takes 4.5% of total execution time, and λ-split takes 1.5%. In figure 5.6 these phases are labeled `lcontract` and `lsplit`, respectively. In addition, the intramodule CPS-optimization phase slows down because it

has more (useful!) work to do. Compile time increases by 7% overall, code size goes up by about 5%. The implementation of λ-contract could undoubtedly be improved; but even so, the cost of the cross-module inliner seems entirely reasonable.

Chapter 6

Macros

Chapter 5 presented a novel technique for performing cross-module optimizations automatically. In particular, by moving code from one compilation unit to another it was possible to rely on existing intra-modular optimization technology. But compilers that perform aggressive automatic inlining are relatively new and are rarely found except in research compilers because that is where such extensive optimizations are most needed. Modern high-level languages often express more traditional programming idioms—for example, loops—using recursion or higher-order functions.

Many programmers who work with mainstream languages still feel more comfortable fine-tuning their code by hand because performance, albeit perhaps not always strictly better, often seems to be more predictable. This can be seen as one explanation why:

```
# define SQUARE(x) ((x) * (x))
```

instead of:

```
double square (double x)
{ return x * x; }
```

is still so popular among C programmers.

Macro definitions can be seen as a way of "moving" code from one source (the C

header file) to a client (the C program file), where its use will be inline-expanded. One of the concerns in the discussion of λ-splitting was the treatment of free variables. If one views macro expansion as a form of user-controlled inlining, then it is reasonable to expect similar problems to arise.

For example, there are two occurrences of free variables in the definition:

```
# define ERROR(s) (print_error (s), exit (1))
```

Here, the free variables are `print_error` and `exit`. This C macro will not function correctly when invoked in a context where these names have been redefined as, for example, in:

```
{ char *print_error = "The printer is broken.";
  ...
  if (is_broken (printer))
     ERROR (print_error);
}
```

Such code will not work, which demonstrates that the C macro system is not capable of dealing with free variables correctly. Unfortunately, that is not the only flaw because it even fails to handle bound variables well. The `SWAP` macro defined as:

```
# define SWAP(x,y) { double temp = x; x = y; y = temp; }
```

produces incorrect code when used with an argument that happens to be named `temp`:

```
double temp = get_temperature_reading ();
⋮
SWAP (temp, x);
```

There are a number of additional, well-known problems with C-style macros. Newer language designs try to avoid them by providing direct support for inlining [ES90].

However, inlining is not the only reason why macros are used. For example, advocates of the Scheme programming language [Ce91] often stress the point that macros allow for meta-programming. The macro system provides ways of reshaping the programming

language itself, to enrich it with new syntactic constructs that capture entire programming paradigms, and make it more convenient to use. The Scheme shell *SCSH* [Shi94] provides a good example of how to tap the power of meta-programming for the purpose of scripting.

Macro systems have a long tradition in computing. Brown [Bro74] discusses how they can be used to enhance software portability by using "Descriptive Languages Implemented by Macro Processors" (DLIMP). He also describes many of the known problems with such approaches.

A notable example of a DLIMP is the implementation of SNOBOL4 in terms of a machine-independent macro language called SIL [Gri72]. SIL macros provide a layer of abstraction that allows for relatively low-cost portability. Only SIL has to be reimplemented for each new machine architecture.

This contrasts with *conditional compilation* (`#ifdef` in C), which is often used to achieve a form of limited portability. Conditional compilation takes advantage of host-language constructs that are known to have different semantics under different installations, while a DLIMP tries to guarantee that there are no such constructs by essentially providing a new, machine-independent programming language.

Hygienic macros in the style now used in Scheme are the first fairly comprehensive solution to the name capture problem in macro systems that also retains most of the meta-programming power. But not everything is well. Macro systems still have significant problems. The lack of a type system is perhaps the most severe. C++ templates can be thought of as macros, but even in a strongly typed language such as C++, template type errors are only discovered when the templates are instantiated.[1] In some cases, when large, vendor-supplied template libraries are involved, this can be much too late.

Macro systems for languages less sophisticated than ML or Scheme can be much sim-

[1] In fact, one of the major reasons to use C++'s templates is to get the kind of genericity that the type system would otherwise not be capable of expressing.

pler. For example, in assembly language one does not need to worry about references to free variables because there is no block structure. To be "hygienic" in such a setting, it suffices to have a separate namespace for macro-generated temporaries so they cannot be confused with names provided by the programmer.

I will now investigate three of the areas that are still problematic for hygienic Scheme-style macro systems. First, I will have a look at how macros interact with ML-style modules and long identifiers. Then I will discuss the problem of separate compilation and linking. Finally, I come back to the problem of compilation management and automatic dependency analysis.

6.1 A review of hygienic macro expansion

Over the last few years some of the technical problems with macro systems have been solved. Kohlbecker's solution for *hygienic* macro expansion [Koh86], which has been improved several times [BR88, Han91, CR91], provides the mechanism for implementing referentially transparent syntactic extensions in block-structured languages. Today most implementations of Scheme [IEE90] support hygienic macros in one or the other form [Ce91, Dyb92].

The hygienic macro expander of Clinger and Rees [CR91] combines two techniques: renaming and syntactic environments.

Renaming: Renaming solves the type of problem that we encountered in the case of the `SWAP` macro:

```
# define SWAP(x,y) { double temp = x; x = y; y = temp; }
```

The macro introduces the name `temp` into its output. Its purpose is to play the role of a temporary that is not to be confused with any other variable, in particular not with an

argument that also happens to be named `temp`. The solution is to create a brand-new name, a "gensym," at macro-expansion time to take its place. The expression `SWAP(temp,x)` would then correctly yield something like:

```
{ double _g_0815 = temp; temp = x; x = _g_0815; }
```

But in order to be able to do that, the macro expander must "know" that the identifier `temp` is introduced as a bound identifier while other names, for example the type name `double`, are not. But in general it is not possible to have this information at the time the macro is expanded. Consider the following variation on `SWAP` that takes the name of the temporary as an argument:

```
# define SWAP3(t,x,y) { double t = x; x = y; y = t; }
```

With this one can define `SWAP2` as:

```
# define SWAP2(x,y) SWAP3(temp,x,y)
```

`SWAP2`, not `SWAP3`, introduces the name `temp`. The fact that it will be used as a bound identifier will not be discovered until `SWAP3` is expanded as well.

Syntactic environments: To be on the safe side, the expansion algorithm will rename every identifier that is introduced by a macro. An invocation of `SWAP(temp,x)` then also replaces `double` with a generated name and expands into:

```
{ _g_0815 _g_0816 = temp; temp = x; x = _g_0816; }
```

The trick is to compile the resulting code in a *syntactic environment* where `_g_0815` is bound to the original meaning of `double`.

What is the "original meaning"? To handle free occurrences of identifiers such as `double` correctly, one wants them to refer to what they meant at the time when `SWAP` was defined, not the time when it was used:

```
char temp;
# define SWAP(x,y) { double temp = x; x = y; y = temp; }
double x, y;
typedef long double;
...
SWAP (x, y);
```

With hygienic macros, this code should work because `SWAP` refers to the *original* type `double` before it was redefined. Therefore, the expansion of the `SWAP` macro will be compiled in an environment where the gensym for `double` refers to the original `double`. The same environment maps the gensym for `temp` to the original `temp`.

The latter effect is a result of the macro expander's inability to distinguish between free and bound occurrences. It does not know yet that it will not need the original meaning of `temp`. But this is not a problem. The variable is subsequently bound locally, and nobody ever refers to the old, global meaning.

Thus, renaming would prevent inadvertent name clashes, but without help from syntactic environments one must be able to distinguish between free and bound occurrences of identifiers. This is difficult. In order to determine the syntactic role (binding or applied occurrence) of a name in the macro's output, it can be necessary to expand other macro instances. These instances themselves might be generated at the time the macro is instantiated, so they did not exist and, therefore, could not have been analyzed at macro definition time. As a result, the algorithm renames too many variables. Syntactic environments are used to hide the effects of such renamings—thereby handling free variables correctly.

6.2 Macros and modules

A long identifier, in its simplest form, is a pair of two simple identifiers. One of them names the module and the other names the member within the module. In Scheme one might write: (*module-id member*) .

Long identifiers are not hygienic

I will now show that hygienic macro systems are not capable of implementing module systems that require the use of long identifiers. First, let me review the properties of hygienic macro systems with respect to how they treat names.

1. Names can be supplied by the definition of the macro or by the macro's list of actual parameters.
2. Names can be introduced into the output as free occurrences, as binding occurrences, or as bound occurrences.
3. Names supplied by the argument list can be compared to names supplied by the macro's definition for the purpose of pattern matching.

To simulate the long identifier `(M x)` using hygienic macros, one must define `M` as a macro. Clearly, the member name `x` is supplied by the macro's argument list. There are four cases:

1. The name `x` will become a free occurrence in `M`'s expansion. According to the rules of hygienic macro expansion, its meaning would be the same as that of the *simple* name `x` with respect to the current context. But that is not how long identifiers should work.
2. The name `x` will become a bound occurrence and refer to a matching binding occurrence. There is no gain unless the binding occurrence manages to bind the name to something that has the desired semantics.
3. The name `x` will become a binding occurrence. Again, this in itself does nothing to solve the problem because it does not specify what the name will be bound to.
4. The name `x` is matched against another `x` in the macro's definition. A popular example for this is the `cond` macro in Scheme, which looks for the keyword `else` in

its argument list. But such a match fails (and is supposed to fail) if the keyword has been redefined. Likewise, the match of `x` in `(M x)` will succeed or fail depending on definitions for the *simple* name `x`. But the simple name `x` should have nothing to do with the one defined in module `M`.

Thus, there is no way of obtaining the intended semantics using the hygienic macro expander. Long identifiers must be added to the language explicitly.

To put it another way, the prefix `M` in `(M x)` establishes a tiny new scope—just for the one occurrence of `x`. Thus, the module prefix binds the names of the module's members, even though these names are not lexically apparent. But bindings that are not lexically apparent are considered non-hygienic. If one looks at it this way, it is no surprise that a mechanism so carefully crafted to be hygienic cannot implement a language feature that is not.

Long identifiers are not completely "hygienic" under the narrow interpretation used by current macro expansion technology. However, this does not imply that one must abandon them as a useful language feature. But it is necessary to carefully state how the two language features interact. Algorithms to implement them must be adapted accordingly.

Qualified names do not need renaming

Existing algorithms for hygienic macro expansion take advantage of the fact that identifiers in block-structured languages must always be interpreted with respect to a "current" syntactic environment. Therefore, they will cease to work properly when there are multiple environments. But that happens when modules (in the style of SML's structures) and long identifiers are added to the language. Unfortunately, the mechanism for macro expansion and the mechanism for the interpretation of modules both use syntactic environments. But they do so in different ways and for different purposes. With modules, not every name will

be looked up in the "current" environment; member names of modules are resolved in the context of the environment that represents the module's definition.

I have already described the use of syntactic environments for hiding the "mistakes" of renaming. The current environment is carefully modified by the macro expander to map back to their original meanings the renamed versions of identifiers that occur free. Unfortunately, `x` in `(M x)` is not looked up in the current environment, and the one representing `M` is not able to undo the effect of renaming.

But in this case `x` never needed to be renamed in the first place. One purpose of renaming is to avoid name clashes for bound variables. If `x` is such a bound variable, then the only code that can access it must have been inserted by the same macro invocation. It is "private" to the macro; placing it inside module `M` gains nothing in terms of who can see it under which name. The other purpose of renaming is to correlate free occurrences of identifiers with the correct environment where they should be looked up. But with `(M x)`, one already knows where `x` should be looked up: in the environment representing module `M`.

6.3 Combining macros and modules

I will now define a small language with modules and long identifiers and devise a hygienic macro system for it. The key idea is that all renamings done to names that later appear as part of a long identifier, prefixed by the name of a module, must be undone before lookup. In implementations based on "tagging," it is easy to undo renaming operations. Fortunately, "tagging" is the renaming scheme already used by existing macro expansion algorithms.

The language

The language offers λ-abstractions and block-structured definition of variables, macros, and modules.

Syntactic entities:

$\mathrm{I} \in \mathrm{Ide}$	identifiers
$\mathrm{Q} \in \mathrm{QIde}$	qualified (long) identifiers
$\mathrm{T} \in \mathrm{Tf}$	macro transformers
$\mathrm{M} \in \mathrm{Mod}$	modules
$\mathrm{E} \in \mathrm{Exp}$	expressions

Syntax of expressions:

```
Exp ⟶  Q | (E E*) | (lambda (I) E)
     | (let ((I E)) E)
     | (let-syntax ((I T)) E)
     | (let-module ((I M)) E)
```

Syntax of modules:

```
Mod ⟶  ()
     | (let ((I E)) M)
     | (let-syntax ((I T)) M)
     | (let-module ((I M)) M)
```

Domain equations:

i,j ∈ *Ide*		identifiers
q ∈ *QIde*	= $Ide + QIde \times QIde$	
		qualified identifiers
t ∈ *Tag*		tags
e ∈ *Env*		environments
d ∈ *Den*	= $Var + Spec + Mac + Mod + U$	
U	= {*unbound*}	error denotation
v ∈ *Var*	= *QIde*	variables
s ∈ *Spec*	= {*lambda, let, let-syntax, let-module* }	
		keywords
Mac	= $R \times Ide* \times Env$	
		macros
r ∈ *R*		macro transform
Mod	= *Env*	modules

Long identifiers are treated as if they were lexical units. This is not strictly necessary, but it makes it somewhat easier to present the algorithm. However, macro transformers can assemble long identifiers from other (simple or long) identifiers. The notation **m.x** shall denote a name **x** qualified by module name **m**.

I have not specified a syntax for macro transformers, but I require that it must be possible to implement the macro compiler *compile*. The discussion applies equally well to both high-level macro languages that are based on pattern matching and to low-level systems. Examples are shown using **syntax-rules** [Ce91].

Macro calls, which take the the form `(Q ...)`, can be used wherever Exp or Mod is required. But they must ultimately expand into another Exp- or Mod-form, respectively.

The algorithm

The algorithm presented here follows ideas in Rees [Ree93] by using a domain *Ide* that consists of symbols and tagged names. Only plain symbols can appear in the original program text. Tagged names are introduced by the macro expander. Tagging is simply a way of uniformly renaming all identifiers inserted by the same instance of a macro.

The environment constructors *ident* and *bind* are taken directly from Clinger and Rees' paper [CR91]. That paper avoids any details of how identifiers are renamed. Consequently, the authors do not mention any tags. Rees [Ree93] later described the tagging approach separately. In my presentation, I make tagging explicit by introducing *tag* and *fork*. In addition, *qualify* and *empty* were added for the module system.

Every macro expansion produces a new expression and the corresponding environment for expansion, which is a combination of the macro's environment of definition and the current environment. Identifiers that are inserted by the macro (as opposed to being supplied as a macro parameter) will be tagged; the tag matches the corresponding *fork* in the environment. Thus, lookup—now with the tag stripped from the name—proceeds in the macro's environment of definition. If the identifier comes from the argument list, then it is not tagged. Therefore, the *fork* node is skipped and the name will be resolved with respect to the current environment.

If a macro is defined inside a module and refers to something in that module by an unqualified name, then upon expansion of the macro outside the module that name must be qualified. In the following example the program on the left should expand into that on the right:

```
(let-module
  ((mod (let ((x 1))
      (let-syntax                          (let-module
          ((mac (syntax-rules ()    ⟹       ((mod (let ((x 1)) ())))
              ((_) x))))                     mod.x)
        ())))))
  (mod.mac))
```

The transcription of `(mod.mac)` is an unqualified, but tagged, version of `x`. Tagging correctly relates it to the environment where `mac` was defined, which was in module `mod`. The algorithm uses *qualify* constructors to annotate such environments with the name of the module they belong to. Everytime *lookup* encounters an environment constructed with *qualify*, it must coerce the denotation obtained from the recursive invocation of lookup accordingly. In the example it would prefix `x` with the module name `mod`.

Consider the following code:

```
(let-syntax
    ((mac (syntax-rules () ((_ m) m.x))))
  (let-module
      ((mod (let ((x 1)) ())))
    (mac mod)))
```

The expansion of `(mac mod)` will rename `x` into *tag* $(t, \mathtt{x})$ for some tag t. The *renamed* version of `x` will be looked up in the environment representing `mod`, where it will not be found because that environment lacks the matching *fork* node. Following the earlier discussion one must strip all tags from the renamed `x`, which will solve the problem.

Identifier and environment constructors:

$$\begin{aligned} \textit{tag} &\in Tag \times Ide \to Ide \\ \textit{empty}, \textit{ident} &\in Env \\ \textit{bind} &\in Env \times Ide \times Den \to Env \\ \textit{fork} &\in Tag \times Env \times Env \to Env \end{aligned}$$

$$\mathit{qualify} \in \mathit{Env} \times \mathit{Var} \to \mathit{Env}$$

Function signatures:

$$\begin{aligned}
\mathit{compile} &\in \text{Tf} \times \mathit{Env} \to R \times \mathit{Ide}* \\
\mathit{coerce} &\in \mathit{Den} \times \mathit{Var} \to \mathit{Den} \\
\mathit{lookup} &\in \mathit{Env} \times \mathit{Ide} \to \mathit{Den} \\
\mathit{qlookup} &\in \mathit{Env} \times \mathit{QIde} \to \mathit{Den} \\
\mathit{transcribe}_E &\in \text{Exp} \times R \times \mathit{Ide}* \times \mathit{Env} \to \text{Exp} \\
\mathit{transcribe}_M &\in \text{Mod} \times R \times \mathit{Ide}* \times \mathit{Env} \to \text{Mod}
\end{aligned}$$

Equations for *lookup***:**

$$\begin{aligned}
\mathit{lookup}\,(\mathit{ident}, i) &= i \\
\mathit{lookup}\,(\mathit{empty}, i) &= \mathit{unbound} \\
\mathit{lookup}\,(\mathit{bind}\,(e, i', d), i) &= (i = i') \to d, \mathit{lookup}\,(e, i) \\
\mathit{lookup}\,(\mathit{fork}\,(r, e_d, e_u), i) &= (i = \mathit{tag}\,(t, i')) \to \mathit{lookup}\,(e_d, i'), \mathit{lookup}\,(e_u, i) \\
\mathit{lookup}\,(\mathit{qualify}\,(e, v), i) &= \mathit{coerce}\,(\mathit{lookup}\,(e, i), v)
\end{aligned}$$

Equations for *coerce***:**

$$\begin{aligned}
\mathit{coerce}\,(v, v') &= (v', v) \\
\mathit{coerce}\,(s, v) &= s \\
\mathit{coerce}\,((r, i*, e), v) &= (r, i*, \mathit{qualify}\,(e, v)) \\
\mathit{coerce}\,(e, v) &= \mathit{qualify}\,(e, v)
\end{aligned}$$

$$\mathit{coerce}\,(\mathit{unbound}, v) \;=\; \mathit{unbound}$$

Equations for *qlookup***:**

$$\begin{aligned}
\mathit{qlookup}\,(e, i) &= \mathit{lookup}\,(e, i)\\
\mathit{qlookup}\,(e, (q_1, (q_2, q_3))) &= \mathit{qlookup}\,(e, ((q_1, q_2), q_3))\\
\mathit{qlookup}\,(e, (q, \mathit{tag}\,(t, i))) &= \mathit{qlookup}\,(e, (q, i))\\
\mathit{qlookup}\,(e, (q, i)) &= \mathit{lookup}\,(\mathit{qlookup}\,(e, q), i); \quad i \neq \mathit{tag}\,(t, i')
\end{aligned}$$

Rewrite rules for expressions: The complete algorithm for hygienic macro expansion is given as a set of rewrite rules defining the "expands to" relation $\xrightarrow{E}$. If $\mathbf{E}$ expands to $\mathbf{E}'$ in e, then I write $e \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}'$. Function *compile* implements the macro language. It returns the list of identifiers captured by the macro together with a transformer program r, which then can be "executed" by either $\mathit{transcribe}_E$ or $\mathit{transcribe}_M$. For simplicity, I have assumed that macros take exactly one argument. The argument can be any symbolic expression.

$$\frac{\mathit{qlookup}\,(e, \mathbf{q}) = \mathbf{v}}{e \vdash \mathbf{q} \xrightarrow{E} \mathbf{v}} \qquad [\mathit{var}_R]$$

$$\frac{\begin{array}{c} e \vdash \mathbf{E}_0 \xrightarrow{E} \mathbf{E}_0' \\ \forall i \in \{1, \ldots, k\}, k \geq 0 : e \vdash \mathbf{E}_i \xrightarrow{E} \mathbf{E}_i' \end{array}}{e \vdash \texttt{(}\mathbf{E}_0\ \ \mathbf{E}_1\ \ \ldots\ \ \mathbf{E}_k\texttt{)} \xrightarrow{E} \texttt{(}\mathbf{E}_0'\ \ \mathbf{E}_1'\ \ \ldots\ \ \mathbf{E}_k'\texttt{)}} \qquad [\mathit{app}_E]$$

$$\frac{\begin{array}{c} \mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{lambda} \\ \mathit{bind}\,(e, \mathbf{x}, \mathbf{x}') \vdash E \xrightarrow{E} E' \\ \mathbf{x}' \text{ is a fresh identifier} \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \ \texttt{(}\mathbf{x}\texttt{)}\ \ \mathbf{E}\texttt{)} \xrightarrow{E} \texttt{(lambda (}\mathbf{x}'\texttt{)}\ \ \mathbf{E}'\texttt{)}} \qquad [\lambda_E]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}) = (r, \langle i, \ldots\rangle, e_d) \\ t \text{ is a fresh tag} \\ \mathit{transcribe}_E(\texttt{(}\mathbf{k}\ \ \mathbf{B}\texttt{)}, r, \langle \mathit{tag}\,(t, i), \ldots\rangle, \mathit{fork}\,(t, e_d, e)) = \mathbf{E} \\ \mathit{fork}\,(t, e_d, e) \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}'\end{array}}{e \vdash \texttt{(}\mathbf{k}\ \ \mathbf{B}\texttt{)} \xrightarrow{E} \mathbf{E}'} \quad [\mathit{mac}_E]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{let} \\ e \vdash \mathbf{E}_1 \xrightarrow{E} \mathbf{E}_1' \\ \mathbf{x}' \text{ is a fresh identifier} \\ \mathit{bind}\,(e, \mathbf{x}, \mathbf{x}') \vdash \mathbf{E}_2 \xrightarrow{E} \mathbf{E}_2'\end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \ \texttt{((}\mathbf{x}\ \ \mathbf{E}_1\texttt{))}\ \ \mathbf{E}_2\texttt{)} \xrightarrow{E} \texttt{(let ((}\mathbf{x}'\ \ \mathbf{E}_1'\texttt{))}\ \ \mathbf{E}_2'\texttt{)}} \quad [\mathit{let}_E]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{let\text{-}syntax} \\ \mathit{compile}\,(\mathbf{T}, e) = (r, i*) \\ \mathit{bind}\,(e, \mathbf{x}, (r, i*, e)) \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}'\end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \ \texttt{((}\mathbf{x}\ \ \mathbf{T}\texttt{))}\ \ \mathbf{E}\texttt{)} \xrightarrow{E} \mathbf{E}'} \quad [\mathit{syn}_E]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{let\text{-}module} \\ e \vdash (\mathbf{M}, \mathit{empty}) \xrightarrow{M} (\mathbf{M}', e_m) \\ \mathbf{x}' \text{ is a fresh identifier} \\ \mathit{bind}\,(e, \mathbf{x}, \mathit{qualify}\,(e_m, \mathbf{x}')) \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}'\end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \ \texttt{((}\mathbf{x}\ \ \mathbf{M}\texttt{))}\ \ \mathbf{E}\texttt{)} \xrightarrow{E} \texttt{(let-module ((}\mathbf{x}'\ \ \mathbf{M}'\texttt{))}\ \ \mathbf{E}'\texttt{)}} \quad [\mathit{mod}_E]$$

Rewrite rules for modules: Rule [mod_E] refers to a second "expands to" relation. The notation $e \vdash (\mathbf{M}, e_m) \xrightarrow{M} (\mathbf{M}', e_m')$ not only describes what a module $\mathbf{M}$ expands into, it also calculates an environment representing the definitions contained in the module. Not surprisingly, the set of rules for relation $\xrightarrow{M}$ is similar to the one defining $\xrightarrow{E}$.

$$e \vdash (\texttt{()}, e_m) \xrightarrow{M} (\texttt{()}, e_m) \quad [\mathit{nul}_M]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}) = (r, \langle i, \ldots\rangle, e_d) \\ t \text{ is a fresh tag} \\ \mathit{transcribe}_M(\texttt{(}\mathbf{k}\ \ \mathbf{B}\texttt{)}, r, \langle \mathit{tag}\,(t, i), \ldots\rangle, \mathit{fork}\,(t, e_d, e))) = \mathbf{M} \\ \mathit{fork}\,(t, e_d, e) \vdash (\mathbf{M}, e_m) \xrightarrow{M} (\mathbf{M}', e'_m)\end{array}}{e \vdash (\texttt{(}\mathbf{k}\ \ \mathbf{B}\texttt{)}, e_m) \xrightarrow{M} (\mathbf{M}', e'_m)} \quad [\mathit{mac}_M]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{let} \\ e \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}' \\ \mathbf{x}' \text{ is a fresh identifier} \\ \mathit{bind}\,(e, \mathbf{x}, \mathbf{x}') \vdash (\mathbf{M}, \mathit{bind}\,(e_m, \mathbf{x}, \mathbf{x}')) \xrightarrow{M} (\mathbf{M}', e'_m)\end{array}}{e \vdash (\texttt{(}\mathbf{k}_0\ \ \texttt{((}\mathbf{x}\ \ \mathbf{E}\texttt{))}\ \ \mathbf{M}\texttt{)}, e_m) \xrightarrow{M} (\texttt{(let ((}\mathbf{x}'\ \ \mathbf{E}'\texttt{))}\ \ \mathbf{M}'\texttt{)}, e'_m)} \quad [\mathit{let}_M]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{let\text{-}syntax} \\ \mathit{compile}\,(\mathbf{T}, e) = (r, i*) \\ \mathit{bind}\,(e, \mathbf{x}, (r, i*, e)) \vdash (\mathbf{M}, \mathit{bind}\,(e_m, \mathbf{x}, (r, i*, e))) \xrightarrow{M} (\mathbf{M}', e'_m)\end{array}}{e \vdash (\texttt{(}\mathbf{k}_0\ \ \texttt{((}\mathbf{x}\ \ \mathbf{T}\texttt{))}\ \ \mathbf{M}\texttt{)}, e_m) \xrightarrow{E} (\mathbf{M}', e'_m)} \quad [\mathit{syn}_M]$$

$$\frac{\begin{array}{c}\mathit{qlookup}\,(e, \mathbf{k}_0) = \mathit{let\text{-}module} \\ e \vdash (\mathbf{M}_1, \mathit{empty}) \xrightarrow{M} (\mathbf{M}'_1, e_{m,1}) \\ \mathbf{x}' \text{ is a fresh identifier} \\ \mathit{qualify}\,(e_{m,1}, \mathbf{x}') = e^q_{m,1} \\ \mathit{bind}\,(e, \mathbf{x}, e^q_{m,1}) \vdash (\mathbf{M}_2, \mathit{bind}\,(e_m, \mathbf{x}, e^q_{m,1})) \xrightarrow{M} (\mathbf{M}'_2, e'_m)\end{array}}{e \vdash (\texttt{(}\mathbf{k}_0\ \ \texttt{((}\mathbf{x}\ \ \mathbf{M}_1\texttt{))}\ \ \mathbf{M}_2\texttt{)}, e_m) \xrightarrow{M} (\texttt{(let-module ((}\mathbf{x}'\ \ \mathbf{M}'_1\texttt{))}\ \ \mathbf{M}'_2\texttt{)}, e'_m)} \quad [\mathit{mod}_M]$$

Summary

Modules and qualified names cannot be "simulated" using hygienic macros because the rules governing long identifiers are not hygienic. Renaming is necessary for ordinary symbols to avoid name clashes and to correctly relate free occurrences to their original meanings. But names that are qualified by module identifiers are neither free nor can they clash with other names. Thus, it would normally not be necessary for them to undergo the same

routine of systematic rewriting.

But I want to combine macros and modules because this way, in an untyped setting, one can simulate generic modules. In the system described, macros are permitted to assemble long identifiers from components during expansion. Therefore, by the time one finds a long identifier `(M x)`, the compiler may have renamed `x` in the course of earlier macro expansions. In such cases it is necessary to undo such renamings before attempting to find a definition for `x` in module `M`.

Tagging is a technique for efficiently renaming identifiers and at the same time duplicating their bindings in the associated environment. Fortunately, the same tagging mechanism can be used to implement the lookup operation for long identifiers.

6.4 Separate compilation and linking

Another problem, orthogonal to modules and long identifiers, arises if one wishes to provide separate compilation. In order to preserve hygiene, the macro expander chooses fresh names for bound variables throughout the program. Furthermore, upon macro invocation it renames identifiers captured by that macro. This applies to both local and "external" bindings. Actual names chosen by the macro expander are arbitrary and will depend on its internal state. This is unfortunate because externally bound names are the basis for program linking.

In the model used here, a program is nothing more than a big unnamed module. This simply means that it consists of a number of nested definitions. Therefore, I will use Mod as the syntax for programs. A program can be broken into an ordered sequence of smaller programs—compilation units.

Let me first introduce the *link* operation on fully macro-expanded text. This operation puts two programs back together into one. Since each program must have an innermost

empty module `()`, one can say that $\mathit{link}\,(\mathbf{P}_0, \mathbf{P}_1)$ substitutes $\mathbf{P}_1$ for this inner `()` in $\mathbf{P}_0$. With the help of *link* one can then describe the process of linking in terms of extending relation $\xrightarrow{M}$ to sequences of programs.

$$e \vdash \langle\rangle \xrightarrow{L} (\,\texttt{()}\,, e) \qquad [\mathit{nil}]$$

$$\frac{\begin{array}{c} e \vdash (\mathbf{P}_0, e) \xrightarrow{M} (\mathbf{P}_0', e') \\ e' \vdash \langle \mathbf{P}, \ldots \rangle \xrightarrow{L} (\mathbf{P}', e'') \\ \mathit{link}\,(\mathbf{P}_0', \mathbf{P}') = \mathbf{P}'' \end{array}}{e \vdash \langle \mathbf{P}_0, \mathbf{P}, \ldots \rangle \xrightarrow{L} (\mathbf{P}'', e'')} \qquad [\mathit{link}]$$

Of course, this does not give us separate compilation. A unit can only be expanded and compiled once all other units on its left in the sequence have already been expanded as well.

This is the same situation that we have in SML. Therefore, I follow the model described by Appel and MacQueen [AM94], just as I have before in the context of compilation management (→ Chapter 3) and λ-splitting (→ Chapter 5).

Compiling a module produces two separate results: an environment describing the module's exports and executable code for the module. While relation $\xrightarrow{M}$ only describes the elaboration of the environment, it typically is the code generation part of compilation where most of time and resources are spent. Moreover, subsequent compilation of other modules only depend on the environment but not on the code.

A modification that only results in different code but not in a different export environment does not require other modules to be recompiled as well. I discussed this idea when I introduced the notion of cutoff recompilation.

However, in the presence of hygienic macros it can happen that the export environment refers to names that the macro expander introduced by way of renaming other identifiers. Such names are annotated with tags, but tags are chosen fresh by the macro expander every time; they are not persistent. Thus, recompilation can introduce spurious differences into the export environments. Such differences would normally cause all depending modules to be recompiled as well.

I will now develop a persistent naming scheme for export environments that is compatible with hygienic macro expansion. Persistent names will be independent of arbitrary choices by the macro expander. Therefore, they avoid spurious incompatibilities in export environments.

The objective of compilation is to produce some form of directly executable code. In the model used here, one first translates each compilation unit into a closed λ-expression, as does the SML/NJ compiler for each ML compilation unit. All free variables of the original module appear as explicit arguments. The return value is a data structure containing the values for the unit's variable definitions. Since the expression has no free variables, it can be compiled without depending on context information.

It is the linker's responsibility to fetch correct values for the arguments of the expression from the "global" environment and to later augment it with new bindings. To be able to do so, it must be provided with information about imports and exports.

Augmenting the global environment

Every compilation unit must be compiled with respect to a current global environment. Global environments belong to a domain that is different from *Env* because I wish to distinguish between ordinary denotations and "global" denotations. For example, variable bindings in the global environment are no longer represented by qualified names. Instead,

they directly associate identifiers with values.

Nevertheless, the overall structure of the global environment is very similar to *Env*. Therefore, given a persistent encoding for names there is a relatively straightforward way of producing export information from a compilation unit:

1. Expand unit **U** in the current global environment: $e_g \vdash (\mathbf{U}, \mathit{empty}) \xrightarrow{M} (\mathbf{U}', e_u)$. Note that e_u does not contain *fork* nodes.
2. Traverse e_u and assign numbers to bindings of variables. This has to be done recursively so that variables in modules are numbered as well.
3. Construct an expression that at runtime will evaluate into a vector v of the so-numbered values. Value i must be stored at $v[i]$. The expression itself replaces the innermost `()` of the compilation unit, thus turning a module into an expression.
4. Convert e_u to a Δ-environment e_u^{Δ}. Δ-environments encode incremental changes to the global environment. They only use persistent data. Cycles and sharing are encoded using explicitly marked edges. Δ-environments are suitable for storage on stable media, for example files in the ambient file system.
5. When **U**'s code has successfully executed, one can instantiate e_u^{Δ} using the current global environment and the vector of export values obtained from running the code.

The following problems must be addressed when creating e_u^{Δ} from e_u:

- Back- and cross-edges, which represent cycles and sharing in e_u, are detected and marked by explicit labels. (In the case of the system that I have described, there can never be cycles. Introducing recursive definitions would change this situation.) Labels are small integers, consecutively chosen beginnig with $0, 1, \ldots$
- Variable bindings are represented by numbers referring to slots of the export vector.

- There are names in the environment structure that have been imported from the global environment while **U** was expanded. In order to be able to re-establish this situation when instantiating e_u^Δ, these names are encoded using *persistent paths*. Persistent paths are also used as part of the import mechanism. I will describe them below.
- All remaining names have been created by tagging other existing identifiers. Tags are encoded by systematically mapping them to small integers.

The conversion can be implemented by a conventional recursive traversal of the data structure that implements e_u. For the language described here, the data structure is a DAG. In general, with recursion, it can be an arbitrary directed graph.

The inverse process of instantiating a Δ-environment with respect to some vector of values and a current global environment is then straightforward.

- Cycles and sharing are re-created according to the labels found in e_u^Δ.
- Variables are bound to values. Those values are fetched from the result vector using the slot numbers that are stored in the Δ-environment.
- For each integer in e_u^Δ that stands for some tag a fresh tag is created.
- Persistent paths are instantiated in the current global environment. This operation is the same that is also used for the import mechanism.
- All other names are created by tagging existing ones according to the integer-to-tag map mentioned above.

Imports

Hygienic macros complicate the problem of specifying how an import value is to be located because identifiers can be names that were generated by the macro system. Furthermore, since environments contain fork points, it may happen that a name must be looked up in

some macro's environment of definition and not in the current global environment. To deal with this, one must be able to specify generated names without mentioning actual tags and to redirect the lookup operation to other environments.

If a name is to be looked up somewhere else and not in the current global environment, then it must have been inserted by a global macro. Therefore, it is necessary to find the macro's definition because that is where one will find the environment required for resolving the lookup. Of course, finding the macro's definition is just another instance of the import problem.

Fortunately, this recursive process is always guaranteed to terminate. Otherwise, there would have been infinitely many expansion steps in the original source program. Therefore, one can refer to any global environment using a sequence of macro names. The macro names have to be looked up in order, starting with the global environment. This sequence is called a *path*, and the global environment itself is represented by the empty path. Given a persistent naming mechanism, one can specify imports using a path for the environment and a persistent name for the identifier to be looked up. The path itself also consists of persistent names.

A persistent naming mechanism is a method for specifying identifiers without mentioning actual tags. Three kinds of names must be considered:

1. Plain symbols that are not tagged at all.
2. Names tagged while expanding **U**.
3. Names that had been tagged before **U** was expanded.

Plain names are easy to deal with. They can represent themselves. The second kind of name can never be subject to a global lookup operation. Tags created while expanding **U** do not occur in the global environment. Every global lookup operation of such an identifier would be certain to fail. Fortunately, this cannot happen. Names tagged in such a way will

always be interpreted in an environment that contains the corresponding *fork* nodes, and those cancel new tags before the lookup operation "reaches" the external environment.

Thus, only the third kind of name is of interest. How can those names find their way into compilation unit **U**? Without non-hygienic extensions to the macro system this is only possible by expanding a global macro. Fortunately, the names that a macro can insert are known because function *compile* computes this information from a transformer. Therefore, it is possible to "name" such an identifier by specifying the macro that inserted it and the position number in the macro's list of captured names. Naming the macro is yet another instance of the import problem. I have already discussed it above.

The module system raises a few additional, albeit minor, issues. First of all, a global lookup can take place with respect to a module's environment. Also, the *qlookup* function might eliminate tags from names before they are fed to the lookup procedure. Therefore, the persistent naming scheme must offer a way of expressing this. Finally, if low-level, non-hygienic macros are permitted, then one must also have a way of specifying identifiers that have been created non-hygienically.

When put together, this results in mutually recursive definitions for imports, paths, and persistent paths called *recipes*. A recipe can either be a symbol, an import specifying some macro together with a position number addressing one of the names captured by that macro, or a recipe plus an "untag" directive, which indicates that the last tag is to be removed.

Rec	$= Symbol + Mc + Untag$	recipes
Mc	$= Imp \times Integer$	name capture
Untag	= *Rec*	untag
Path	$= Rec*$	paths
Imp	$= Path \times Rec$	imports

Recipes and imports are prescriptions for re-enacting some of the same operations that the macro expander would perform if one would let it expand the original source of **U**

again. However, these operations are "distilled" to focus efforts on just the names that are important for external linkage.

Compiling one unit

Global environments are similar, but not identical, to ordinary environments. Therefore, global environments belong to a now domain *GEnv*. The rules for macro expansion that I presented earlier cannot be used directly with elements of *GEnv*. Therefore, I introduce *extern* as a constructor that takes elements of *GEnv* to elements of *Env*.

$$\mathit{extern} \in \mathit{GEnv} \times \mathit{Path} \rightarrow \mathit{Env}$$

Aside from the technical issue of dealing with two different domains, this also has a practical advantage by providing a way of detecting free variables in the compilation unit. It is convenient to annotate *extern* nodes with the path that is the persistent name for the global environment in question. The coercion function *gcoerce*, which is analogous to *coerce* in the case of module environments, takes global denotations (*GDen*) to elements of *Den*.

$$\mathit{gcoerce} \in \mathit{GDen} \times \mathit{Path} \times \mathit{Rec} \rightarrow \mathit{Den}$$

A name is a free name if lookup for that name reaches a *extern* node. Therefore, the task to detect free variables can naturally be carried out by *gcoerce*. For each free variable it calculates the import specification, invents a new identifier, and remembers the relationship between the two. The fresh name is returned from *lookup* in the form of a *Var* denotation. In the end one can close over the fully expanded program by wrapping it in λ-abstractions. The list of imports specifies the values that the linker must fetch from the global environ-

ment. These values will be passed as arguments to the executable code.

How does one calculate recipes? This can be done "on the fly," during macro expansion. Domain *Ide* is treated as an abstract datatype. Internally, identifiers consist of two parts. One part is the actual name and the other one is the current recipe. Every time the macro expander constructs a new identifier, it also remembers how this was done. To make that work, one must adjust the definition of recipes, so they can express the fact that a name carries a newly generated tag. Such new tags will ultimately go away before the name becomes subject to global lookup. But in the meantime it is necessary to keep track of them. All operations that create names must also compute the corresponding recipe. Thus, one pays a constant penalty per such operation. Still, the asymptotic complexity of the algorithm stays the same.

Experiments with a prototype implementation confirm that the algorithms presented here perform well in practice. I expect to be able to combine my solution with existing compilers for Scheme and Scheme-like languages.

6.5 Compilation management

In chapter 4 I have shown how features of a programming language that seem harmless in comparison with the power of a macro system can have severe detrimental effects on the feasibility of automatic analyses such as finding the dependencies between compilation units.

The possibility of structures being "opened," for example, turns an initially rather straightforward task into a challenging problem. If unrestricted **open** is permitted at top level, then it even becomes NP-complete. But all the difficulties stem from the fact that opening a structure confuses the knowledge of which identifiers are bound, and what they are bound to. One must already know the definition of the structure that was being opened

to be able to understand the resulting scopes.

The same can be said about macros. The whole purpose of macros is to provide new ways of binding variables. When confronted with the instance of an unknown macro, one knows *nothing* about the resulting scope rules. It will be no surprise to find that this confuses the dependency analyzer.

In fact, one frequently stated criticism is that macros make it all-too-easy to obfuscate the language. A few well-designed macros, written by expert language designers, might indeed simplify certain idioms and make programs easier to read. But such power should be used sparingly, though the systems in existence seem to encourage abuse.

Complexity of dependency analysis

I demonstrated that dependency analysis is not tractable if different sources can provide definitions for the same identifier. This effect is independent of macros or **open**. Therefore, one should certainly adopt restriction 1 from page 68 here as well:

> *In each group there can be at most one source that provides a top-level definition for any given symbol. A source can define a name that was already defined by the context, but uses of that name in any of the sources will then refer to the new definition.*

Let the context environment provide the definitions for `flag`, `a`, and `a*` that are shown in figure 6.1. Two compilation units, S and S', invoke `a` and `a*`, respectively. S contains `(a flag flag c1)`, and the code in S' is `(a* flag flag c2 c3)`. Either macro is sensitive to how `flag` is currently bound. If the original definition is in place and has not been redefined since, then the first pattern matches. Otherwise the first pattern will not match, and the macro expands into an empty module with no definitions. Therefore, compiling S before S' will result (among others) in a definition for `c1`, while `c2` and `c3`

```
(let ((flag 0))
  (let-syntax ((a (syntax-rules (flag)
      ((_ flag f c1)
       (let ((f 1))
         (let ((c1 0))
           ())))
      ((_ _ _ _) ()))))
    (let-syntax ((a* (syntax-rules (flag)
        ((_ flag f c2 c3)
         (let ((f 1))
           (let ((c2 0))
             (let ((c3 0))
               ()))))
        ((_ _ _ _ _) ()))))
      ())))
```

Figure 6.1: **Macros make dependency analysis difficult.** Even hygienic macros are problematic for dependecy analyzers, because like **open** in SML they can be used to modify the scope of other variables.

will be defined if the order is reversed.

This construction can be extended into an NP-hardness proof by reduction from SAT in a fashion analogous to the proof for claim 3 in chapter 3. Apparently, hygiene did not help with dependency analysis.

For efficient and reliable dependency analysis, it would be necessary to significantly reduce the expressiveness of the macro system. Restrictions similar to those used by CM for SML will not suffice. The analyzer would still have to expand macros as it goes. Scheme's hygienic macro system is simply too powerful. The macro language is Turing-complete, expansion is not even guaranteed to terminate, and termination is not decidable [Göd31, Tur37].

6.6 Macros in large-scale programming

Macro systems are a popular feature of many programming languages. But they have a number problems in the following areas:

1. Avoiding inadvertent capture by bound variables
2. Correct treatment of free variables
3. Relationship between long identifiers and renaming
4. Separate compilation, linking, and naming of imports
5. Efficient dependency analysis
6. Type systems
7. Termination

The first two were solved by existing hygienic macro systems, but these systems have yet to find wide-spread use in languages other than Scheme. I have discussed possible solutions to items 3 and 4. The last three problem areas, however, must certainly be regarded as serious obstacles to reliable, large-scale programming.

Chapter 7

Conclusions

I have investigated a variety of issues that arise with separate compilation. A major portion of this work is concerned with compilation management and cross-module optimization, both in the framework of the ML programming language.

The group model employed by SML/NJ's compilation manager enables modular large-scale programming. Traditionally, this has been difficult because modularity is often jeopardized by restrictions on definability of names or availability of definitions when there is only one global namespace for program linking. This contrasts with CM's approach where groups of sources are arranged into a hierarchical structure. Export environments are combined only when necessary, thereby avoiding most name clashes. The few remaining clashes can always be solved by local modifications because they never affect unrelated components of a program.

CM can be seen as extending the language ML, augmenting it with hierarchical coarse-grain modularity where separately compiled source files are the basic building blocks. But this extension is rather modest. The group model is intuitive, CM's configuration language is surprisingly small and simple, and an automatic analysis frees the programmer from the tedious task of having to keep track of intermodule (but intragroup) dependencies.

The framework for an automatic cross-module optimization technique based on λ-calculus is integrated with CM. λ-calculus is a powerful language for expressing programs and program transformation. Inlining is closely related to function application, β-reduction, and substitution. But this is where λ-calculus is at its best. Therefore, it is no surprise that cross-module inlining will benefit from being cast in this framework.

Compilation units can be viewed as functions. The process of linking applies them to imports, thus obtaining exports. By splitting the functions into expansive and inlinable parts, we are able to regroup the code in such a way that cross-module inlining turns into the task of performing ordinary intramodular optimizations. The functions representing compilation units are split in such a way that algebraic function composition (the B combinator) can be used to recombine the parts.

As an alternative to automatic cross-module optimizations, I also investigated macro systems. Macros are a language feature that is still popular with many programmers. The development of hygienic macro expansion algorithms for Scheme have solved many of the traditional problems with macro systems. The work presented here identified and solved two problems that arose when I tried to unify hygienic macros with an ML-style module system, with the presence of long identifiers, and with separate compilation. But I also showed that other serious problems are still open.

7.1 Other languages

How strongly did my work depend on Standard ML, and how viable are the approaches if one tries to apply them elsewhere? This question should be asked in two parts. First, is it possible to implement hierarchical modularity, automatic dependency analysis, and cross-module optimizations for other programming languages? And second, if so, can the solution be made as elegant as CM?

Certainly, the answer to the first part must be "yes." I have demonstrated in chapter 3 how hierarchical modularity—the group model—can be implemented in terms of simple operations on environments: layering, filtering, and renaming. It is not a great challenge to implement the same operations for other systems, for example for the symbol tables in UNIX object files. Efficient calculation of dependencies for Standard ML proved to be unusually difficult. For many languages this will in fact be easier, as discussed in section 1.3.

Cross-module optimizations can be (and have been) implemented for languages other than ML. With my work I was able to leverage the power of λ-calculus for this purpose. With compilers that use different intermediate formats, one will not have this luxury. An intermediate format that is not capable of expressing higher-order functions and nested function definitions will make it difficult if not impossible to use my technique.

The answer to the second part of the question, the question of whether a solution would be as elegant as CM, is much less clear. I do not have a measure for elegance, and even if I did—I could only speculate. But I can say that ML, and SML/NJ in particular, made it especially pleasant and rewarding to create a compilation manager for it.

In part this is due to the fact that ML is a good implementation language for compilers and compilation management tools. Compilation management, like compilers, benefits from most of the language's strengths and suffers from few of the weaknesses [App97, ch. 1]. SML/NJ provides the "visible compiler" interface, which made it easy to implement type-safe linking as well as cutoff recompilation.

Even more importantly, ML is a good target for a compilation management tool. It is a very elegant language with a module system widely regarded as being one of the most sophisticated and expressive in existence. CM did not have to add much to support hierarchical modularity. That made it possible to keep the configuration language simple. ML itself is based on λ-calculus, and compilers take advantage of that. The development of λ-splitting was only possible because the SML/NJ compiler uses various λ-calculi as its

intermediate representations for programs.

Aside from the many positive aspects of ML, there have been some that, as described below, proved difficult to deal with. These cases have been rare, but, nevertheless, language designers should take note. Some of the difficult problems addressed by my work would not have come up had I chosen a different programming language.

The simplicity of CM's model benefited from being built upon and integrated with one—and only one—very expressive and elegant programming language. One should expect similar tools for similar languages to be comparably elegant. General-purpose compilation managers cannot define themselves as extensions of only one language. They may be more versatile, but they are also more complicated because for different languages many language-specific details must be taken care of differently. Without a fixed base language, they also need a significantly richer notation for expressing dependencies.

7.2 Lessons for language design

Chapter 4 was concerned with the design of an efficient dependency analysis algorithm for ML. One language feature—the potential of having more than one top-level definition for the same identifier—renders the general analysis problem NP-complete. But I was not interested in completely eliminating this feature because a requirement for global uniqueness of names causes problems with modularity.

CM's approach avoids these difficulties by eliminating multiple definitions locally, within individual groups, while globally there is no such restriction. To incorporate this rule into ML's definition it would be necessary to explain the notion of compilation units and the idea of groups as part of the document. The original definition and commentary [MTH90, MT91] only briefly discussed separate compilation of closed functors; the revised definition [MTHM97] has dropped every mention of it.

I like to view CM as an extension to ML; the result is a hybrid that is composed of a powerful programming language and a minimal configuration language. Some decisions in the design of one part can only be justified or even described in terms of the other part. I have tried to keep CM's impact on the ML language as small as possible, but I was not able to make it zero.

The **open** construct of ML and equivalent features in other languages have been criticized before, and I have done so again. It introduces bindings for names that are not lexically apparent, which is its most troubling aspect. The unrestricted use of **open** would also lead to an NP-complete dependency analysis problem in the particular model that I have chosen. Therefore, it was banned from the top level. My model can accomodate slightly less severe restrictions; different models may require different restrictions.

Programs are not only read by analysis tools; humans read them as well. A language construct like **open** that serves to confuse the analysis tool is also likely to confuse the human reader. It would be much better if **open** had been replaced with the equivalent of Modula-3's `IMPORT`, where the names of the identifiers to be defined must be listed explicitly.

In fact, ML already has the facilities for this style of programming. If one wishes to refer to `A.B.f` simply by the name `f`, then instead of opening `A.B` one can write

```
val f = A.B.f
```

Unfortunately, this does not always work as intended because **val** declarations strip information about the identifier's status. In this example, `A.B.f` can be a constructor for some datatype, but `f` will be just a variable. The new datatype replication facility that was introduced with SML'97 is meant to solve this problem. Unfortunately, like **open**, it does this by introducing definitions that are not lexically apparent.

At the very heart of hygienic macros lies the idea that the names bound by binding

constructs must be lexically apparent. I have argued that this is a desirable property, but it is not sufficient. Consider the following expression, where `bnd` is a macro:

```
(bnd a b (+ a b))
```

One of the many possible definitions for the macro would locally bind `a` to the value of `b` and then evaluate the body expression `(+ a b)`. Another possible definition binds `b` to the value of `a`. In both cases the name of the variable being bound is lexically apparent, and yet we must know the definition of `bnd` to understand which variable is being bound.

In general, I recommend against any language construct that can cause a change of a definition in one part of the program to modify scopes in another part of the program. This concerns ML-style opening of structures as well as most macro systems, including hygienic macros.

7.3 Future work

The compilation manager has been used for more than two years by a growing user community. The moment it was introduced it became the primary tool for maintaining SML/NJ's compiler. The model I have chosen works well in practice, and the implementation is efficient enough for everyday use. Therefore, I expect that most of the maintenance work will be concerned with cosmetic issues and minor bug fixes.

CM's analysis tracks module-level definitions only. Furthermore, the restriction it imposes on `open` is the least permissive of those I discussed. In chapter 4 I have presented an algorithm that works with a less restrictive rule. But before such an algorithm is implemented, one should investigate whether the effort would be worthwhile.

So far I have provided only the framework but not the heuristics that are necessary to implement $\underline{1}$-decomposition as part of λ-splitting. λ-splitting must also be integrated with

more recent versions of the compiler. These versions use a typed language called FLINT as their new intermediate representation [Sha97b].

The compiler's closure-conversion algorithm [SA94], though already quite sophisticated, must be improved to undo any harmful effects of λ-contract (β-reductions and let-floating). It may turn out that λ-contract itself becomes unnecessary once we start using FLINT.

7.4 Enabling technology

Cross-module inlining can open new opportunities. Now that the penalties for using abstract data types across module boundaries have been eliminated, one can clean up all the ad-hoc inlining of arithmetic primitives that pervade SML/NJ.

If we can consistently remove penalties for function calls, then it becomes practical to use new implementation techniques that rely on such function calls. In chapter 1 I have already discussed the efficient representing of lists. But the `list` type is only one example. Standard ML has always had the problem that efficient implementations of datatypes break down at functor-parameter boundaries [App93]. By representing *concrete* data types as *abstract* data types, where functions and function calls represent constructors and pattern-matches, one can solve this problem. Without inlining this would have been totally impractical, but I expect that now one can eliminate penalties in almost all cases.

Of course, there are alternatives that do not rely on cross-module inlining. Shao's work on FLINT uses constructors that are parameterized by types. Most of them are specialized at compile time, thereby removing the penalty for run-time type analysis. Intensional type information is used to handle the remaining cases. λ-splitting can perform just as well if we can ensure that inlining will take place in those cases where FLINT's parameterized constructors get specialized. I will have to investigate how our heuristics can be tuned to

achieve this goal.

However, cross-module inlining should not only be seen as yet another way of implementing certain ML peculiarities. It serves as an encouragement to use more abstraction and to write modular code because it will consistently eliminate the associated overhead.

Appendix A

CM reference manual

A.1 Group description syntax

description-file	→ *group-description* | *library-description* | *alias*
group-description	→ **Group** [*export-filter*] **is** *member-list*
library-description	→ **Library** *export-filter* **is** *member-list*
export-filter	→ *export-symbol* { *export-symbol* }
export-symbol	→ *name-space* `Identifier`
name-space	→ **structure** | **signature** | **functor** | **funsig**
member-list	→ { *member* }
member	→ `Pathname` [**:** `Class`]
alias	→ **Alias** `Pathname`

`Identifier`, `Pathname`, and `Class` are lexical classes consisting of non-empty strings without white space, colons, parentheses, or semicolons. Comments in the style of Standard ML (text between balanced pairs of `(*` and `*)`) or in the style of Scheme (text extending from a semicolon to the end of the line) are permitted. They count as delimiters like white space.

Aliases

The maximum nesting depth for aliases is limited to 32.

A.2 Examples

A simple group description

```
Group is
    main.sml
    read.sml
    write.sml
    calculate.sml
    ../lib/smlnj-lib.cm
```

A group with export filter

```
Group
    structure Table
    signature TABLE
    structure Main
    functor A
    funsig A
is
    main.sml
    a/fct.sml
    a/fsig.sml
    table/sources.cm
    RCS/parser.grm,v
```

A.3 Preprocessor

Syntax

line → *nonpreprocline*
→ *preproc*
nonpreprocline → line not starting with #
preproc → *if* { *line* } *elif-opt else-opt endif*
→ *error*
if → *beginning-of-line* # **if** *expression end-of-line*
elif-opt → { *elif* { *line* } }
elif → *beginning-of-line* # **elif** *expression end-of-line*
else-opt → [*else* { *line* }]
else → *beginning-of-line* # **else** *end-of-line*
error → *beginning-of-line* # **error** *text end-of-line*

Expressions

Expressions denote integer quantities; 0 is used for for false and non-zero values for true. There are four forms of atomic expressions:

1. Integer literals evaluate to the corresponding integer.
2. A symbol evaluates to the value bound to that symbol or to 0 if the symbol is not defined.
3. The expression `defined`(*symbol*) evaluates to 1 if *symbol* is defined, or to 0 if it is not defined.
4. The forms:
   - `defined (signature` *sigid*`)`,
   - `defined (structure` *strid*`)`,
   - `defined (functor` *fctid*`)`,

- and `defined (funsig` *fsigid*`)`

test to see if the given ML module is defined in the base environment.

Expressions are formed using a variety of binary operators, all of which are left-associative. Operators are listed with increasing precedence. Those that appear on the same line have equal binding strength:

```
||
&&
== !=
< <= > >=
+ -
/ *
```

Logical disjunction `||` and conjunction `&&` are short-circuiting operations. The unary operators for logical and numerical negation are `!` and `-`, respectively. Parentheses can be used for grouping.

Predefined symbols

Depending on architecture, operating system, and configuration one symbol out of each of the following groups will be predefined to 1:

OS	Architecture	Byte order	ML word size
OPSYS_UNIX OPSYS_WIN32 OPSYS_MACOS OPSYS_OS2	ARCH_SPARC ARCH_MIPS ARCH_ALPHA ARCH_X86 ARCH_HPPA ARCH_RS6000 ARCH_POWERPC	BIG_ENDIAN LITTLE_ENDIAN	SIZE_32 SIZE_64

Additionally, in version *xxx.yy* of the compiler `SMLNJ_VERSION` will be set to *xxx*, and `SMLNJ_MINOR_VERSION` evaluates to *yy*.

Example

```
Group is
    a.sml
    b.sml
# if (SMLNJ_VERSION >= 109 || defined(structure SMLofNJ))
    util.sml
# elif (SMLNJ_VERSION < 108)
# error The version of SML/NJ is too old for this software.
# else
    util-workaround.sml
# endif
```

A.4 Default classes

`.sig .sml .fun`	Sml
`.grm .y`	MLYacc [TA90]
`.lex .l`	MLLex [AMT89]
`.burg`	MLBurg [GG93]
`,v`	RCS [Tic85]
`.cm`	CMFile
`.nw`	Noweb

A.5 Available tools

Items shown are predefined tools in CM's extensible toolbox. The *out-of-date* condition mentioned here refers to the situation where at least one of the targets is either missing or older than the source.

member class	processor	source	targets	conditions
MLYacc	ml-yacc	*file*	*file*.sig *file*.sml	out-of-date
MLLex	ml-lex	*file*	*file*.sml	out-of-date
MLBurg	ml-burg	*file*.burg	*file*.sml	out-of-date
		file	*file*.sml	
RCS	co -q	*file*,v	*file*	target missing
		RCS/*file*,v	*file*	
Noweb	notangle	*file*.nw	*file*.sig *file*.sml	out-of-date
		file	*file*.sig *file*.sml	
		root@*file*.nw	*root*	
		root@*file*	*root*	

A.6 Export rules

A group with export filter exports all symbols listed by its filter regardless of their origin.

A group without export filter exports all symbols defined by sources of the group and all non-masked symbols exported by ordinary subgroups of the group. Note, that libraries are required to have an export filter.

A.7 Binfile names

Architecture	binfile directory
DEC Alpha (32 bit)	`CM/alpha32-`*os*
Sparc	`CM/sparc-`*os*
MIPS (little endian)	`CM/mipsel-`*os*
MIPS (big endian)	`CM/mipseb-`*os*
HP-PA	`CM/hppa-`*os*
IBM RS6000	`CM/rs6000-`*os*
Intel x86	`CM/x86-`*os*
bytecode	`CM/bytecode-`*os*

A.8 Feature summary

Most of CM's functionality must be accessed through members of a structure called `CM`. For brevity I will often drop the prefix `CM` and write `make` instead of `CM.make`.

Root description file

Most actions of CM are driven by a DAG resulting from dependency analysis. In order to perform such an analysis it is necessary to know the root group of the hierarchy to be analyzed. The explicit argument used by many functions for naming the root description file is a string.

All functions in the `CM` structure that take an explicit root description as one of their arguments have a counterpart without such a parameter. The counterpart implicitly uses an internal default of the form `(!rootfile)`. This reference cell is not directly accessible. Instead, its contents can be set via

```
val set_root:  string -> unit
```

The initial value is `"sources.cm"`, unless the operating system environment variable `CM_ROOT` is set, in which case its value will be used. Note, that the value of `!rootfile` is always treated as the name of a CM-style description file—regardless of its name.

As a lexical convention, names of functions taking an explicit string argument for naming the root description file end with an apostrophe. The names of the other functions—those that use the default—can be obtained by stripping away the apostrophe. Example:

```
CM.make ();
```

is roughly equivalent to

```
CM.make' (!rootfile);
```

Compilation

```
val make':  string -> unit
val make:  unit -> unit

val recompile':  string -> unit
val recompile:  unit -> unit
```

`CM.recompile()` analyzes the system and performs all necessary recompilation steps. A recompilation step is necessary if one of the following is true:

- the binfile is missing
- the binfile is older than its source file
- a predecessor in the dependency graph has been recompiled and CM has discovered that its new version is not compatible with the existing binfile

The last point is important: Unlike `make` in UNIX, CM is often able to avoid recompiling certain units even if their predecessors had been recompiled.

If the result of the predecessor's compilation is still compatible with the existing bin-file, then no further action is necessary. In particular, just touching a file (changing its modification time stamp) does not cause a recompilation of the entire system. Also, if a change to a source file does not change the interface (the static environment) exported by the compilation unit, then it will not trigger subsequent recompilations of dependent files.

`CM.recompile()` only compiles but does not execute any code. Therefore, no new bindings will be added to the top-level environment. This function does not keep compiled code in main memory to avoid wasting resources.

`CM.make()` and `CM.recompile()` perform the same analyses and recompilation steps. But `CM.make()` also executes the code in all units (i.e., not only the ones that needed to be recompiled). If there are no errors, then the top-level environment will be augmented with bindings for the symbols exported by the root group.

Compilation errors

In the case of an error while compiling a source, CM cannot proceed processing other files that depend on the one with the error. However, this does not affect unrelated branches of the dependency graph. Therefore, CM can press on and "keep going." The function:

```
val keep_going:  bool option -> bool
```

called with an argument of `(SOME true)` will enable this feature.

Since CM must run all tools and parse all source files before it can even start to build the dependency graph it will not be able to continue when errors occur during any of those operations.

Topological sort

Sometimes it is useful to obtain a topologically sorted list of a program's sources. The following functions calculate such lists:

```
val names': string -> string list
val names: unit -> string list

val binfiles': string -> string list
val binfiles: unit -> string list

val strings': string -> string list
val strings: unit -> string list
```

All lists produced by these commands are sorted according to the same principle: Let $S(s)$ be the SML source associated with string s. If s and t are distinct members of the list and s appears to the left of t, then $S(s)$ does not depend—either directly or indirectly—on $S(t)$.

`CM.names` produces lists of SML source file names, `CM.binfiles` returns lists of binfile names, and calls to `CM.strings` yield lists of descriptions for SML sources. These descriptions are often just the file name—except when the SML source is the result of running some tool, in which case this fact will be mentioned by the description.

The function `CM.mkusefile` allows one to create a file with a topologically sorted list of `use` commands. The second or sole string argument is the name of the file to be written.

```
val mkusefile': string * string -> unit
val mkusefile: string -> unit
```

This can be useful for maintaining stand-alone versions of a system.

Stand-alone systems

The technique of sequentially reading and compiling a collection of SML source files is inherently less expressive in its management of name spaces than what can be done with CM's groups, libraries, and export filters (→ Chapter 3).

Therefore, a much better alternative to `CM.mkusefile` is to use `CM.sa` (`sa` = *S*tand *A*lone). Like `mkusefile` this function produces a small program expressed in SML source language that can subsequently be read and compiled by SML/NJ:

```
val sa':  string * string -> unit
val sa:  string -> unit
```

However, such a program—unlike the one generated by `mkusefile`—uses special support from SML/NJ to reproduce precisely the same treatment of namespaces that CM itself would use internally when running `CM.make`. Furthermore, the program will check for the presence of binfiles and load those whenever possible. This can be a useful aid when building systems that no longer depend on the presence of CM. CM itself is bootstrapped using this facility.

Visualization

Sometimes it is helpful to look at a picture of the DAG representing the dependencies of a program. The function `CM.dot` can be used to produce input for the DOT program [KN93], which is a tool for automatically drawing such graphs. Figure 2.1 on page 24 shows such a DOT drawing of the dependency graph for CML [Rep91].

```
val dot':  string * string -> unit
val dot:  string -> unit
```

An invocation of this function writes a DOT-specification into the file that was named by the second (or sole) argument. DOT-specifications contain ordinary text. Layout parameters are located near the top of the file. Any text editor can be used to adjust them when

necessary.

The pictures use ellipses for `.sig`-files, rectangular boxes for `.sml`-files, and diamond-shaped boxes for all others. Lines between nodes show direct dependencies. Solid lines indicate that the nodes belong to the same group; dashed lines are used for edges which cross between different groups. A dotted line connects a node to the name of a symbol imported from the base system. These symbols are displayed as plain text labels.

Stabilization

An invocation of `CM.stabilize false` stabilizes the root group of the system. In the process of stabilization CM runs the equivalent of `CM.recompile()` in order to update all of the binfiles. Once this has been successful it creates the stablefile.

If the group to be stabilized refers to other groups, then those subgroups should already be stable. `CM.stabilize true` will also process and stabilize all subgroups, sub-subgroups, and so forth:

```
val stabilize': string * bool -> unit
val stabilize: bool -> unit
```

An invocation of `CM.destabilize` reverts a stable group to its original, non-stable state:

```
val destabilize': string -> unit
val destabilize: unit -> unit
```

Preprocessor symbols

The top-level structure `CM` contains a substructure `SymVal` with the following functions:

```
val lookup: string -> int option
val define: string * int -> unit
val undef: string -> unit
val undefall: unit -> unit
```

Definitions for specific preprocessor symbols can be added using `define` and removed

via `undef`. An invocation of `undefall` clears all definitions. The `lookup` function provides a way of determining the current value associated with a given symbol.

Autoloader

With autoloading enabled, the command:

```
CM.autoload' "util.cm";
```

makes all definitions exported from the group described by `util.cm` available at the SML top-level. However, it does that without actually compiling or loading anything. Instead, CM monitors code entered at top level. If it finds that a symbol exported from `util.cm` is being used and that the definition for that symbol has not already been supplied earlier, then it will calculate and load the minimal set of sources required to provide the desired definition.

Here is a summary of the functions used to control the autoloader:

```
val autoloading:  bool option -> bool

val autoload:  unit -> unit
val autoload':  string -> unit

val autoList:  unit -> string list
val clearAutoList:  unit -> unit
```

The autoloading mechanism can be turned on or off using the `autoloading` command. Since it always returns the previous setting, one can invoke it with `NONE` to query the current status without actually changing it.

The functions `autoload` and `autoload'` are analogous to `make` and `make'`, except they do not actually load any module but register it for autoloading instead. The function `autoList` returns a list of all groups that are currently registered for autoloading and `clearAutoList` resets the corresponding internal registry.

Miscellaneous

The global behavior of CM is controlled by a few state variables. Part of this state is a set of boolean flags which can be manipulated by calling interface functions of type `bool option -> bool`. All of them always return the previous setting. An argument of `SOME` *x* is used to set the value to *x*, `NONE` only queries the current state without modifying it. I have already shown such a state variable in the case of `autoloading`:

```
val keep_going:  bool option -> bool (*CM_KEEP_GOING*)
val verbose:  bool option -> bool (*CM_VERBOSE*)
val show_exports:  bool option -> bool (*CM_SHOW_EXPORTS*)
val parse_caching:  int option -> int
```

The comments show names of (shell-)environment variables that can be set to `true` or `false` in order to define an initial value for the corresponding state variable.

I have described `keep_going` already; `verbose`, which is initially `true`, can be used to silence CM's chatting about its current activities by setting it to `false`. The `show_exports` flag—when set to `true`—will cause CM to report the symbols exported by filter-less groups.

With `parse_caching` one can limit the number of parse trees that are kept in main memory. Keeping parse trees after dependency analysis and reusing them for compilation can save time. However, with large programs this uses too much memory, so CM offers a way of tuning its behavior.

The path—an internally kept string list—specifies a set of alternative directories, which CM must consider in the case that a named subgroup cannot be found locally. An initial path is taken from `CM_PATH` (a shell-variable) or from a built-in default. `CM_PATH` must specify a colon-separated list of directories in the style of `/bin/sh`'s `PATH`. One can change the path interactively by invoking `CM.set_path`:

```
val set_path:  string list option -> string list
```

This function also returns the old setting. Passing `NONE` as an argument can be used to query the current setting.

`CM.sweep` and `CM.clear` are used to explicitly manipulate CM's in-core caches:

```
val sweep:  unit -> unit
val clear:  unit -> unit
```

`CM.sweep()` removes all internally cached binfiles that are no longer consistent with the external (file-system) cache or the respective sources. For correctness this will never be necessary because CM always performs consistency checks before using a cached item. However, it can help to reduce memory usage. `CM.clear()` empties all in-memory caches.

A.9 All CM commands

Setting the default root file

```
val set_root:  string -> unit
```

Compilation

```
val make':  string -> unit
val make:  unit -> unit

val recompile':  string -> unit
val recompile:  unit -> unit
```

Dealing with compilation errors

```
val keep_going:  bool option -> bool
```

Topological sort

```
val names':  string -> string list
val names:  unit -> string list

val binfiles':  string -> string list
val binfiles:  unit -> string list

val strings':  string -> string list
val strings:  unit -> string list
```

Generating list of `use` commands

```
val mkusefile':  string * string -> unit
val mkusefile:  string -> unit
```

Building stand-alone systems

```
val sa':  string * string -> string
val sa:  string -> string
```

Graph drawing

```
val dot':  string * string -> unit
val dot:  string -> unit
```

Stabilization

```
val stabilize':  string * bool -> unit
val stabilize:  bool -> unit

val destabilize':  string -> unit
val destabilize:  unit -> unit
```

Preprocessor symbols

```
val lookup:  string -> int option
val define:  string * int -> unit
val undef:  string -> unit
val undefall:  unit -> unit
```

Autoloader

```
val autoloading:  bool option -> bool

val autoload:  unit -> unit
val autoload':  string -> unit

val autoList:  unit -> string list
val clearAutoList:  unit -> unit
```

Miscellaneous

```
val keep_going:  bool option -> bool (* CM_KEEP_GOING *)
val verbose:  bool option -> bool (* CM_VERBOSE *)
val show_exports:  bool option -> bool (* CM_SHOW_EXPORTS *)
val parse_caching:  int option -> int

val set_path:  string list option -> string list

val sweep:  unit -> unit
val clear:  unit -> unit
```

A.10 Adding new tools to CM

Structure `CM.Tools: CMTOOLS` is a repository of types and functions useful for adding new tools to CM. It is not necessary to recompile CM when augmenting it this way.

The interface for customizing CM is not as convenient as writing scripts for `make`. However, it was designed to be very general because the full power of SML can be used.

For the most common situations there are useful pre-defined building blocks available in `CM.Tools`.

Basics

Tools correspond to "tool classes." Source files are classified according to what tool is needed to process them.

The notion of a tool class includes:

1. A *name* for the class. This is a simple string consisting of lower-case letters.

2. A *rule*: The rule is a function from member names (strings) to a "target" lists, where each target is another member name together with an optional tool class. Most of the time member names refer to files, but this is not always the case, as can be seen in the case of the *root*@*file*.`nw` notation used by the `noweb` tool.

3. A *validator*: The validator takes a source file name and the target list as produced by the rule and determines whether or not the tool needs to be invoked at all.

4. A *processor*: The processor implements the actual tool, i.e., it takes source file name and targets and runs the tool.

```
type fname = string
type class = string
type target = fname * class option
```

Member names passed to rules, validators, and processors are exactly the ones that appear in the description file. Relative names are resolved relative to the directory the description file appears in. I call this directory the "context." When running a validator or a processor CM temporarily changes its working directory to the context to allow for relative filenames to be processed correctly.

```
type validator =
  { source:  fname, targets:  target list } -> bool
type processor =
  { source:  fname, targets:  target list } -> unit
```

Ideally, rules should also run with the working directory set to the corresponding context. However, changing the working directory can be relatively expensive and is often not necessary for a rule to work correctly. To account for that, CM offers a somewhat involved interface for its rules, so it is up to the programmer to decide whether or not the context should be set or not.

```
type rulethunk = unit -> target list
type rulecontext = rulethunk -> target list
type rule = fname * rulecontext -> target list
```

Generic rules take the name of the source and also a rule context as an argument. The context is a function that accepts a parameterless procedure (the "rule thunk") as an argument. It then calls the thunk with the working directory set properly. The part of a rule's implementation that depends on the working directory must therefore be placed inside the thunk.

The thunk is not needed for rules that do not care about the working directory. In this case one can safely ignore (and never call) the context function, thereby avoiding the underlying calls to the `chdir` system call. Most of the time this will be the case when target file names can be derived from the source name directly. Only in more complicated situations it may be that the rule needs to open the file and inspect its contents.

The most common rules ("simple rules") will either always ignore the context or always use it. This is captured by the type `simplerule`. Conversion routines `dontcare` and `withcontext` convert simple rules to generic ones:

```
type simplerule = fname -> target list

val dontcare:  simplerule -> rule
val withcontext:  simplerule -> rule
```

To register a new tool class with CM, one must use `addToolClass`:

```
val addToolClass:
    { class:  class, rule:  rule,
      validator:  validator,
      processor:  processor } -> unit
```

Tool classes and classification

The tool class for a given source can be determined in three different ways:

1. If the source name is a member in a CM description file, then it can be followed by a colon and the name of the corresponding tool class. Internally, tool classes use lower-case names; tool names in CM description files are case-insensitive.

2. If a source is the product of running a tool on another source, then the rule for that tool may have specified the tool class for its target.

3. If the description file omits the class specification, or if a tool's rule does not provide this information for one of its targets, then CM tries to infer the class name automatically.

Automatic classification and classifiers

Automatic classification is done based on the name of the source. In the most common case the decision is based on the filename extension ("suffix"). However, it is also possible to take more or all of the file name into account. Classification is done by "classifiers," which come in two flavors:

```
datatype classifier =
    SFX_CLASSIFIER of string -> class option
  | GEN_CLASSIFIER of fname -> class option
```

Suffix classifier (`SFX_CLASSIFIER`) A function from suffix strings to class option.

General classifier (GEN_CLASSIFIER) A function from source names to class option.

Classifiers can be added using addClassifier:

```
val addClassifier:  classifier -> unit
```

The defaultClassOf function provides a way of invoking the built-in classification mechanism explicitly:

```
val defaultClassOf:  fname -> class option
```

Structure CM.Tools contains some functions for conveniently creating the most common classifiers, validators, and processors:

- Make a classifier which looks for a specific file name suffix:

```
val stdSfxClassifier:
  { sfx:  string, class:  class } -> classifier
```

- Two validators—one verifies time stamp consistency, the other one only probes the existence of the targets:

```
val stdTStampValidator:  validator
val stdExistenceValidator:  validator
```

- Make a processor that runs a given shell command with the source name as its only argument. The tool argument is used when raising the ToolError exception upon failure (see below).

```
val stdShellProcessor:
  { command:  string, tool:  string } -> processor
```

Tools exception

A tool should raise exception ToolError to signal that it was unable to complete its operation normally:

```
exception ToolError of { msg:string, tool:string }
```

The fields `msg` and `tool` can contain arbitrary strings. However, `tool` is intended to describe the originator of the exception, while `msg` should give a more detailed indication of what exactly went wrong.

Recommended strategy for adding tools

When adding new tools to CM, I recommend writing a functor that takes the `CM.Tools` structure as its argument. This way the tool can easily be installed by instantiating the functor.

Suppose one wants to extend CM with a tool for an improved version of ML-Yacc. The old ML-Yacc should still be available, though. One creates a new class `BetterYacc`, a processor that runs the command `new-ml-yacc`, and has CM recognize files whose names end in `.ngrm` or `.ny` as input for this tool.

The rule maps input file *x* to output files *x*`.sig` and *x*`.sml`—both of them classified as belonging to class `Sml`, and the validator compares time stamps in the file system.

The following sample code is highly stylized; experienced programmers could easily "compress" it to only a few lines:

```
functor YaccSourceFun (structure Tools:  CMTOOLS) = struct
 local
   val command = "new-ml-yacc"

   fun simplerule source = let
     val smlfile = source ^ ".sml"
     val sigfile = source ^ ".sig"
     fun sml f = (f, SOME "sml")
   in
     [sml sigfile, sml smlfile]
   end

   val validator = Tools.stdTStampValidator

   val processor = Tools.stdShellProcessor
         { command = command, tool = "Better-ML-Yacc" }

   (* install BetterYacc class *)
   open Tools
   val class = "betteryacc"
   fun sfx s = addClassifier
         (stdSfxClassifier { sfx = s, class = class })
 in
   val _ = addToolClass
         { class = class, rule = dontcare simplerule,
           validator = validator, processor = processor }
   val _ = sfx "ngrm"
   val _ = sfx "ny"
 end
end
```

A.11 Using CM for compiling the compiler

Introduction

One important use of CM from the perspective of the compiler development team is the maintenance of SML/NJ's compiler. The compiler is "just another SML program," but the circumstances under which it is integrated into the rest of the system—especially the fact

that it must compile itself—can require non-standard treatment [App94]. For that reason CM offers a special interface, which facilitates controlling "batch" compilation, bootstrapping, and the process of retargeting the compiler to a different machine architecture.

Batch compilation—the structure CMB

The batch compiler is implemented as a structure called `CMB`, which uses its own private copy of a full-fledged `structure CM` whose behavior has been modified in minor ways to account for the specifics of recompiling the compiler.

In order to use `CMB` it is necessary to run CM in the compiler's source directory. Batch compilation is started by invoking `CMB.make()`:

```
val make:  unit -> unit
```

Batch compilation—the internals

Phases

1. Construction of the core environment by compiling some dedicated source files located in the `boot` directory. This step compiles three files, the names of which are hardwired into `CMB`:

   boot/assembly.sig defines a signature that describes the interface to the run-time system. (The run-time system is not written in SML.)

   boot/dummy.sml implements a structure that matches the run-time system's signature. This structure is used as a placeholder in order to allow further compile-steps to proceed. The code obtained from compiling this file is never used because the boot procedure will replace it with the actual run-time system.

boot/core.sml implements the core environment. This file is the only one that directly depends on the result of compiling `boot/dummy.sml`. The place of the latter will be taken by the run-time system. It is necessary to compile `boot/core.sml` in a special way to account for this.

2. Compilation of the remaining sources in the boot directory in some fixed order. This order is specified in `boot/all-files.cm`. The format of this file is a group description file, but CM does not perform dependency analysis. It uses the order in which the members are listed.
3. CM reads the description file `boot/pervasives.cm` and compiles the members listed there. Anything defined here will eventually make up the so-called pervasive environment, which also serves as the initial basis for compiling the actual compiler sources. Now the compilation of the `boot` directory has completed.
4. Analysis of dependencies among the compiler's source files. The bulk of the compiler's sources can be understood by CM's dependency analyzer directly.
5. Recompilation of the compiler's sources as far as necessary. This step also takes advantage of CM's basic functionality such as cutoff recompilation. However, there are some differences:
   - All binfiles are stored in one single directory. The name of that directory depends on target architecture and operating system. For example, when compiling for a DEC Alpha machine that is running a flavor of Unix, binfiles are placed into `bin.alpha32-unix`.
   - The base environment for the compilation is the one constructed by compiling the `boot` directory.
6. Generating list files.

architecture	stem	binlist
DEC Alpha (32 bit)	`Alpha32`	BINLIST.alpha32
Sparc	`Sparc`	BINLIST.sparc
MIPS (little endian)	`MipsLittle`	BINLIST.mipsel
MIPS (big endian)	`MipsBig`	BINLIST.mipseb
HP-PA	`Hppa`	BINLIST.hppa
IBM RS6000	`RS6000`	BINLIST.rs6000
Intel x86	`X86`	BINLIST.x86
bytecode	`ByteCode`	BINLIST.bytecode

Table A.1: Target architectures, associated name stems, and list-files

List-files

In order to finally build a new SML/NJ system it is necessary to combine the various binfiles. However, this cannot be done by CM or other SML programs. The binfiles must be loaded into an empty run-time system.

The boot mechanism of the run-time system reads certain list files. They are used to record the names of binfiles that are needed to build a new compiler. The two most important list files—`BOOTLIST` and `BINLIST`—are written by the batch compiler in step 6. Both of them (like all the other list-files as well) are located in the binfile directory.

`BOOTLIST` contains the names of binfiles for the various sources in the `boot` directory. However, neither `assembly.sig` nor `dummy.sml` are mentioned.

`BINLIST` lists all remaining binfiles necessary to build SML/NJ on the target machine. In order to create this file CM selects one single compilation unit as the root of the dependency graph and writes a topologically sorted list of binfiles belonging to nodes reachable from this root. The rootfile selected must define a structure whose name depends on the target architecture. It consists of `Int` concatenated with the stem that can be looked up in table A.1. For example, for a target architecture of "MIPS big endian" the structure's name is `IntMipsBig`.

In addition to `BINLIST` there will be another list-file—`SRCLIST`—naming the sources for these binfiles. This file will be used to find the sources in the case that static environments must be re-created from sources at boot-time instead of simply extracting them from binfiles. Similarly, `BOOTSRC` names the sources for binfiles listed in `BOOTLIST`. Note that file names in `BOOTLIST` and `BINLIST` are relative to the binfile directory, while file names in `BOOTSRC` and `SRCLIST` are relative to the compiler's source directory.

Finally, there will be a set of list-files—one per supported target architecture (→ Table A.1)—which specify names of binfiles for cross compilers. Again, each of these files contains a topologically sorted list of binfiles. Their creation proceeds completely analogous to creating `BINLIST`: There must be one compilation unit defining some structure whose name depends on the cross compiler's target architecture. This unit is selected as the root of the dependency graph used to produce the topological ordering. To obtain the name of the structure, CM concatenates the stem shown in table A.1 with `VisComp`. Example: The structure pertaining to cross compilers for the HP-PA architecture is `HppaVisComp`.

Controlling the batch compiler

Files

There are three files that are used to control batch compilation. All of them have the format of a group description without export filter.

boot/all-files.cm The files `boot/all-files.cm` lists the source files in directory `boot` with the exception of:

- `boot/assembly.sig`,
- `boot/dummy.sml`, and
- `boot/core.sml`.

Even though it has the format of a group description it will not be used as input for dependency analysis. Instead, the order of members in this "group" matters—they are compiled top-to-bottom.

boot/pervasives.cm After all members of `boot/all-files.cm` have been compiled CM will then process the members of `boot/pervasives.cm`. Members of this file are not subject to dependency analysis. The environment resulting from compiling `boot/pervasives.cm` is remembered as the *pervasive environment* (aka. *initial basis*).

all-files.cm File `all-files.cm` is the root description for the remaining system. It will serve as input to the dependency analyzer. Therefore, the order of its members does not matter.

Flags

`CM` shares its internal state with `CMB`. This means that the interface functions described in section A.8 can also be used to control the behavior of `CMB`. Example:

```
CMB.CM.verbose (SOME false);
CMB.CM.keep_going (SOME true);
```

This is equivalent to:

```
CM.verbose (SOME false);
CM.keep_going (SOME true);
```

Cross-compilation and retargeting

Retargeting the batch compiler corresponds to creating a new instance of `CMB` where the compiler has been replaced with one that produces code for a different architecture. In order to be able to retarget, one must already have a complete set of working binfiles for

the compiler running on the host architecture.

Structure CMR contains the retarget function, which is responsible for building a new CMB:

```
structure CMR: sig
  val retarget:
       { bindir:  string, cpu:  string,
         os:  string } -> unit
end
```

For example, if the host machine is a DEC Alpha and one wants to cross-compile for a Sparc running some flavor of UNIX, then one must invoke:

```
CMR.retarget
  { bindir = "bin.alpha32",
    cpu = "sparc", os = "unix" };
```

This command reads bin.alpha32/BINLIST.sparc and loads the files listed therein. The result is a compiler that produces code for a Sparc machine running UNIX, even though it itself still runs on the Alpha. The new compiler is used to construct a new structure CMB which then replaces the old one.

Known CPU names are alpha32, sparc, mipsel, mipseb, hppa, rs6000, x86, and bytecode. Operating system specifiers are unix, macos, os2, and win32.

Once CMR.retarget has completed successfully it suffices to run

```
CMB.make ();
```

in order to produce Sparc binfiles for the compiler.

Appendix B

Source code for FFT benchmark

B.1 An abstract datatype for complex numbers

I implemented the following structure `Complex` as a separate compilation unit. For timing measurement it was later used as part of an implementation of the Fast Fourier Transform (→ Chapter 5 and section B.2).

```
structure Complex = struct

    type complex = { re: real, im: real }

    fun re (r: real) = { re = r, im = 0.0 }
    fun im (i: real) = { re = 0.0, im = i }

    fun add (x1: complex, x2) = let
        val { re = r1, im = i1 } = x1
        val { re = r2, im = i2 } = x2
    in
        { re = r1 + r2, im = i1 + i2 }
    end

    fun sub (x1: complex, x2) = let
        val { re = r1, im = i1 } = x1
        val { re = r2, im = i2 } = x2
    in
        { re = r1 - r2, im = i1 - i2 }
    end
```

```
    fun mul (x1: complex, x2) = let
        val { re = r1, im = i1 } = x1
        val { re = r2, im = i2 } = x2
    in
        { re = r1 * r2 - i1 * i2, im = r1 * i2 + r2 * i1 }
    end

    fun quo (x1: complex, x2) = let
        val { re = r1, im = i1 } = x1
        val { re = r2, im = i2 } = x2
        val ccdd = r2 * r2 + i2 * i2
        val abcd = (r1 + i1) * (r2 - i2)
        val ac = r1 * r2
        val bd = i1 * i2
    in
        { re = (r1 * r2 + i1 * i2) / ccdd,
          im = (i1 * r2 - r1 * i2) / ccdd }
    end

    fun exp ({ re, im }: complex) = let
        val f = Math.exp re
    in
        { re = f * Math.cos im, im = f * Math.sin im }
    end

    fun inv ({ re, im }: complex) = let
        val d = re * re + im * im
    in
        { re = re / d, im = ~ (im / d) }
    end
end
```

B.2 Sample Implementation of FFT

Together with a suitable realization of structure `Complex` (→ Section B.1) the following recursive implementation of FFT was used to obtain timing measurements for demonstrating the effectiveness of λ-splitting (→ Chapter 5).

```
structure RecFFT = struct

    (* operations on complex numbers from structure Complex *)
    val ++ = Complex.add
    val -- = Complex.sub
    val ** = Complex.mul
```

```
    val // = Complex.quo

    (* make them infix operators like *, /, +, - *)
    infix 7 ** //
    infix 6 ++ --

    (* some useful constants *)
    val pi = Math.pi (* 3.14159265358979323846264338 *)
    val c0 = Complex.re 0.0          (* complex 0 *)
    val c1 = Complex.re 1.0          (* complex 1 *)
    val pii2 = Complex.im (2.0 * pi) (* 2*pi*i *)

    fun fft (a, forward) = let

        (* split polynomial into two subpolynomials *)
        fun split [] = ([], [])
          | split (x1 :: x2 :: r) = let
                val (r1, r2) = split r
            in
                (x1 :: r1, x2 :: r2)
            end
          | split _ = raise Fail "split: odd list"

        (* the recursive FFT *)
        (* n = length a, wn = e^(2*pi*i/n) *)
        fun rfft (a as _ :: _ :: _, wn) =
            let
                val (a0, a1) = split a
                val wn2 = wn ** wn
                fun loop (_, [], _, yy0, yy1) = rev (yy1 @ yy0)
                  | loop (w, y0 as y0h :: y0t,
                             y1 as y1h :: y1t,
                             yy0, yy1) =
                    loop (w ** wn,
                          y0t, y1t,
                          (y0h ++ w ** y1h) :: yy0,
                          (y0h -- w ** y1h) :: yy1)
                  | loop _ = raise Fail "rfft:loop: odd list"
            in
                loop (c1, rfft (a0, wn2), rfft (a1, wn2), [], [])
            end
          | rfft (a, _) = a

        val (a, n, _) = FFTUtil.expand a
        val cn = Complex.re (Real.fromInt n)
        val wn = Complex.exp (pii2 // cn)
    in
        if forward then rfft (a, wn)
        else map (fn x => x // cn) (rfft (a, Complex.inv wn))
    end
end
```

Appendix C

Benchmark results in numbers

C.1 Compiler passes as benchmarks

λ-splitting. Tables C.1 and C.2 show the running times of various compiler passes of the SML/NJ compiler for the case that the compiler itself was compiled with and without λ-splitting. Individual compiler passes serve as my benchmarks. For either measurement I had λ-contract enabled. Therefore, differences in timing are due to λ-splitting itself. All times are shown in seconds.

λ-contract. I ran my benchmarks (phases of the SML/NJ compiler) unmodified and with λ-contract enabled. The resulting times for the Alpha and the Pentium are shown in tables C.3 and C.4.

λ-splitting applied to a cleanly modularized program. I reinserted signature constraints and abstractions that had been removed from the source of the SML/NJ compiler by the efficiency-conscious software engineers. The resulting code represents a cleanly modularized program, which is unable to benefit from SML/NJ's ad-hoc inlining mechanism.

Benchmark	λ-contract only	λ-contract + λ-split
parse	6.75	6.81
elaborate	147.33	136.40
pickle	21.81	19.58
unpickle	4.22	4.17
translate	6.45	6.13
codeopt	1.62	1.60
convert	4.33	3.70
lcontract	14.86	14.53
lsplit	4.70	4.23
cpsopt	14.22	13.15
fmclose	1.82	1.55
closure	7.60	6.75
cpsgen	26.95	25.55
liveness	4.65	4.71
schedule	3.05	2.82
execute	56.77	49.82
ALL	330.45	304.67

Table C.1: **Alpha.** Timing measurements for λ-splitting on a DEC Alpha 21064. The numbers indicate seconds of CPU time.

Benchmark	λ-contract only	λ-contract + λ-split
parse	14.45	13.82
elaborate	250.11	236.45
pickle	41.43	42.23
unpickle	10.13	9.85
translate	15.35	13.81
codeopt	3.76	3.74
convert	9.11	8.24
lcontract	35.09	33.96
lsplit	11.05	10.12
cpsopt	28.50	28.47
fmclose	4.28	3.62
closure	18.47	17.65
spill	3.46	3.39
cpsgen	29.69	31.31
execute	103.90	90.45
ALL	586.16	554.22

Table C.2: **Pentium.** Timing measurements for λ-splitting on an Intel Pentium. The numbers indicate seconds of CPU time.

Benchmark	unmodified	λ-contract
parse	6.75	6.75
elaborate	134.92	147.33
pickle	21.80	21.81
unpickle	4.31	4.22
translate	6.21	6.45
codeopt	1.62	1.62
convert	4.26	4.33
lcontract	14.72	14.86
lsplit	4.94	4.70
cpsopt	14.01	14.22
fmclose	1.81	1.82
closure	7.75	7.60
cpsgen	26.63	26.95
liveness	4.78	4.65
schedule	2.95	3.05
execute	56.52	56.77
all	316.98	330.45

Table C.3: **Alpha.** Timing measurements for λ-contract on a DEC Alpha 21064. The numbers indicate seconds of CPU time.

Benchmark	unmodified	λ-contract
parse	14.45	13.82
elaborate	250.11	236.45
pickle	41.43	42.23
unpickle	10.13	9.85
translate	15.35	13.81
codeopt	3.76	3.74
convert	9.11	8.24
lcontract	35.09	33.96
lsplit	11.05	10.12
cpsopt	28.50	28.47
fmclose	4.28	3.62
closure	18.47	17.65
spill	3.46	3.39
cpsgen	29.69	31.31
execute	103.90	90.45
ALL	586.16	554.22

Table C.4: **Pentium.** Timing measurements for λ-contract on an Intel Pentium. The numbers indicate seconds of CPU time.

Benchmark	original	unoptimized	λ-splitting
parse	6.75	10.93	7.53
elaborate	147.33	198.71	143.89
pickle	21.81	36.00	24.37
unpickle	4.22	9.88	4.25
translate	6.45	8.42	6.87
codeopt	1.62	2.89	2.29
convert	4.33	6.60	4.41
lcontract	14.86	36.89	14.26
lsplit	4.70	9.99	5.17
cpsopt	14.22	25.55	17.21
fmclose	1.82	3.33	1.94
closure	7.60	16.97	10.19
cpsgen	26.95	66.29	38.03
liveness	4.65	11.31	7.40
schedule	3.05	8.86	3.32
execute	56.77	70.05	52.64
ALL	330.45	528.68	347.51

Table C.5: **λ-splitting for modularized code.** Timing measurements for λ-splitting vs. ad-hoc inlining on a DEC Alpha 21064. The numbers indicate seconds of CPU time.

Table C.5 shows the running times of my benchmarks for three cases:

original: The original benchmarks compiled using ad-hoc inlining.

unoptimized: Benchmarks where signature constraints had been re-inserted, and which, thus, were compiled without any cross-module inlining.

λ-splitting: Benchmarks with signature constraints as in the unoptimized case but compiled using my λ-splitting optimization.

C.2 Fast Fourier Transform

Table C.6 shows the timing results that I obtained from running a simple implementation of FFT (→ Appendix B). I conducted experiments with four different problem sizes (FFT

problem size	original	λ-splitting	one file
$50,000 \times 1$	10.2	9.0	8.8
$50,000 \times 2$	19.8	18.3	18.4
$100,000 \times 1$	25.1	22.6	22.6
$100,000 \times 2$	52.0	45.8	45.5

Table C.6: **λ-splitting for an implementation of FFT.** Timing measurements for FFT on a DEC Alpha 21064. The numbers indicate seconds of CPU time.

of 50,000-element list, FFT + inverse FFT of 50,000-element list, FFT of 100,000-element list, FFT + inverse FFT of 100,000-element list) and ran each with code compiled in three different ways (using the original compiler, using a compiler with λ-splitting enabled, compiling the entire benchmark as one compilation unit using the original compiler).

Bibliography

[Ada80] Military standard: Ada programming language. Technical Report MIL-STD-1815, Department of Defense, Naval Publications and Forms Center, Philadelphia, PA, 1980.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[ALL96] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *Proc. 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 83–91. ACM Press, May 1996.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, volume 29, pages 13–23. ACM Press, June 1994.

[AMT89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, December 1989.

[Ans90] American National Standards Institute, Inc., New York. *American National Standard for Information Systems, Programming Language C ANSI X3.159–1989*, 1990.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[App93] Andrew W. Appel. A critique of Standard ML. *J. Functional Programming*, 3(4):391–430, 1993.

[App94] Andrew W. Appel. Axiomatic bootstrapping: A guide for the compiler hacker. *ACM Transactions on Programming Languages and Systems*, 16(6):1699–1718, November 1994.

[App97] Andrew Appel. *Modern Compiler Implementation in ML, Basic Techniques*. Cambridge University Press, 1997.

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM TOSEM*, 3(1):3–28, January 1994.

[BA97] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 112–124. ACM Press, June 1997.

[Bar81] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[BDGK94] Koen De Bosschere, Saumya Debray, David Guneman, and Sampath Kannan. Call forwarding: a simple interprocedural optimization technique for dynamically typed languages. In *POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 409–420, 1994.

[BE93] Mark R. Brown and John R. Ellis. Bridges: Tools to extend the Vesta configuration management system. Technical Report 108, Digital Equipment Corp. Systems Research Center, June 1993.

[Blu95] Matthias Blume. Standard ML of New Jersey compilation manager. Manual accompanying SML/NJ software, 1995.

[BR88] Alan Bawden and Jonathan Rees. Syntactic Closures. In *1988 ACM Conference on Lisp and Functional Programming*, pages 86–95, 1988.

[Bro74] P. J. Brown. *Macro Processors and Techniques for Portable Software*. John Wiley & Sons, London, New York, Sydney, Toronto, 1974.

[Bro93] Andrei Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 266–277, 1997.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Systems Research Center, Digital Equipment Corp., Palo Alto, CA, November 1989.

[Ce91] William Clinger and Jonathan Rees (editors). Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[CHT91] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[CL93] Sheng-Yang Chiu and Roy Levin. The Vesta repository: A file system extension for software development. Technical Report 106, Digital Equipment Corp. Systems Research Center, June 1993.

[Cle94] Geoffrey M. Clemm. The Odin System — Reference Manual, 1994.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachussetts London, England, 1990.

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[Cor93] Intel Corporation. *Pentium Processor User's Manual, Volume 1: Pentium Processor Data Book*. Intel Literature Sales, Mt. Prospect, Illinois, 1993.

[CR91] William Clinger and Jonathan Rees. Macros That Work. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[DH88] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[Dig92] Digital Equipment Corp., Maynard, MA. *DECchip(tm) 21064-AA Microprocessor Hardware Reference Manual*, first edition, October 1992.

[DuB96] Paul DuBois. *Software Portability with imake, 2nd Edition*. O'Reilly and Associates, Sebastopol, CA, 2nd edition, September 1996.

[Dyb92] R. Kent Dybvig. Writing Hygienic Macros in Scheme with Syntax-Case. Technical Report 356, Indiana University Computer Science Department, June 1992.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997. To appear.

[Fel79] S. I. Feldman. Make – a program for maintaining computer programs. In *Unix Programmer's Manual, Seventh Edition, Volume 2A*. Bell Laboratories, 1979.

[Fer95a] Maria F. Fernandez. *A Retargetable, Optimizing Linker*. PhD thesis, Princeton University, Princeton, NJ, 1995.

[Fer95b] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proc. ACM SIGPLAN '95 Conf. on Programming Langauge Design and Implementation*, volume 30, pages 103–115, 1995.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *1993 Conference on Programming Language Design and Implementation.*, pages 21–25, June 1993.

[GA95] Marcelo J. R. Gonçalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *Proc. Seventh Int'l Conf. on Functional Programming and Computer Architecture*, pages 293–305. ACM Press, 1995.

[GG93] Florent Guillaume and Lal George. *ML-Burg Documentation*. AT&T Bell Laboratories, 1993. distributed with SML/NJ software.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica and verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[Gou94] Jean Goubault. Generalized boxing, congruences and partial inlining. In *Static Analysis Symposium '94*, number 864 in Lecture Notes in Computer Science, pages 147–161. Springer, 1994.

[Gri72] Ralph E. Griswold. *The Macro Implementation of SNOBOL4*. W. H. Freeman and Company, San Francisco, 1972.

[Gun96] Carl A. Gunter. Abstracting dependencies between software configuration items. In *Proceedings of the Fourth Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 167–178. ACM Press, October 1996.

[Han91] Chris Hanson. A Syntactic Closures Macro Facility. *LISP Pointers*, IV(4):9–16, December 1991.

[HL93] Christine B. Hanna and Roy Levin. The Vesta language for configuration management. Technical Report 107, Digital Equipment Corp. Systems Research Center, June 1993.

[HLPR94a] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. A Compilation Manager for Standard ML of New Jersey. In *1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 136–147, 1994.

[HLPR94b] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU-CS-94-116, Department of Computer Science, Carnegie-Mellon University, February 1994.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[HS97] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-FOX-97-01, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, June 1997.

[IEE90] IEEE Standard 1178-1990: IEEE Standard for the Scheme Programming Language, 1990.

[JPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *1996 SIGPLAN International Conference on Functional Programming*, pages 3–12, May 1996.

[JW78] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report, Second Ed.* New York, 1978.

[JW96] Suresh Jagannathan and Andrew Wright. Flow-directed Inlining. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, May 1996.

[Kep91] David Keppel. A portable interface for on-the-fly instruction space modification. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* in *SIGPLAN Notices*, 26(4):86–95, April 1991.

[KKR+86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, 21(7):219–33, July 1986.

[KN93] Eleftherios Koutsofios and Stephen C. North. Drawing graphs with *dot*, October 1993.

[Koh86] Eugene Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, August 1986.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988.

[KTU94] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

[LL96] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, pages 137–148, May 1996.

[LM93] Roy Levin and Paul R. McJones. The Vesta approach to precise configuration of large software systems. Technical Report 105, Digital Equipment Corp. Systems Research Center, June 1993.

[LS83a] Butler W. Lampson and Eric E. Schmidt. Organizing software in a distributed environment. In *ACM SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1–13, June 1983.

[LS83b] Butler W. Lampson and Eric E. Schmidt. Practical use of a polymorphic applicative language. In *Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 237–255, January 1983.

[Mac90] David B. MacQueen. A higher-order type system for functional programming. In *Research Topics in Functional Programming*, pages 353–68, Reading, MA, 1990. Addison-Wesley.

[Mai90] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *17th Annual ACM Symp. on Principles of Prog. Languages*, pages 382–401, New York, Jan 1990. ACM Press.

[MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox PARC, 1979.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991.

[MT94] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Proc. European Symposium on Programming (ESOP'94)*, pages 409–423, April 1994.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Oho92] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 154–165. ACM Press, Jan 1992.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, New York, 1987.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[Ree93] Jonathan Rees. The Scheme of Things: Implementing Lexically Scoped Macros. *LISP Pointers*, VI(1):33–37, January-March 1993.

[Rei94] Mark B. Reinhold. Cache performance of garbage-collected programs. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, volume 29, pages 206–217. ACM Press, June 1994.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, volume 26, pages 293–305. ACM Press, 1991.

[Rep93] John H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993.

[SA94] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 150–161, New York, 1994. ACM Press.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc 1995 ACM Conf. on Programming Language Design and Implementation*, volume 30, pages 116–129. ACM Press, 1995.

[Sch77] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

[SGN88] Robert W. Scheifler, James Gettys, and R. Newman. *X Window System: C Library and Protocol Reference*. Digital Press, Bedford, MA, 1988.

[Sha97a] Zhong Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 85–98, New York, 1997. ACM Press.

[Sha97b] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, June 1997.

[Sha97c] Zhong Shao. Typed cross-module compilation. Technical Report YALEU/DCS/TR-1126, Department of Computer Science, Yale University, July 1997.

[Shi94] Olin Shivers. A Scheme shell. Technical Report TR-635, Laboratory for Computer Science, MIT, 1994.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Boston, 1992.

[Ste78] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[SZH85] Daniel C. Swinehart, Polle T. Zellweger, and Robert B. Hagmann. The structure of cedar. In *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, volume 20, pages 230–245, 1985.

[TA90] David R. Tarditi and Andrew W. Appel. ML-Yacc, version 2.0. Distributed with Standard ML of New Jersey, April 1990.

[Ten81] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Tic85] Walter F. Tichy. RCS—A System for Version Control. *Software—Practice & Experience*, 15(7):637–654, July 1985.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, New York, 1996. ACM Press.

[Tol94] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, pages 1–11, New York, 1994. ACM Press.

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

[Wir82] Niklaus Wirth. *Programming in MODULA-2*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, second edition, 1982.

[Wir88a] N. Wirth. The programming language Oberon. *Software—Practice and Experience*, 18(7), July 1988.

[Wir88b] Niklaus Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7), July 1988.