Hierarchical Modularity

MATTHIAS BLUME
Princeton University
and
ANDREW W. APPEL
Princeton University

To cope with the complexity of very large systems, it is not sufficient to divide them into simple pieces because the pieces themselves will either be too numerous or too large. A hierarchical modular structure is the natural solution. In this article we explain how that approach can be applied to software. Our compilation manager provides a language for specifying where individual modules fit into a hierarchy and how they are related semantically. We pay particular attention to the structure of the global name space of program identifiers that are used for module linkage because any potential for name clashes between otherwise unrelated parts of a program can negatively affect modularity. We discuss the theoretical issues in building software hierarchically, and we describe our implementation of CM, the compilation manager for Standard ML of New Jersey.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*Modules and interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules; packages*

General Terms: Design, Languages, Management

Additional Key Words and Phrases: Compilation management, linking, modularity, modules, name visibility, program structure

1. INTRODUCTION

Imagine working on some large software project that consists of a “main” module along with various other parts of its implementation (see Figure 1). In the course of development we want to be able to add, refine, and replace components without causing too much disturbance to the rest of the system. In particular, a modification to one module should not require other modifications in parts that are not conceptually related. The following wish list illustrates the array of problems (some of which are really the same problem viewed from different angles):

Structural Refinement. The same principles that are used for structuring entire programs should also be applicable to parts of the program. For example, the

This work was supported in part by NSF Grant CCR-9625413.
Authors' addresses: M. Blume, Research Institute for Mathematical Sciences, Kyoto University, Kyoto, 606-01 Japan; email: blume@kurims.kyoto-u.ac.jp; A. W. Appel, Department of Computer Sciences, Princeton University, Princeton, NJ 08540; email: appel@cs.princeton.edu

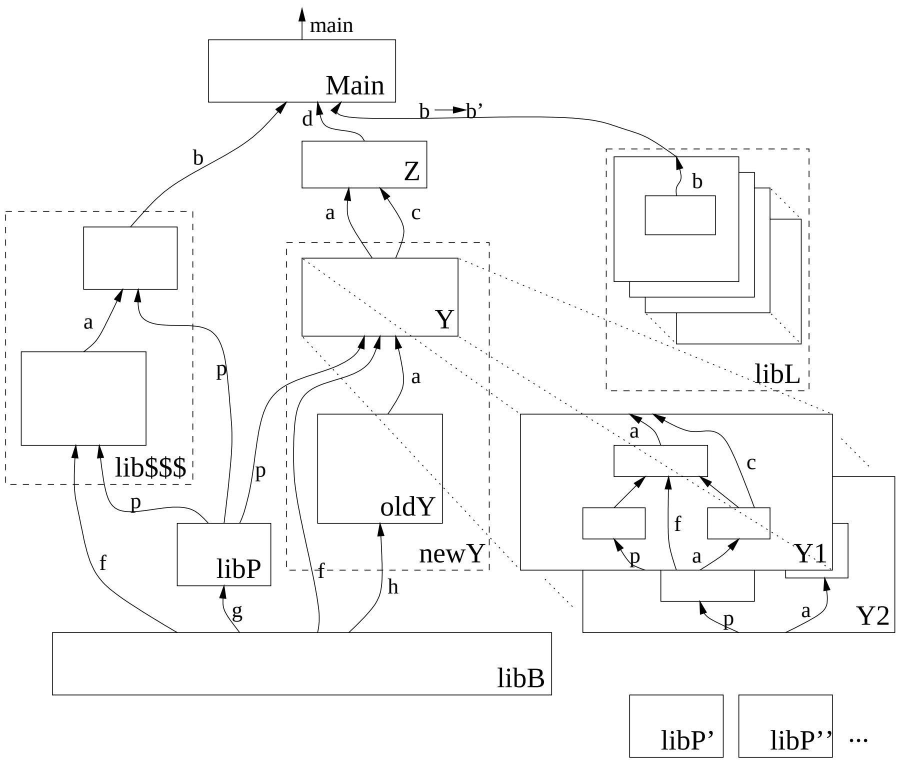


Fig. 1. Schematic view of a software project.

subsystem `newY` consists of two parts (`Y` and `oldY`), but to the outside it presents itself as only one module.

Grouping. Conversely, by grouping existing modules together one should be able to form larger modules. Examples for such "library" modules are `lib$$$` or `libL`. (Although "structural refinement" and "grouping" really refer to the same underlying idea, we prefer to use the former in the case of top-down development and the latter when constructing programs in a "bottom-up" fashion from existing parts.)

Sharing. In addition to its components `Y` and `oldY`, `newY` makes use of `libP`, a library of "primitives," and `libB`, the "base" API. But both `libP` and `libB` are also used by other modules. Therefore, groups of modules produced by structural refinement should not have to be disjoint.

Libraries without Source Code. Suppose we purchase a library such as `lib$$$` from a software vendor. It must be possible to use it without having to inspect, recompile, or modify its constituent parts because the vendor may not want to provide source code.

Libraries with Imports. Libraries themselves usually rely on other libraries. In our example, `lib$$$` imports from `libP` and `libB`. Provided that the exported interfaces match the import interface of `lib$$$`, it should be possible to use `lib$$$` with any user-supplied version of `libP` and `libB`.

Extending Interfaces. Module `newY` was developed by extending an existing older version called `oldY`. We want to be able to directly reuse the original code of `oldY` by adding wrapping code in `Y`. `Y` may export an interface that is richer (i.e., contains more definitions) than the one exported by `oldY`.

Minimizing Dependencies. Moving from `oldY` to `Y` only affects the components `Z` and—perhaps indirectly—`Main`. It should not be necessary to recompile, much less modify, any other module.

Cutoff Recompilation. Certain modifications to the export interface of `Y` will affect the export interface of `Z` and therefore also cause `Main` to be recompiled; others do not. A compilation unit should have to be recompiled only if its own source code or its imports have actually changed.

Names in Separate Modules. Internally, the implementation of the library `lib$$$` uses the name a for communication between its own modules. Since a does not appear in the library's interface, the library's use of a should never clash with any other use of a—regardless of whether or not a is the name of a variable, a type, a module, or even a name space.

Redefinitions. Module `oldY` exports a to `Y`, which uses it to form a different a exported to `Z`. This can only be done if there are ways to avoid a clash between these two uses of a.

Name Clashes within One Module. Module `Main` needs to refer to b from `lib$$$` as well as to b exported by `libL`. We have no authority over the names that `lib$$$` exports. Perhaps the same is true for `libL`, and even if it were not, we might not want to modify `libL` for a variety of reasons. To be able to use either b simultaneously, it should be possible to "rename" one of the definitions at the time of import.

Multiple Alternative Module Implementations. Experimenting with an implementation can mean trying out various different approaches for implementing the same module. In our example, we provide more than one implementation for `Y`. Replacing one of them (`Y1`) with another (`Y2`) should not entail other modifications or recompilations unless there are changes in the export interface. The versions can safely differ in how they take advantage of imports that are available to them: `Y1` imports f while `Y2` does not.

Profiling, etc. Sometimes, we want to have several versions of the same object (e.g., `libP`) that are all derived from the same source code. This enables us to transparently and selectively provide profiling or debugging support. It is important that if all versions have the same interface there will be no need to modify or recompile any of `libP`'s clients. (One of the clients is `lib$$$`, for which we have no access to source code.)

Location. Typically, components of a program are stored in some sort of file system that is provided by the underlying operating system. During development, files may frequently change their names and locations. For example, it could be that `oldY` used to be called `Y`. The name change itself should not make it necessary to recompile any of the modules. But this requires that a file name like `Y` is not somehow hard-wired into the result of compiling client modules like `Z` (or, even worse, the source code of `Z`).

Our compilation manager software, "CM," is able to gracefully handle all of the situations that we have listed. The group model provides support for structural refinement and grouping. It also avoids name clashes and allows for redefinitions when necessary. Dependency graphs are DAGs and, therefore, permit sharing. Libraries without source code are supported by the *stable library* mechanism. Libraries can import from other libraries, and type-safe linking will ensure consistency. Interfaces can be extended using nested groups. A built-in analyzer automatically calculates the dependencies between compilation modules; cutoff recompilation helps minimize actual recompilation work. Finally, a separation of programming- and configuration-languages makes it possible to seamlessly relocate or even replace compilation units as well as entire libraries.

CM is not the first solution to the list of problems we showed, much less the only one. For example, Feldman's **make** [Feldman 1979] and its modern derivatives—together with tools that explicitly manipulate object files, library archives, and environments (i.e., symbol tables)—let their expert users achieve many if not all of the goals that we have outlined. One could also put these manipulations under control of a sophisticated system like CAPITL [Adams and Solomon 1993]. CM uses the same techniques internally, but provides a simple declarative specification language as an interface to these operations.

On the other hand, neither the operation of **make** nor that of CAPITL in itself is based on intermodule semantics of their user's programming language the same way CM's operation is based on ML's intermodule semantics. Instead of requiring dependencies to be spelled out in full, CAPITL improves on **make** by using a logical inference engine to calculate dependencies from a declarative database. It may be possible to emulate CM's behavior using CAPITL, but doing so would require to implement CM's syntactic and semantic analyses as a CAPITL specification, or, to put it differently, to implement CM on top of CAPITL. Also, CAPITL's logic inferencer has performance problems due to an inefficient search strategy. On the upside, the use of a persistent database avoids the struggle with native file system semantics. In this area CM could take some lessons.

Explicit symbol table manipulations provide great power to work around any conceivable problem related to name resolution. For example, a solution based on ad hoc link-time renaming of identifiers has been reported for Vulcan, an experimental Modula-2+ programming language environment that was developed as part of Vesta [Brown and Ellis 1993]. But such power comes at a hefty price: a program can no longer be understood solely in terms of the programming language that it is written in. Instead, one must take into account the semantics of the tools that happen to be used to manipulate the various intermediate objects that emerge in the process of its compilation. These semantics are often system- or version-dependent, are rarely documented with the same rigor that should be applied to programming language definitions, and in their power have far-reaching impact on the meaning of the overall program. This situation is especially dissatisfying if the original programming language was aiming at semantic clarity and well-foundedness.

To avoid these problems we control the full power of explicit symbol table manipulations by providing what amounts to a programming language extension: Standard ML becomes SML+CM. Using the new "+CM" part, the programmer can express any desired grouping of modular components directly without having to know about the operations that CM will invoke on his or her behalf. We were pleased to find that a very modest extension to SML was sufficient to achieve the desired effect. The result does not have the feel of a completely new programming language. We also believe that programming with CM is easier than programming with **make** and associated tools.

Finally, we can explain CM's internals in terms of low-level operations on environments. Therefore, our work is clearly not restricted to ML-like languages but can be carried over to others fairly directly.

2. ASPECTS OF MODULARITY AND COMPILATION MANAGEMENT

2.1 Separate Compilation

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 [Parnas 1972]. But the so-established physical boundaries—when placed cleverly—can have a profoundly positive impact on overall program structure. For a long time separate compilation has been a key element of modern software engineering [Cardelli 1997].

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.

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 intermodule dependencies. Type-safe linking is an extension to ordinary linking (i.e., name/address resolution) that notifies the programmer of inconsistency between the type at which one module exports an identifier and the type at which another module imports it. This link-time notification is valuable and prevents many kinds of run-time bugs. But link-time notification is still later than necessary: 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.

2.2 Hierarchical Modularity

Modularization and separate compilation are not synonymous, even though in most cases a compilation unit boundary is also a module boundary. Modern programming languages tend to adequately support modularization within compilation units.

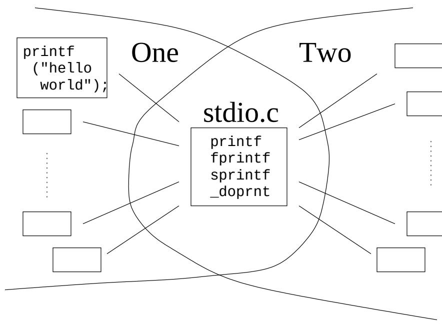


Fig. 2. Sharing one source file. Only a few programming languages let the programmer control the interface of one source file to guarantee that it can be shared between several clients. In C one would declare `_doprnt` as *static*, because local functions should not be exposed. But the C programmer has no way to prohibit `stdio.c` from referring to symbols in system One; these references might prevent it from being used in system Two.

This article is concerned with those modules that consist of at least one and potentially span more than one compilation unit.

Modular structure, like the one that we assumed for the introductory example (Figure 1), arises naturally in many engineering tasks. To cope with complexity, large projects are routinely divided into parts. These parts interoperate according to interface specifications. Interfaces hide much of the internal complexity of each part's implementation from other parts.

In a good modularization, no module should be overly complex, and there should not be too many modules. This can only be accomplished with a hierarchical structure, with each module constructed from submodules. Of course, the subdivision of a part must not be manifest in its interface. Seen from the outside, each part acts as a single module while internally it can be further structured into smaller subcomponents. The result is a *hierarchy of modules.*

Hierarchical modularity is not a new concept—it can be found in most very large engineering projects. But at least in the case of software systems it has been applied only informally, since programming languages and compilation management have not provided active support [Jacobson 1987].

There are many real-world examples that show the need for hierarchical modularity. Figure 2 depicts a situation where one module (`stdio`) is shared by two different projects (One and Two). We would like to administer its interface so that the compiler can guarantee the absence of unwanted dependencies on either One or Two. Even that is not fully supported by most programming languages. But for large-scale programming we need to take this further. It is important to allow `stdio` itself to be a group of source files, with a summary interface that controls such a group's interface to its various clients (see Figure 3).

Figure 4 illustrates how CM would be used to deal with this situation. The library `stdio` is implemented as a CM group, and clients like One and Two must explicitly list that group as an import to use its services. No client can depend on implementation details that are not explicitly advertised by the group's export interface. Moreover, the subgroup relationship is unidirectional. The `stdio` group

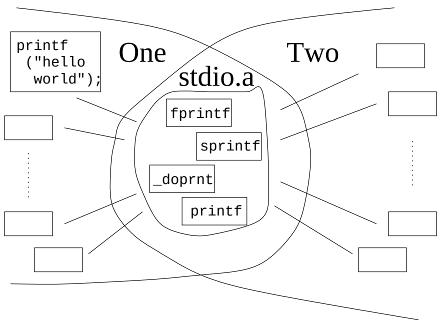


Fig. 3. Sharing groups of source files. Often we want a group of sources to act like one module. An explicit interface placed on `stdio.a` as a whole (as opposed to its constituent components) guarantees the absence of undesired dependencies on either One or Two and does not expose local objects like `_doprnt`. The C language has no mechanisms for expressing either of these requirements.

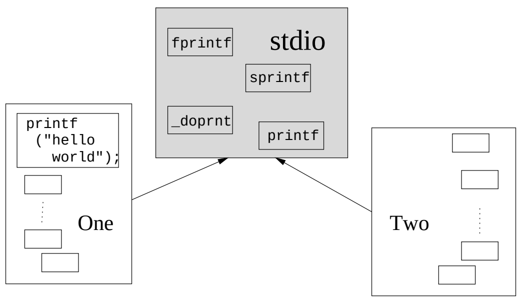


Fig. 4. Controlling sharing using groups. CM can structure the project shown in Figure 3 into a collection of groups. The export interface on group `stdio` will make sure that implementation details, such as the local function `_doprnt`, are not exposed to its clients, One and Two. At the same time, since `stdio` is specific in not listing either One or Two in its imports, it cannot depend on either one of them.

does not (and cannot) mention either One or Two as part of *its* imports. Even if the design of a group caters to specific needs of selected clients, subsequent modification to any one client cannot cause incompatibilities with others if they were compatible before.

2.3 Semantics versus Dependencies

Compilation management deals with two separate problems. On the one hand, it must provide a way of describing semantic relationships between separate modules: how names are resolved across module boundaries and how hierarchies of modules are formed. On the other hand, it has to determine intermodule compilation dependencies: which subset of sources needs to be compiled or recompiled in what order. Essentially, this part is an optimization of the compilation process.

There are two ways of going about this: one can start with dependency information and derive the resulting semantics of the program, or one can start with a semantic description and try to infer dependency information.

For an automatic tool, the first route is more convenient because the tool does not really care about semantics; it just needs to follow the dependencies, and that is easy if they are given. Moreover, as exemplified by **make**, this approach works in a generic way with almost arbitrary programming languages and tools. The drawback is that the programmer who wants to know the meaning of a program must read the descriptions of compilation dependencies annotated with associated "build" operations (e.g., Unix commands), understand their semantics as well as the semantics of their interactions, and infer the program's overall meaning—a challenging task in many cases.

CM takes the other approach. It first provides a language for describing the semantics of a system that is divided into separate modules. This is the *group* model described in Section 4. It specifies the hierarchy of modules and the scoping rules to be used for references that span module boundaries. Once the specification of semantic aspects is given, CM can then calculate the resulting compilation dependencies from there.

2.4 Names and Name Spaces

When a module exports one of its definitions, then this definition becomes known to other modules by some name x. To be able to refer to this x, a second, importing module cannot use x for some other purpose at the same time. This is not a problem because the programmer of the second module certainly has to be aware of x anyway, so he or she can easily avoid a name conflict. Therefore, coordinating the development of conceptually related components is relatively easy. Name clashes will be detected early in the development cycle, and the occasional conflict can be resolved where it occurs. This does not require any modifications to unrelated components.

It is much more troubling if the names in conceptually unrelated modules must also be kept apart. If the second module does not import x, then it may easily happen that its programmer does not even know about the first module's existence. In fact, not having to know about unrelated modules is clearly desirable for a programmer who participates in a large project that has many modules. Preventing clashes means that some amount of coordination is necessary, making development of otherwise independent modules more difficult. We will discuss some examples for this undesirable effect later (see Figures 5, 12, and 13).

Library designers attack this problem by using naming conventions, for example, by adding some prefix to all exported identifiers. But in reality, such preventive measures are applied not nearly universally, and even if they were—there is still no guarantee that the prefixes themselves will not clash. As a result, it sometimes can be difficult to simultaneously use the libraries sold by two different vendors, which is especially bothersome if they are not conceptually connected and are used for different purposes in unrelated parts of the program. Even when the entire source code for all libraries is available, it can be a challenging problem to resolve all naming conflicts [Ford et al. 1997, Section 4.7.2].

Most modern languages use block structure to provide many different localized scopes for nonexternal names. Scoping permits the same name to be used multiple times independently. To solve the problem with external name clashes, we simply extend this idea to the global name space. With block structure at the group level, we can place unrelated libraries into different scopes and never again need to worry about name clashes between them.

Related modules should be within the same scope because they need to have access to each other's exported identifiers. Unrelated modules should be in different scopes to avoid potential name clashes. But related modules will be within the same enclosing group, while unrelated modules will be in different groups. Therefore, to fully support hierarchical modularity, *external scoping must follow the program's group structure*. Managing the global name space according to this rule is CM's most important aspect.

2.5 Programming Language versus Configuration Language

Do we really need another language layer? This question must be asked considering that many modern languages already have a module layer intended for dealing with large-scale programming.

Name clashes have been with us ever since the first programming language that used names, i.e., Church's λ-calculus [Church 1941; Barendregt 1981], and just as the λ-calculus uses α-conversion (renaming of bound variables) to work around name conflicts, any programming language or environment should provide some mechanism for dealing with this problem. If we cannot avoid name conflicts, then the only hope is to restrict them in such a way that they do not compromise modularity.

With CM there are two places where name conflicts can still occur. The first is the programming language itself, but these conflicts are guaranteed to be "modular," i.e., of a benign kind. We will later explain the notion of modular conflicts and show how to deal with them. The second kind of name clash occurs in CM's configuration language. The names in that language are actually file names of the underlying operating system. Therefore, solving these conflicts amounts to moving files or directories around. Such operations do not require knowledge about the contents of those files and in our implementation do not incur any recompilations.

2.6 The ML Module System versus CM

CM's merits in the area of handling files or managing recompilation are clear. But with its configuration language, albeit simple, it also provides expressive power that goes beyond that of the underlying ML language alone. The following simple example (see Figure 5) makes this clear:

Consider a project involving three programmers, Alice, Bob, and Chet. Alice, who is in charge of the project, knows both Bob and Chet, but Bob and Chet are each unaware of the other's existence. Alice asks Bob to write module X; Chet is assigned module Y. Both X and Y are large enough so that Bob and Chet independently subdivide their tasks. As a result, Bob delivers an implementation of `structure X` together with a separate auxiliary module implementing some `structure Y` that he needed in order to implement `X`. And as it so happens, Chet provides an implementation of his `structure Y` together with an auxiliary module that implements a helper `structure X`.

Meanwhile, Alice has written her main module which uses both Bob's `structure X` and Chet's `structure Y`. She now faces the problem of putting the source files into an order that lets him compile the final program. Unfortunately, there is no

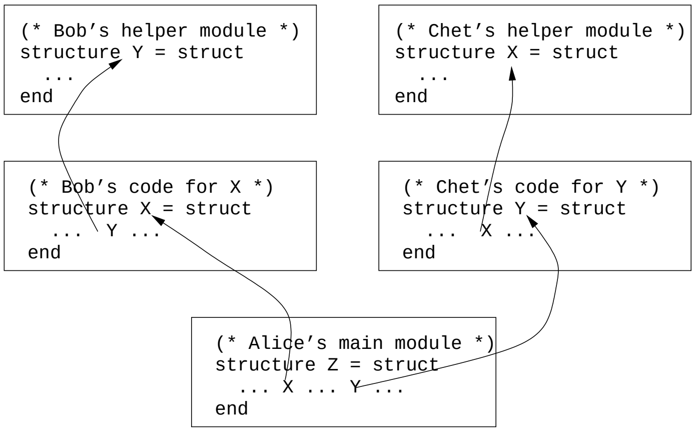


Fig. 5. An ordering puzzle. In plain Standard ML there is no ordering for the compilation of the five source files that would give the correct result. Moreover, the glue code that Alice needs to be able to make the program work cannot be written without inspecting internal internal details (in this case: the names of the helper modules) of Bob's and Chet's solutions.

possible ordering of ML source files that has the desired effect because Bob's and Chet's helper modules always get into the way and cause name conflicts.

The only possible solution for Alice (short of modifying Bob and Chet's code) is to insert glue modules that locally rename one structure so that it does not conflict with its namesake in the other programmer's code. To insert this glue code, Alice needs to know about the internals of Bob's and Chet's implementations of X and Y—something she wanted to avoid.

The analysis of Bob's and Chet's source code and the subsequent insertion of glue modules (or rather, of renaming operations) could be mechanized, but the ML module system itself does not do that—CM does. Using CM, Bob and Chet each would deliver a group. These groups would export precisely what Alice asked for (and nothing else). Alice could then put together the final program without even having to know anything about the internal structure of her colleagues' solutions.

The scenario described above may seem simplistic. The danger of name conflicts in real programs may be relatively low. However, in the end even a small danger is one more unnecessary detail that the programmer will have to keep in mind. CM's group model is really a simple and natural idea, but as such it is very effective at solving the problem.

2.7 Other Programming Languages

The modules systems of some languages, for example "name spaces" in C++ [Ellis and Stroustrup 1990] and "packages" in Ada [DoD 1980] or Java [Arnold and Gosling 1996], provide means for spreading the implementation of a named module across several compilation units. In the case of our previous example (see Figure 5), Bob and Chet could each have implemented their respective helper modules without "leaving" their assigned main module: Bob would implement `X` and helper module `X.Y` while Chet uses the names `Y` and `Y.X`.

Does this mean that CM is nothing more than a fix for the badly designed module system of just one particular language? To see why we answer this question negatively, one should first notice that the same trick actually works even in ML: Alice would have told Bob to implement `X` using helper modules whose names must all start with `X_`, while Chet could only use auxiliary global names that start with `Y_`. Thus, we cannot say that ML's module system is somehow fundamentally weaker.

But more importantly, both the ML "low-tech" approach and the solution in C++ or Ada require prior agreement between programmers. Both of them will break down when such prior agreement cannot be obtained—a situation that we encounter every time a module that already existed is to be integrated into a new project. In other words, programming language module systems tend to give good support for top-down program design but fail in the case of bottom-up development.

When it comes to module systems, the approaches taken by most modern languages all look very similar to each other: some form of bundling of definitions is used, and the bundle itself appears under a new name—the module name. By definition, names in different modules cannot clash. However, the module itself has a name that is *outside* the module. Consequently, although much less likely, there is always the potential for top-level module names to cause the same problems they were supposed to solve by removing other names from the global name space.

SML structure names can clash; C++ name space names can interfere with one another; and Modula-3 [Cardelli et al. 1988] 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 that in their sources, it now becomes impossible to relink 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.

3. THE CM SOFTWARE

CM is our compilation manager for Standard ML of New Jersey (SML/NJ for short) [Appel and MacQueen 1991]. Standard ML [Milner et al. 1990; 1997] enjoys considerable popularity within the programming language research community but has yet to make an impact on "real-world programming." Still, even though CM's concrete realization is tightly integrated with ML, the ideas behind its design are much more broadly applicable to many languages. A "CM for C" or a "CM for Java" could be built along similar lines. Indeed, work is is progress for such a "CM for Java" [Bauer et al. 1999]. The main building blocks—tools for linking and for manipulating symbols tables—already exist and are waiting to be put together.

The purpose of CM is comparable but not equal to that of other compilation and configuration managers. Examples are **make** [Feldman 1979], Odin [Clemm 1994], the System Modelling language of Mesa and Cedar [Mitchell et al. 1979; Lampson and Schmidt 1983a; 1983b; Swinehart et al. 1985], and Vesta [Levin and McJones 1993; Hanna and Levin 1993; Chiu and Levin 1993]. *Group descriptions* play the same role for CM that system models play for Vesta and makefiles play for **make**.

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

Of course, these features come at a cost. CM is not a generic compilation management tool in the spirit of **make** but is geared specifically to deal with ML. CM's extensible toolbox can accomodate a variety of other language processors, but those are not fully integrated with the sophisticated management of SML code.

CM and SML are tightly coupled at the level of source files and at the level of object files. At the source level, CM implements automatic dependency analysis [Blume 1999] and, thus, must understand the programming language it deals with. At the object level, CM must handle SML/NJ's own formats for files, symbol tables, and executables. Neither of these two points has anything to do with the concept of hierarchical modularity itself. But being based on SML/NJ made the implementation of CM easier, while automatic dependency analysis made CM more convenient to use.

CM is predated by the "Incremental Recompilation Manager" [Harper et al. 1994a], which later became known as SC [Harper et al. 1994b]. We developed CM based on our study of SC, and the availability of SC's source code was of great help when we started. IRM and SC were first to take advantage of SML/NJ's visible compiler interface with its support for *cutoff recompilation*. CM improves on SC by using a group model that implements hierarchical modularity, offers full support for libraries, and permits the use of explicit export interfaces.

The CM software has become an integral part of the SML/NJ project [Blume 1995]. It has been used to good advantage by the SML/NJ compiler development team, by many of SML/NJ's users, and even for teaching [Appel 1998].

3.1 Cutoff Recompilation

Aside from name space management, scoping, and grouping, 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 exported by `a.sml.bin` to be able to produce the binfile `b.sml.bin`. Therefore, whenever the static environment exported by `a.sml.bin` changes, `b.sml` must also be recompiled.

The approach taken by **make** safely approximates this by recompiling `b.sml` every time `a.sml` gets recompiled. However, this can be overly pessimistic. Deep dependency graphs, which occur frequently in SML programs, lead to many unnecessary recompilations. As long as the static environment in `a.sml.bin` stays the same, recompiling `a.sml` does not require subsequent recompilation of `b.sml`. Cutoff recompilation [Adams et al. 1994]—the strategy used by CM—takes advantage of this observation. For efficiency, instead of comparing entire static environments, CM only compares relatively small fingerprints [Gunter 1996]. The fingerprinting method is based on CRC polynomials [Broder 1993].

Fingerprinting means that there exists a possibility, although extremely unlikely, that two interfaces are erroneously found to match when they really do not match.

With our 128-bit fingerprints the probability for such a mistake is somewhere on the order of 2^{-100} which is much smaller than the probability of a hardware failure. Still, it may be considered an imperfection, and we have chosen to live with it.

3.2 Groups

A group consists of a list of "exports" and a list of "members." Exports are simply ML symbols; a member can either be an ML source file or another group. The *group description file* has the following general format:

Group *export-symbol …* **is** *member …*

For convenience, the programmer can choose to leave the list of export symbols empty. In this case CM will provide a suitable default. A "library" is a special kind of group whose constituent modules become part of the main program only if the main program directly or indirectly refers to them. Library descriptions look like group descriptions with the keyword **Group** replaced by **Library**. Moreover, the export list of a library cannot be left empty.

Probably the simplest example for a group is one that has an empty export list and no subgroups (i.e., no members that are groups):

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

Once the group description file is in place, CM can then analyze the dependencies among components of the system, determine a feasible ordering of (re-)compilation steps, and carry them out as necessary.

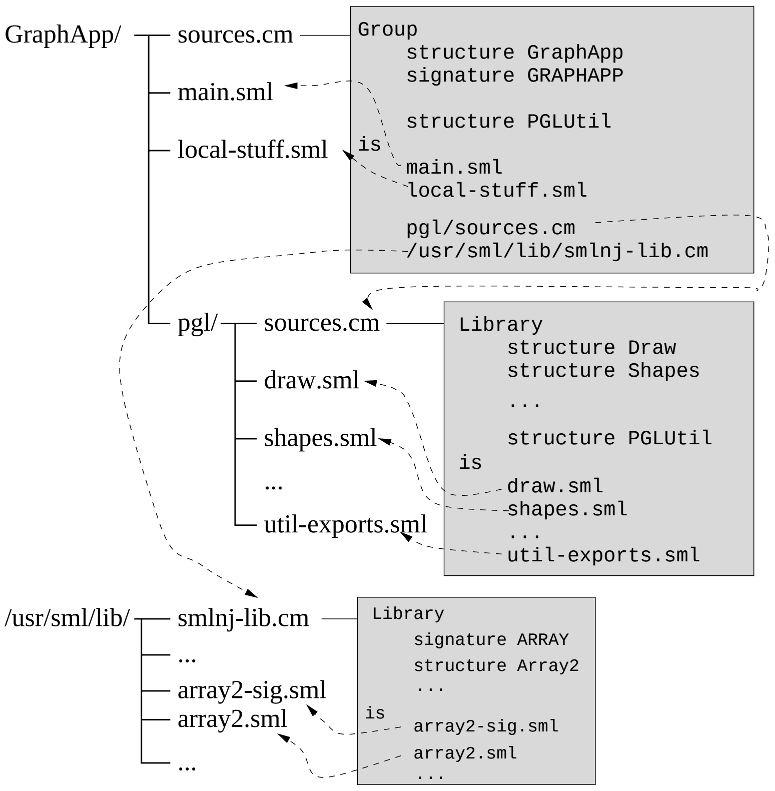
3.3 Hierarchies

As we have explained, large programs should be broken into multiple groups, and the groups should then be arranged into a hierarchy. This makes it easier to manage large systems because related sources are kept together, while unrelated sources are kept apart. 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. In languages that permit cycles between modules, we believe that it would be reasonable to allow cycles *within the same group,* but we have not investigated this idea in depth.

Groups and Subgroups. As shown in Figure 6, 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 subgroup's definition.



```
Group
    structure GraphApp
    signature GRAPHAPP

    structure PGLUtil
is
    main.sml
    local-stuff.sml

    pgl/sources.cm
    /usr/sml/lib/smlnj-lib.cm
```

```
Library
    structure Draw
    structure Shapes
    ...
    structure PGLUtil
is
    draw.sml
    shapes.sml
    ...
    util-exports.sml
```

```
Library
    signature ARRAY
    structure Array2
    ...
is
    array2-sig.sml
    array2.sml
    ...
```

Fig. 6. 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 path names in description files refer to files in the directory that contains the description.)

`sources.cm`	`util/sources.cm`

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

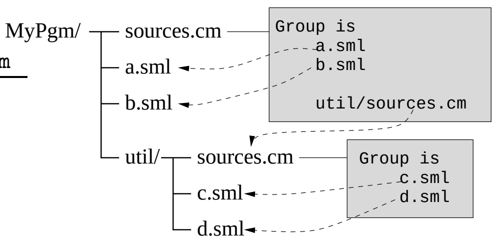


Fig. 7. A simple group hierarchy. Suppose `a.sml` and `b.sml` are sources of a group that needs to refer to a subgroup containing `util/c.sml` and `util/d.sml`. In this case one could create two description files, and one of them will then refer to the other.

```
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
```

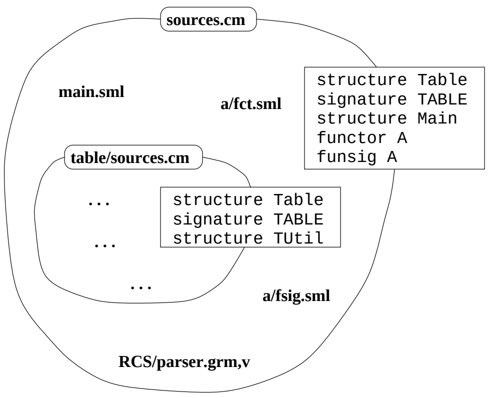


Fig. 8. A group with an export list. Notice how the outer group may export some (but not necessarily all) of the definitions that are exported by the inner group.

Usually, CM will automatically identify description files in its member list by their names. Those that end in `.cm` are treated as names of other group description files. Relative file names are resolved with respect to the directory that contains the description file (see Figure 7).

Auxiliary Tools. Some members are neither SML sources nor group descriptions. They require special processing before CM's analyzer can understand them. This is done by "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`, and then the ML-Yacc tool will take over to produce `parser.grm.sig` and `parser.grm.sml`, which are finally processed by CM directly.

The built-in toolbox of CM is extensible. It allows for seamless addition of new tools to the existing set of predefined ones by writing a few lines of Standard ML.

Export Lists. Every group has an export list. The list, which consists of ML symbols, restricts ("filters") the set of definitions that are exported. If the export list for a group is given explicitly, then the group will export definitions for precisely the symbols listed, regardless of whether they are defined in the group itself or in one of its subgroups. If the programmer does not provide an explicit export list, then CM supplies a default.

Export lists 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. Section 4 discusses how such summary interfaces can improve separation between software components.

The list of symbols that makes up an export list appears 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 list could look like the one in Figure 8.

Libraries. A library is a special kind of group. The differences between libraries and ordinary groups are small and mainly concern the notion of implicit export interfaces. First, a library itself cannot use an empty export list. The programmer must provide the list of exported symbols explicitly. With this choice we intend to enforce some discipline when creating libraries. And second, the library's exports will not be reexported by an importing group if that group uses an implicit export interface. The idea here is that an incidental member of a library will not accidentally pollute the export list of a client.

Preprocessor. At the time it reads a description file, CM applies a simple, C-like preprocessor that allows for conditional linking, such as

```
Group is
    a.sml
# if (SMLNJ_VERSION < 110)
#   error This version of SML/NJ is too old.
# elif (defined (OS_UNIX))
    b-unix.sml
# else
    b-nonunix.sml
# endif
```

3.4 The Role of Dependency Analysis

CM provides a language for specifying the semantic structure of large programs that consist of many separately compiled modules by arranging these modules into a hierarchy of *groups* and *libraries*. The hierarchy more or less directly reveals dependencies between groups, while dependencies between individual source files *within* each group are not given explicitly. Here CM's dependency analysis maintains the illusion of unordered source collections.

This balancing act—explicit dependencies between groups but implicit dependencies within groups—is important. One must impose certain restrictions on the source language to be able to make dependency analysis tractable [Blume 1999].[1] However, some of these restrictions should not be used indiscriminately for the entire program but only within groups. As we have seen, this is particularly true for the requirement of disjointness of exported names.

Dependency analysis within groups significantly simplifies the task of writing group descriptions. It 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 lan-

[1]As discussed in the paper we cite here, there are many possible ways to restrict the language. For CM we chose the following two: one cannot export definitions for the same name from two or more different sources of the same group, and the use of ML's **open** construct is prohibited at the top level.

guage, one can imagine adding dependency analysis using an auxiliary program that calculates and generates specifications. Examples are the **imake** and **makedepend** tools that generate input for **make** from C source code [DuBois 1996].

3.5 Caches, Files, 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.

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. CM not only stores compilation results in binfiles but also keeps them in main memory. This often reduces file system traffic in ongoing edit-compile-debug sessions because data is readily available in main memory and does not need to be reloaded from auxiliary storage.

Dependency analysis is much less expensive than compilation, but it still has its cost because the 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. 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. Sizes tend to be around 2–5% of that of their corresponding source files. (In our implementation it was convenient to create one dependency file for each source file.)

The existence of cached information often means that CM does not even need to consult the original source file. However, this is not always the case, since the result of compiling one source file does not only depend on that source but also on the environment in effect. This environment is the result of compiling the source's predecessors in the dependency graph. Modifications to any of them may require the source to be recompiled as well.

We can guarantee that a source does not need to be consulted if we ensure that all ancestors in the dependency graph stay fixed. This leads to CM's notion of *stable groups*, which are groups explicitly designated to remain unaltered for the foreseeable future. Examples of stable groups are central libraries that are installed and maintained by the system administrator. The process of stabilization creates a special version of a dependency file called the *stablefile*. It acts as an archive, summarizing all individual depedency files and binfiles.

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.

4. THE GROUP MODEL

Linking separately compiled modules associates the imports of each module with the corresponding exports of other modules. This association is mediated by a global name space in which linking takes place.

If the global name space is not sufficiently structured, then problems for modularity arise. The exact nature of these problems depends on the structure of the name

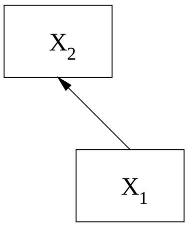


Fig. 9. Direct relation. Two program fragments X_1 and X_2 are directly related if X_2 explicitly refers to a definition that is exported from X_1.

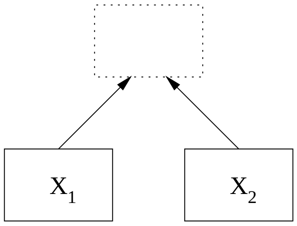


Fig. 10. Indirect relation. Two program fragments X_1 and X_2 are indirectly related if there is a third fragment that explicitly imports from both.

space. We will analyze two examples, C and ML. These cases are not strawman examples: all commonly used programming languages that the authors are aware of suffer from these or from similar problems.

In the following discussion we will first introduce the notion of *directly or indirectly related modules*. It will enable us to state the conditions on naming-related conflicts that we consider acceptable for modularity. These conditions are satisfied by CM's group model, as we will demonstrate when we formally develop a simple calculus for linking.

4.1 Relationships between Modules

Name conflicts are especially troublesome when the interfering parts of the program are unaware of each other's existence. It seems obvious that modules that are conceptually related will have to know about each other anyway. Therefore, we are interested in avoiding naming conflicts precisely in those cases where there is no such conceptual link. Of course, "conceptual link" is not precise language. We will strengthen this idea and define what we mean by *unrelated* modules.

Definition 1. Two groups are said to be 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 we call them unrelated (see Figures 9 and 10).

As shown in Figure 11, this relationship between modules is not transitive.

Definitions are said to *interfere* if they can cause name conflicts in parts of the program that are unrelated. Name conflicts between related modules are acceptable because—as we will see later—they can be resolved by simple local modifications that affect only the part of the program where they occur. We will call such conflicts *modular.*

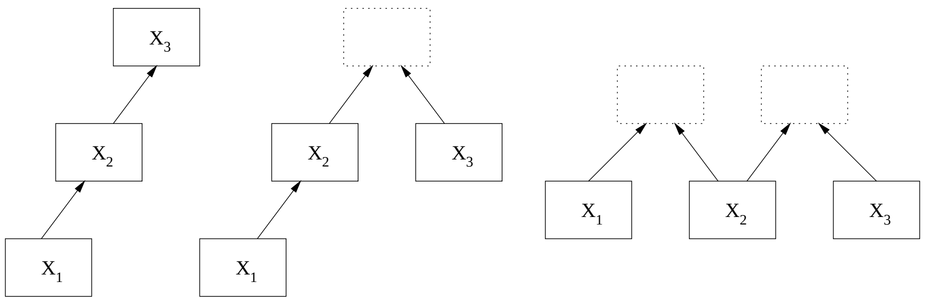


Fig. 11. Module relationship is not transitive. In all three examples, modules X_1 and X_3 are unrelated even though X_2 is related to each. This nontransitivity means that the potential for a name clash in one part of the system does not easily spread throughout the entire program.

Definition 2. A name conflict (and its associated restriction on name usage) is called modular if it affects only related parts of the program (i.e., parts that according to Definition 1 are either directly or indirectly related).

4.2 Definability and Availability: The Cases of C and ML

C [Kernighan and Ritchie 1988] is a language where the potential for naming conflicts causes restrictions on *definability* of names. While this is avoided in ML, there still is a potential for conflicts—causing restrictions on the *availability* of definitions.

Definability. A name n is *definable* at a given program p point if adding a definition for n at p will not cause a name conflict with another definition in the same program.

Availability. A definition d is *available* at a given program point p if the name defined by d can be used at p and refers to the meaning that was assigned by d.

A C program cannot use certain identifiers because they are potentially taken by libraries or other parts of the program, even if those are unrelated. Thus, these identifiers are no longer definable (see Figure 12). An ML program 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 (see Figure 13).

Examples and Discussion. Because of the global name space's lack of structure, definitions that are conceptually 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 an inelegant and clumsy solution to the more general problem. It cannot give guarantees of nonabuse, 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 that `printf`, `sprintf`, and

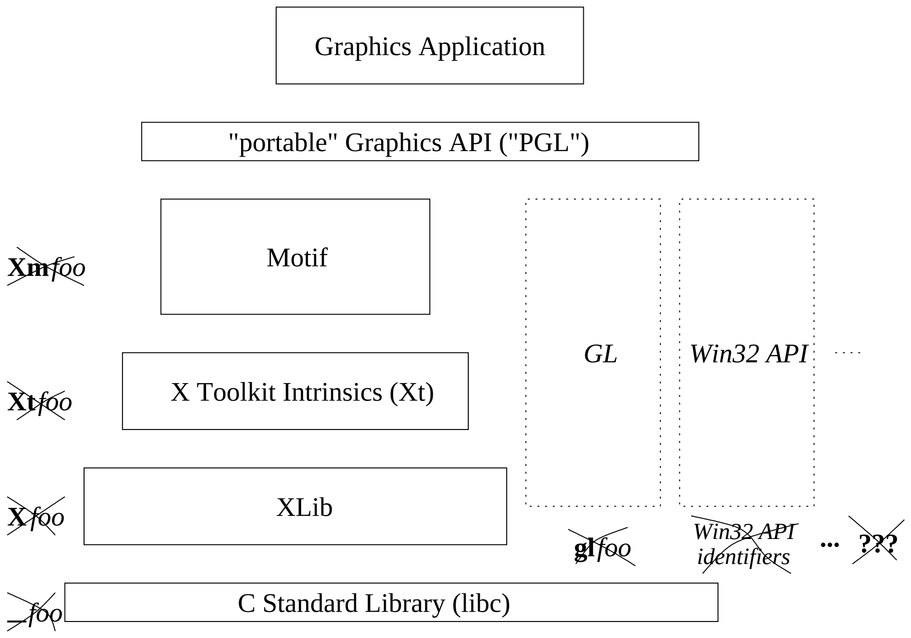

Graphics Application
"portable" Graphics API ("PGL")
Motif
X Toolkit Intrinsics (Xt)
XLib
C Standard Library (libc)
GL
Win32 API
~~Xmfoo~~
~~Xtfoo~~
~~Xfoo~~
~~_foo~~
~~glfoo~~
~~Win32 API identifiers~~
...
~~???~~

Fig. 12. Restrictions on definability inhibit modularity. Imagine a "portable graphics library" (PGL) that provides a uniform API implemented on top of either X/Motif, Silicon Graphics's GL, or Win32. The programmer of this graphics application in C, who ideally would like PGL 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.

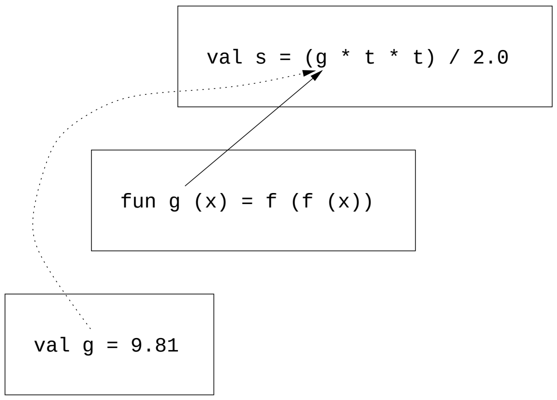


Fig. 13. Restrictions on availability inhibit modularity. In this SML program a value for `s` was meant to be calculated in terms of the gravitational constant `g` that was defined earlier. But a third, conceptually unrelated compilation unit redefines `g`, so the original definition becomes unavailable. 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.

so forth, all be implemented in the same compilation unit. Neither this approach nor the use of magic names scales well.

Modularity suffers if the programmer who uses a "portable" API of graphics routines, like the fictitious PGL in Figure 12, has to worry about how it is implemented. With only one single global name space the programmer must be careful not to use any of the symbols taken by, for example, the X Windows libraries [Scheifler et al. 1988], if the program later has to be 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.

The client of PGL in Figure 12 cannot use certain identifiers because they are potentially taken by the libraries that represent PGL. With CM, one would implement PGL as a group whose export list mentions only those identifiers that are supposed to be accessible by its clients. Consequently, the application programmer could not even tell how PGL is implemented; the application code will be truly independent of the libraries underlying PGL's concrete realization.

A CM 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.

4.3 Environments and Linking

Cardelli [1997] 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" [Appel and MacQueen 1994], is a prerequisite of our analysis of the linking problem, but not the focus. In place of Cardelli's linksets our notation uses functions to express operations on environments and equations for describing their properties. They directly correspond to actual operations on real symbol tables and reflect the implementation of CM.

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 ρ is a partial mapping from long identifiers Ide^+ to denotations D. Long identifiers are nonempty sequences of simple identifiers. They are used to express access to members of a structure by means of a "dot notation" as can be 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 $z \in \mathrm{Ide}$ is a simple identifier and where $Y = \langle id_1, \ldots, id_k\rangle$.

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

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

The domain D of denotations depends on the programming language. In SML it would correspond to the compilation unit's static and dynamic semantics. In C,

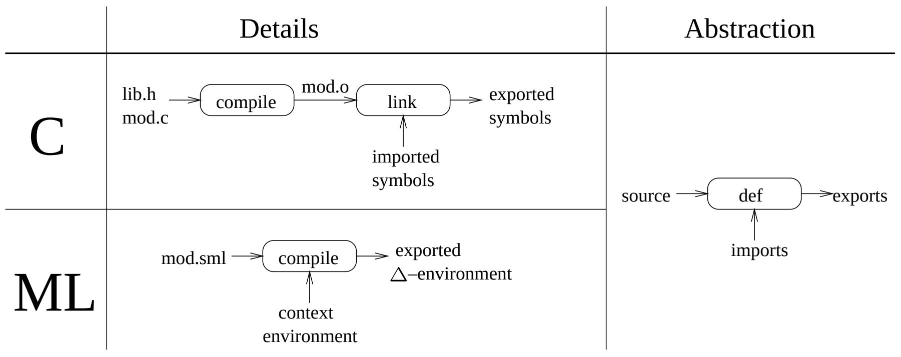


Fig. 14. Compiling, linking, context. A C source file is first compiled and subsequently linked with respect to a table of imported definitions. In the case of SML, compiling and linking are combined into one step, but, again, there is a context environment representing definitions that are imported from other compilation units. We 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.

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, we instead use a domain of labels $l \in \text{Lab}$. These labels uniquely identify each definition of a given program. Thus, we 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.

$$\begin{aligned} D &= \text{Lab} \\ \rho \in U &= \text{Ide}^{+} \to D \end{aligned}$$

The domain $dom(\rho) \subset \text{Ide}^{+}$ of an environment ρ is the set of identifiers that are bound there. We call $\{\text{Hd}(x) \mid x \in dom(\rho)\}$ the head domain $dom_H(\rho) \subset \text{Ide}$ 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:

$$Y.z \in dom(\rho) \Rightarrow Y \in dom(\rho)$$

Environments can be combined using the $\ltimes$ operator:

$$\begin{aligned} \ltimes &: U \times U \to U \\ \rho_1 \ltimes \rho_2 &= \lambda x. \begin{cases} \rho_1(x); & \text{Hd}(x) \in dom_H(\rho_1) \\ \rho_2(x); & \text{otherwise} \end{cases} \end{aligned} \qquad (1)$$

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

Compiling and Linking. We use the *def* operator (see Figure 14) 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:

$$def : \text{Source} \times U \to U$$

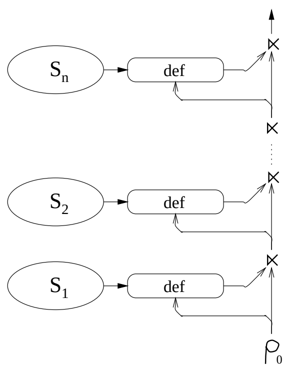


Fig. 15. Compilation environments for SML. In SML the context environment that is used for compiling a source is built incrementally by layering the exports of sources that were compiled earlier on top of the initial basis environment ρ_0.

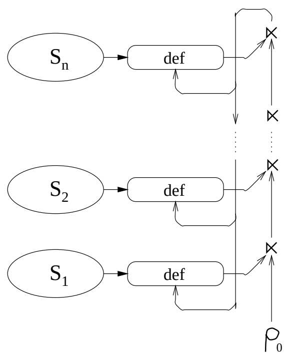


Fig. 16. Compilation environments for C. Conceptually, every source of a C program is compiled with respect to the same global environment, which itself is constructed by layering the exports from all sources on top of some initial basis environment ρ_0. Implementations resolve the circularity at link-time; but at compile time the compiler takes information from header files and uses it as an approximation of the global environment.

Programming languages differ in how they calculate the input environment for *def*. Consider a program consisting of sources $s_1, \ldots, s_n$ written in SML. The export environment ρ_i of source s_i is

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

where ρ_0 is the initial basis environment. This situation is depicted in Figure 15.

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 (see Figure 16):

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

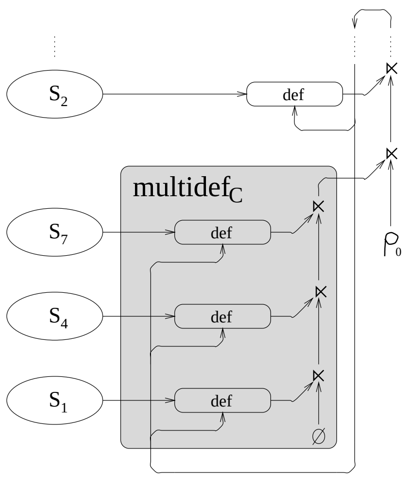


Fig. 17. Linking subsets of C sources. The *multidef*$_C$ operator is the C-specific extension of *def*. It calculates the export environment for a given subset of sources. *multidef*$_C$ returns an incremental delta environment that binds only those symbols explicitly defined in that subset of sources. This is consistent with the way *def* works for single sources.

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:

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

Because of the resulting commutativity of $\ltimes$, 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. We define

$$multidef_{lang} : 2^{\text{Source}} \times U \to U$$

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

C. Every global definition is visible in the entire program. Therefore, the *multidef*$_C$ operator passes its context argument (which in fact is the global environment) to all individual calls to *def* for each of the constituent sources. The resulting delta environments are combined using $\ltimes$, thus yielding a delta environment for the entire subset (see Figure 17):

$$multidef_{\text{C}}(\{s_1, \ldots, s_n\}, \rho) \;=\; def(s_1, \rho) \ltimes \cdots \ltimes def(s_n, \rho)$$

SML. *multidef*$_{\text{SML}}$ must first use a dependency analyzer $\mathcal{A}$ to turn the set of sources into a sequence [Blume 1999]:

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

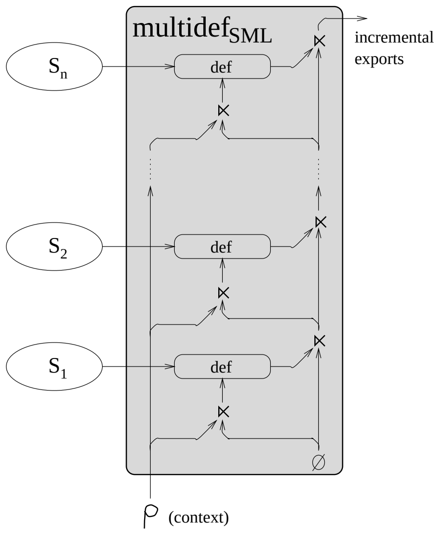


Fig. 18. Compiling (ordered) sets of SML sources. The *multidef*$_{\text{SML}}$ operator extends *def* to sets of SML sources. Individual exports are incrementally layered on top of an initially empty environment. The result is a delta environment only containing bindings for identifiers defined in $\{s_1, \ldots, s_n\}$, which is analogous to the behavior of *def*.

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:

$$\begin{aligned} \mathit{step}(\langle\rangle, \rho, \rho') &= \rho' \\ \mathit{step}(\langle s_i, s_{i+1}, \ldots\rangle, \rho, \rho') &= \mathit{step}(\langle s_{i+1}, \ldots\rangle, \rho, \mathit{def}(s_i, \rho' \ltimes \rho) \ltimes \rho') \\ \mathit{multidef}_{\text{SML}}(S, \rho) &= \mathit{step}(\mathcal{A}(S, \rho), \rho, \emptyset_U) \end{aligned} \tag{2}$$

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 (see Figure 18). This explains why there are two occurrences of $\ltimes$ in the definition for *step*.

SML, Using Partial Orders. If s_i occurs to the left of s_j in $\mathcal{A}(\{s_1, \ldots, s_n\}, \rho)$, then intuitively this means that s_j "depends" on s_i. To be able to optimize recompilation, we may want to 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.

One can represent such a partial order as a DAG of sources given by the "predecessor" function $\mathcal{P} : \text{Source} \to 2^{\text{Source}}$. The dependency analyzer must calculate $\mathcal{P}$:

$$\mathcal{A} : \left(2^{\text{Source}} \times U\right) \to \left(\text{Source} \to 2^{\text{Source}}\right)$$

Now *multidef*$_{\text{SML}}$ must be revised accordingly. If we compile a source set S with respect to ρ, then we must express the result $\mathit{def}_{\mathcal{P}}(s, \rho)$ of compiling a source $s \in S$

in terms of the base environment ρ, s itself, and the exports of its predecessors $s' \in \mathcal{P}(s)$[2]:

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

In general, the summations in these equations still require a total order imposed on the set of sources. This problem is resolved if we ensure that export domains of individual sources are disjoint. It turns out that such a restriction is necessary anyway because otherwise dependency analysis becomes intractable [Blume 1999]. Therefore, CM enforces this rule within each group—but not globally! (If we were to impose such a restriction globally, then it would cause restrictions on definability.)

As an important aspect of our group model we also require that each name x imported by a source $s \in S$ (i.e., x free in s) and exported by another source $s' \in S$ will be resolved to the definition in s'. In other words, definitions within S always take precedence over definitions for the same name that come from the context environment ρ. Formally, this can be expressed by the following invariant on $\mathcal{P} = \mathcal{A}(S, \rho)$:

$$\forall s, s' \in S : \forall x \in \mathrm{Ide}^+ : (x \text{ free in } s \wedge x \in \mathit{dom}(\mathit{def}_{\mathcal{P}}(s', \rho)) \wedge s \neq s') \Rightarrow s' \in \mathcal{P}(s)$$

4.4 Definability and Availability Revisited

Definability. Formally, the definability of a name in C can be determined according to the following rule:

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

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.

4.4.0.1 *Availability.* 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,[3] 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:

$$
\begin{array}{l}
\text{definition for } x \text{ from } s_i \text{ available in } s_j \text{ iff} \\
\quad \forall k : (i < k) \wedge (k < j) \Rightarrow \mathrm{Hd}(x) \notin \mathit{dom}_H(\rho_k)
\end{array}
$$

[2]The summation $\sum^{\ltimes}$ is based on the binary operation $\ltimes$.

[3]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.

Thus, in SML, unlike the situation in C, definitions do not interfere with the definability of names. But their impact on availability is still nonlocal: the programmer has to be aware of all the definitions exported from *earlier* sources. Again, the resulting implicit coupling inhibits modularity.

4.5 Groups

Groups localize the effects of definitions. The group model is constructed in such a way that names in unrelated groups do not interfere. Only 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. This establishes a direct relationship and makes conflicts between the two groups modular.

Formally, a group g : Grp is a triple (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.

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

Given a set of groups $G \subset \mathrm{Grp}$ 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, which is a reduction of the number of definitions that are exported by a group, 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.

$$\begin{aligned} \mathcal{F} &: U \times 2^{\mathrm{Ide}} \to U \\ dom_H(\mathcal{F}(\rho, E)) &= dom_H(\rho) \cap E \\ \mathrm{Hd}(x) \in E &\Rightarrow \mathcal{F}(\rho, E)(x) = \rho(x) \end{aligned}$$

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 (the "standard library"). $\mathcal{C}(I)$ 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.

$$\begin{aligned} \mathcal{C} &: 2^{\mathrm{Grp}} \to U \\ \mathcal{C}(I) &= (\sum_{i \in I}^{\ltimes} \mathcal{E}(i)) \ltimes \rho_0 \end{aligned} \tag{3}$$

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

$$\begin{aligned} \mathcal{E} &: \mathrm{Grp} \to U \\ \mathcal{E}(S, I, E) &= \mathcal{F}(multidef_{\mathrm{SML}}(S, \mathcal{C}(I)) \ltimes \mathcal{C}(I), E) \end{aligned} \tag{4}$$

A group (S, I, E) can reexport part of its own context $\mathcal{C}(I)$. For this, the definition to be reexported must be named in E and cannot be redefined by any of the sources $s \in S$.

We have chosen to consider unordered sets of imported groups. But the summation in Eq. (3) is order-independent only if the domains of the imported environments are disjoint. Therefore, we require that indirectly related groups do not export definitions for the same identifier.

$$\begin{aligned} &\forall (S, I, E) \in \mathrm{Grp} : (\{i_1, i_2\} \subset I \Rightarrow \\ &\quad \mathit{dom}(\mathcal{E}(i_1)) \cap \mathit{dom}(\mathcal{E}(i_2)) = \emptyset \end{aligned} \tag{5}$$

4.6 How Modular Conflicts Can Be Resolved

In our formalism, the equivalent of a name conflict is the case where two environments whose domains are not disjoint participate in the same $\ltimes$ operation. In the group model, no such operation ever applies to export environments of unrelated modules. Therefore, all possible conflicts are in fact modular (see Definition 2).

To be practical, it is very important for the group model that those remaining modular conflicts 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 $\ltimes$ according to Eq. (2), which defines $\mathit{multidef}_{\mathrm{SML}}$.

(2) A source provides a definition for an identifier that is also defined by one of the imported groups.

(3) In violation of Eq. (5) 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 (see Eqs. (2)–(4)). 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, we show how to rename a long identifier:

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

Renaming in environments can then be defined in terms of identifier renaming. If ρ' is obtained from ρ by renaming x to y, then looking up y in ρ' is the same as looking up x in in ρ. Thus, R and $\mathcal{R}$ take their arguments x and y in opposite order (Eq. (6)).

$$\begin{aligned} \mathcal{R} &: U \times \mathrm{Ide} \times \mathrm{Ide} \to U \\ \mathcal{R}(\rho, x, y)(z) &= \rho(R(z, y, x)) \end{aligned} \tag{6}$$

In general, to formally describe renaming one must extend the notion of groups. Imported groups (see domain Eq. (7)) are described by regular groups and an arbitrary number of identifier pairs. The pairs specify renaming operations (see Eq. (8)).

$$\mathrm{Imp} = \mathrm{Grp} + (\mathrm{Imp} \times \mathrm{Ide} \times \mathrm{Ide}) \tag{7}$$

$$\begin{aligned} \mathrm{Grp} &= 2^{\mathrm{Source}} \times 2^{\mathrm{Imp}} \times 2^{\mathrm{Ide}} \\ \mathcal{C}(I) &= (\sum_{i \in I}^{\ltimes} \mathcal{E}'(i)) \ltimes \rho_0 \\ \mathcal{E}(S, I, E) &= \mathcal{F}(\mathit{multidef}_{\mathrm{SML}}(S, \mathcal{C}(I)) \ltimes \mathcal{C}(I), E) \\ \mathcal{E}'(g) &= \mathcal{E}(g) \\ \mathcal{E}'(i, x, y) &= \mathcal{R}(\mathcal{E}'(i), x, y) \end{aligned} \tag{8}$$

In the special case when dealing with a language like ML, renaming does not have to be built into the compilation manager explicitly but can be obtained by using "administrative" groups. Thus, CM currently only implements the original model without renaming.

An administrative group imports a binding under one name and exports it under a different one. This is possible because renaming in ML can be expressed in source language (see Figure 19). 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`.

This behavior of ML exhibits limitations to our approach of abstracting denotations as labels. To deal with this case correctly, we could have introduced an explicit equivalence relation on labels. However, it seemed unnecessary to further complicate our calculus just to be able to handle this minor, language-specific point.

C and Scheme are examples for languages where administrative groups do not work because there is no general way of defining one name to be an alias for another. A variable definition in these languages creates a new, unique meaning. This means that a CM-like compilation manager would have to implement renaming directly. It should also be noted that even in Standard ML 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, right now this is not an issue because CM does not deal with type or value definitions at the top level. Only structures, signatures, and functors are tracked.

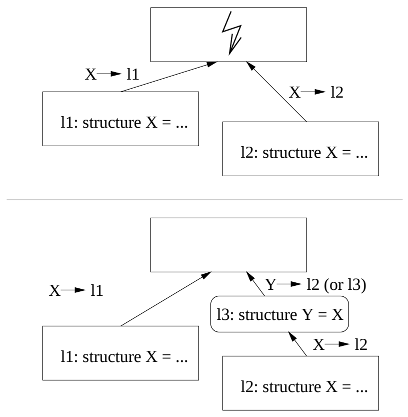


Fig. 19. Administrative groups. In SML one can achieve the effect of renaming upon import by administrative groups because the source language provides facilities of defining one identifier to be a proper alias of another. Many other languages, including C and Scheme, lack such a language feature. For those languages it is necessary to build renaming into the group model.

4.7 Implicit Export Interfaces

We have already mentioned that group descriptions for nonlibraries may leave their export list empty. In this case CM provides a default. The group then exports everything defined by its own sources and everything that is imported from other nonlibrary subgroups. Formally, let $(S, I, \cdot)$ be a group and $I_L \subseteq I$ be the set of imported groups that are libraries. The export list $\hat{E}$ provided by CM is calculated as

$$\hat{E} = \mathit{dom}(\mathit{multidef}_{\mathrm{SML}}(S, \mathcal{C}(I)) \ltimes \sum_{i \in I \setminus I_L}^{\ltimes} \mathcal{E}(i)). \tag{9}$$

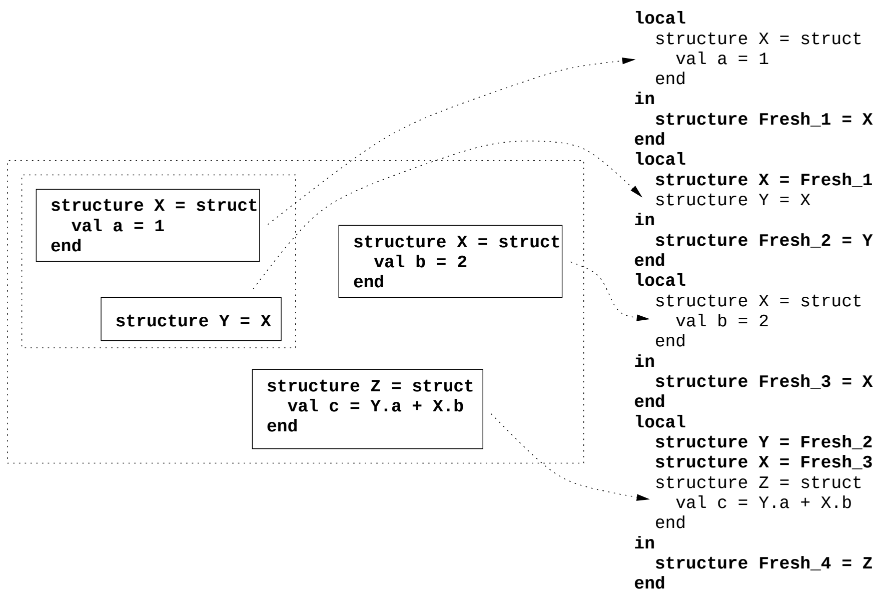
4.8 Back to ML

We can provide a more self-contained explanation for the meaning of a program managed under CM by providing a translation back to Standard ML. This translation will also provide the dynamic part of CM's semantics (link-time execution and state). So far we had only dealt with the static part.

The translation will turn the entire program consisting of many individual compilation units into one single piece of ML source code. As such, it is not meant as a practical way of implementing CM but merely as a means of gaining insight into CM's semantics. See Figure 20 for an example.

The translation will embed the code contained in each individual compilation unit into one single skeleton of glue code. The glue code will play the role of the global environment, providing import bindings for each compilation unit and remembering the resulting export bindings. The final result of the translation is a concatenation of source fragments where the fragments themselves are obtained from the original sources by the following method.

We start with the full dependency graph of the program. This graph is directed



```
structure X = struct
  val a = 1
end
```

```
structure Y = X
```

```
structure X = struct
  val b = 2
end
```

```
structure Z = struct
  val c = Y.a + X.b
end
```

```
local
  structure X = struct
    val a = 1
  end
in
  structure Fresh_1 = X
end
local
  structure X = Fresh_1
  structure Y = X
in
  structure Fresh_2 = Y
end
local
  structure X = struct
    val b = 2
  end
in
  structure Fresh_3 = X
end
local
  structure Y = Fresh_2
  structure X = Fresh_3
  structure Z = struct
    val c = Y.a + X.b
  end
in
  structure Fresh_4 = Z
end
```

Fig. 20. Back to ML. The left-hand side of this figure schematically shows the CM group structure of a simple program. Solid boxes are Standard ML source files; dotted boxes symbolize grouping. The right-hand side shows the source of an equivalent single-source Standard ML program. In this program there are explicit renaming operations where CM would normally have handled them automatically.

and acyclic. First we pick a postorder traversal of that graph. The code fragments generated from the original sources will be concatenated in that order. Postorder guarantees that each source appears after the sources it depends on.

We have left some nondeterminism in our description because we have not specified the actual traversal. This leaves some freedom in how the order of link-time side effects is chosen. One could eliminate the uncertainty, which perhaps would be undesirable from the purist's point of view, by prescribing an ordering among sibling nodes of the dependency graph and by requiring a particular kind of traversal. In our actual implementation we have not done that because so far it has not been a problem in practice. As a workaround it is always possible for the programmer to create artifical static dependencies which serve to enforce the desired ordering. (Another way of making this aspect of the semantics more predictable would be to have the programmer supply a total ordering of the sources in each group—thereby losing one of main features that makes CM so convenient to use. A practical advantage of such a design would be that dependency analysis becomes nearly trivial.)

A code fragment S' is obtained from its source S by embedding S into a set of pre- and postdefinitions. The postdefinitions map bindings exported by S to fresh identifiers, while the predefinitions provide bindings for the free identifiers of S by referring to those auxiliary identifiers that are bound by the postdefinitions of the predecessors of S.

We use our environment formalism to obtain the correct set of pre- and postdefinitions. Therefore, we change the domain for environments to be the set of program identifiers: $D = \text{Ide}$. In other words, the compilation environment ρ for a source S maps the source's free identifiers to other identifiers: $\rho : \text{Ide} \to \text{Ide}$.

The code of S' first establishes the free identifiers of S as aliases of their corresponding names in ρ. Let $x_1 \ldots x_n$ be the free identifiers of S. Then there will be a definition for each i that equates x_i with $\rho(x_i)$. The precise syntax of that definition in Standard ML depends on what name space x_i belongs to. In particular, we use

$$\text{namespace}(x_i)\ \ x_i\ = \ \rho(x_i)$$

where namespace(x) is either **structure**, **functor**, **signature**, or **funsig**.

Finally, let $y_1 \ldots y_m$ be the names bound and exported by S. We generate fresh names $z_1 \ldots z_m$, bind them to those exports, and let the export environment $\mathit{def}(S, \rho)$ map each y_j to the corresponding z_j. By keeping the x_i and y_j in local scopes we can be sure that there are no name clashes, since the z_j are all chosen fresh. Thus, the fragment S' derived from S looks as follows:

```
local
   namespace(x_1)  x_1 = ρ(x_1)
   ⋮
   namespace(x_n)  x_n = ρ(x_n)
   S
in
   namespace(y_1)  z_1 = y_1
   ⋮
   namespace(y_m)  z_m = y_m
end
```

5. CONCLUSIONS

The group model employed by SML/NJ's compilation manager enables modular large-scale programming. Traditionally, this has been difficult because the potential for naming conflicts creates nonmodular dependencies between otherwise unrelated parts of a program. In the cases that we have examined, these problems manifested themselves as restrictions on definability of names or availability of definitions.

CM's group model arranges sources into a hierarchical structure. Export environments are combined only when necessary, thereby avoiding most name clashes. Although it cannot eliminate all clashes, it controls them in such a way that they do no longer compromise modularity because local modifications suffice to resolve them.

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.

How viable are our approaches if we tried to apply them to languages other than ML? This question should be asked in two parts. First, is it possible to implement hierarchical modularity and automatic dependency analysis 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." We have demonstrated 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.

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. We do not have a measure for elegance, and even if we did, we could only speculate. But we 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 particularly well suited to the implementation of compilers and—it turns out—compilation management tools. 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.

Aside from the many positive aspects of ML, there have been some that proved difficult to deal with. These cases mostly concern automatic dependency analysis and have been described separately [Blume 1999].

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.

REFERENCES

Adams, P. and Solomon, M. 1993. An overview of the CAPITL software development environment. Tech. Rep. CS-TR-93-1143, Computer Sciences Department, University of Wisconsin – Madison. 4.

Adams, R., Tichy, W., and Weinert, A. 1994. The cost of selective recompilation and environment processing. *ACM TOSEM 3,* 1 (January), 3–28.

Appel, A. W. 1998. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, England.

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey. In *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, M. Wirsing, Ed. Springer-Verlag, New York, 1–13.

Appel, A. W. and MacQueen, D. B. 1994. Separate compilation for Standard ML. In Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation. *SIGPLAN Notices 29,* 6 (June), 13–23.

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. Addison Wesley, Reading, MA.

BARENDREGT, H. P. 1981. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam.

BAUER, L., APPEL, A. W., AND FELTEN, E. W. 1999. Mechanisms for secure modular programming in Java. Tech. Rep. TR-603-99, Department of Computer Science, Princeton University. July.

BLUME, M. 1995. Standard ML of New Jersey compilation manager. Manual accompanying SML/NJ software.

BLUME, M. 1999. Dependency analysis for Standard ML. *ACM Trans. Program. Lang. Syst. 21,* 4 (July), 790–812.

BRODER, A. 1993. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, R. Capocelli, A. D. Santis, and U. Vaccaro, Eds. Springer-Verlag, 143–152.

BROWN, M. R. AND ELLIS, J. R. 1993. Bridges: Tools to extend the Vesta configuration management system. Tech. Rep. 108, Digital Equipment Corp. Systems Research Center. 6.

CARDELLI, L. 1997. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*. 266–277.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1988. Modula-3 report. Tech. Rep. Research Report 31, DEC Systems Research Center, Palo Alto, CA.

CHIU, S.-Y. AND LEVIN, R. 1993. The Vesta repository: A file system extension for software development. Tech. Rep. 106, Digital Equipment Corp. Systems Research Center. June.

CHURCH, A. 1941. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ.

CLEMM, G. M. 1994. The Odin System — Reference Manual.

DoD 1980. Military standard: Ada programming language. Tech. Rep. MIL-STD-1815, Department of Defense, Naval Publications and Forms Center, Philadelphia, PA.

DUBOIS, P. 1996. *Software Portability with imake, 2nd Edition*, 2nd ed. O'Reilly and Associates, Sebastopol, CA.

ELLIS, M. A. AND STROUSTRUP, B. 1990. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA.

FELDMAN, S. I. 1979. Make – a program for maintaining computer programs. In *Unix Programmer's Manual, Seventh Edition, Volume 2A*. Bell Laboratories.

FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., AND SHIVERS, O. 1997. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*.

GUNTER, C. A. 1996. Abstracting dependencies between software configuration items. In *Proceedings of the Fourth Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 167–178.

HANNA, C. B. AND LEVIN, R. 1993. The Vesta language for configuration management. Tech. Rep. 107, Digital Equipment Corp. Systems Research Center. June.

HARPER, R., LEE, P., PFENNING, F., AND ROLLINS, E. 1994b. A Compilation Manager for Standard ML of New Jersey. In *1994 ACM SIGPLAN Workshop on ML and its Applications*. 136–147.

HARPER, R., LEE, P., PFENNING, F., AND ROLLINS, E. 1994a. Incremental recompilation for Standard ML of New Jersey. Tech. Rep. CMU-CS-94-116, Department of Computer Science, Carnegie-Mellon University. Feb.

JACOBSON, I. 1987. Object oriented development in an industrial environment. In *OOPSLA '87: Object-Oriented Programming Systems, Languages and Applications*. ACM SIGPLAN, 183–191.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, New Jersey 07632.

LAMPSON, B. W. AND SCHMIDT, E. E. 1983a. Organizing software in a distributed environment. In *ACM SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*. 1–13.

LAMPSON, B. W. AND SCHMIDT, E. E. 1983b. Practical use of a polymorphic applicative language. In *Tenth Annual ACM Symposium on Principles of Programming Languages*. 237–255.

LEVIN, R. AND MCJONES, P. R. 1993. The Vesta approach to precise configuration of large software systems. Tech. Rep. 105, Digital Equipment Corp. Systems Research Center. June.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

MITCHELL, J. G., MAYBURY, W., AND SWEET, R. 1979. Mesa language manual. Tech. Rep. CSL-79-3, Xerox PARC.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12 (Dec.), 1053–1058.

SCHEIFLER, R. W., GETTYS, J., AND NEWMAN, R. 1988. *X Window System: C Library and Protocol Reference*. Digital Press, Bedford, MA.

SWINEHART, D. C., ZELLWEGER, P. T., AND HAGMANN, R. B. 1985. The structure of Cedar. In ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments. *SIGPLAN Notices 20,* 7, 230–245.

Received July 1998; revised April 1999; accepted July 1999