Fully-Parameterized Modules
or
The Missing Link

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

Abstract

We describe a module system that is a fusion of a traditional linker such as Unix's `ld` with the explicit interfaces and cross-module type-checking of languages such as ML [10], Ada [11], and Modula-3 [6]. The modules are fully parameterized: a module specification includes only the imported interfaces and not the modules that supply them. Using fully-parameterized modules makes programs easier to read and modify. Although allowed by such languages as ML, use of fully-parameterized modules is discouraged by the need to specify the complete module dependency graph. We solve this by using a linker to add module interconnections automatically, avoiding the need for explicitly specifying large and complex module dependency hierarchies. When necessary, either for clarity or to resolve ambiguities, module dependencies can also be specified explicitly. The module system is largely independent of both the core language used within modules and the configuration language used to describe how the modules are used. The system is defined as a collection of abstract data types, and as such can be embedded in many existing programming languages, avoiding the need for a special-purpose configuration language.

1 Introduction

Parameterized modules, such as ML's functors, have many benefits, including increased program clarity and modularity. Their use is typically hampered by the need for the programmer to specify the complete module dependency graph [1]. For more traditional languages, for example C and FORTRAN, the dependency graph between program components is inferred automatically. An object-code linker, such as Unix's `ld`, matches uses of variables with their definitions. There are two problems with this approach. First, the lack of explicit interfaces makes inter-module type-checking difficult. Second, the linker uses a single name space for all global variables, leading to frequent name clashes.

In this paper we present a module system that combines the well-defined semantics, type-safety, and clarity of a modern module system with the convenience of a classic object-code linker. Instead of matching variable definitions to uses, the linker matches implementations of interfaces to modules that require them. The modules are fully-parameterized: each specifies the interfaces for which it requires implementations (imports) and those for which it provides them (exports). Typically a program contains only a single provider of a particular interface, in which case the linker can safely connect the providing module all of the modules that import the interface. In ambiguous situations, where there are two or more modules that implement the same interface, the programmer must resolve the ambiguity, either by linking subsets of the program's modules separately or by explicitly connecting exporters with importers.

By merging the two paradigms we have designed a module system that has significant advantages over traditional linkers:

- User-provider matching is done using generated interfaces, not just names. Accidental name clashes cannot occur.
- Static information from interfaces allows inter-module type checking at compile time
- Individual user-provider links can be added explicitly if necessary.

and over more modern module systems:

- All modules are fully parameterized.
- Using a linker instead of functor application to specify the program dependency graph is both more succinct and more powerful.
- The choice of configuration language is independent of the core language used in the modules, avoiding the need for a specialized configuration language.

A final advantage over both paradigms is the possibility of arbitrarily intermixing compilation and linking; see Section 3.4.

The rest of this paper is organized as follows. We begin with an example that contrasts the two existing approaches. The main body is divided into three sections: a description of the module system, a discussion of the issues raised by various core language features, and a collection of extensions that increase the power and expressiveness of the module system. We finish with a comparison with previous work.

2 Explicit and implicit dependencies

The following is an example of a fully-specified dependency graph. For simplicity functor application is written as ML function application, instead of using ML's more verbose functor notation.

```
structure Colors = Colors();
structure ColoredGraph = Graph(Colors);
structure Search = DFS(ColoredGraph, Stack(ColoredGraph.node));
structure main = GraphColorer(Colors, Graph, Search);
```

If we want to produce a second version of `main` that uses breadth-first search instead of depth-first we can parameterize the program over the search functor (assuming we have some form of higher-order functors). It is slightly awkward that breadth-first search requires a queue instead of a stack. (`funct` assumed to be a version of `fn` that creates anonymous functors).

```
functor MakeMain (SearchF) =
  structure Colors = Colors();
  structure ColoredGraph = Graph(Colors);
  structure GraphSearch = SearchF(ColoredGraph);
  GraphColorer(Colors, Graph, GraphSearch);
main0 = MakeMain(funct Graph => DFS(Graph, Stack(Graph.node)));
main1 = MakeMain(funct Graph => BFS(Graph, Queue(Graph.node)));
```

In the absence of multiple implementations of an interface the explicit dependency graph is a duplication of information in the module definitions. The presence of multiple implementations makes it awkward to express the dependency graph, which in turn makes it hard to determine exactly which implementations are used by which module.

If the same program were written in C, with each functor being written as a separate C file, the same dependency graph would be expressed as the following:

```
ld -o main Colors.o Graph.o DFS.o Stack.o GraphColorer.o
```

Note that the files can be listed in any order. Parameterizing it over two different versions of search is simple (`-r` tells `ld` that the output file is meant to be incomplete):

```
ld -o same -r Colors.o Graph.o GraphColorer.o
ld -o main0 same DFS.o Stack.o
ld -o main1 same BFS.o Queue.o
```

For program as tiny as the one shown above it doesn't make much difference which method is used. For larger programs the explicit dependency graph rapidly becomes unwieldy. C doesn't solve the problem either, of course. C has a single top-level name space and no explicit interfaces. The above example works only if no two modules export the same name.

3 The module system

In this section we will define a minimal version of the module system. Various extensions are presented in Section 5.

A program is organized as a set of modules, each with its own portion of source code and top-level name space. Names defined in one module are made visible in others by way of interfaces, which are essentially sets of names with associated static information. Each module specification has two sets of interfaces: those for which it requires bindings (imports) and those for which it supplies them (exports). Within a program modules that import a particular interface can be linked to any module that exports that same interface. Once all of the modules within a program have been linked together, intermodule references in the source code can be resolved to the correct module-specific name.

The minimal module system consists of constructors for interfaces, modules, and programs, two linking operators, and the algorithm used to resolve inter-module references.

3.1 Interfaces, modules and programs

Interfaces are finite maps from names to static information, along with a unique identifier (uid). Unique identifiers are found both in interfaces and modules, and are akin to locations in a store. There is one-to-one correspondence between interface uids and interfaces. The two kinds of uids are used by the linker to match up exporters and importers of interfaces.

$$\begin{aligned} n &\in \text{Name} \\ N &\in \mathcal{P}(\text{Names}) \\ i &\in \text{Interface} = \text{Interface-Uid} \times \text{Static-Env} \\ se &\in \text{Static-Env} = \text{Name} \stackrel{fin}{\rightarrow} \text{Static} \end{aligned}$$

The exact nature of the static information depends on the core language used in the module. For ML, static information would include types and the types of values; for Scheme types would not be required but macros would have to be handled in some fashion. The extend to which modules can be compiled separately is determined by the availability of adequate static information in the interface; see Section 3.4.

A module consists of a unique identifier, a set of required interfaces (imports), a set of provided interfaces (exports), and the module's code. As with interfaces, there is one-to-one correspondence between module unique identifiers and modules. The module system itself does not make use of the code in a module, which may be program text, filenames, object code, or whatever else is appropriate for the core language.

$$m \in \text{Module} = \text{Module-Uid} \times \mathcal{P}(\text{Interface})_{imports} \times \mathcal{P}(\text{Interface})_{exports} \times \text{Code}$$

A program is a set of modules and an environment containing inter-module linking information. The link environment maps the pair of importing module and interface to a module exporting bindings for the interface. A program's link environment may only reference modules in the program's module set.

$p \in \text{Program} = \mathcal{P}(\text{Module}) \times \text{Link-Env}$

$le \in \text{Link-Env} = (\ \text{Module}_{importing} \times \text{Interface}\ ) \xrightarrow{fin} \text{Module}_{exporting}$

The core module language consists of constructors for interfaces, modules, and programs and two linking functions. *Make-interface* and *make-module* are both simple constructors. Each call to *make-interface* and *make-module* creates a new unique identifier.

$\textit{make-interface} : se \rightarrow\ i$

$\textit{make-interface}(se) = \langle \textit{new-interface-uid}(), se \rangle$

$\textit{make-module} : I_{imports} \times I_{exports} \times \textit{code} \rightarrow\ m$

$\textit{make-interface}(I_{imports}, I_{exports}, \textit{code}) = \langle \textit{new-module-uid}(), I_{imports}, I_{exports}, \textit{code} \rangle$

Make-program constructs a new program from two existing programs or modules. A module argument to *make-program* is converted to programs containing that module and an empty link environment. When passed two programs *make-program* makes a new program containing the union of the two sets of modules and a merge of the two link environments. The two link environments must be mutually consistent; in the resulting link-environment a module can be linked to at most one provider for each interface it imports. A module argument passed to *make-program* is coerced to a program with an empty link environment: $m \Rightarrow \langle \{m\}, \{\} \rangle$. *Make-program* can easily be extended to take an arbitrary number of programs and modules as arguments.

$$\textit{make-program} : p \times p \rightarrow\ p \text{ or } \textit{error}$$

$$\textit{make-program}\ (\langle M_0, le_0 \rangle, \langle M_1, le_1 \rangle)$$

$$= \begin{cases} \langle M_0 \cup M_1, le \rangle & \text{if } \forall \langle m, i \rangle \in dom(le_0) \cap dom(le_0), \\ & le_0(\langle m, i \rangle) = le_1(\langle m, i \rangle) \\ \textit{error} & \text{otherwise} \end{cases}$$

where

$$le(\langle m, i \rangle) = \begin{cases} le_0(\langle m, i \rangle) & \text{if } \langle m, i \rangle \in dom(le_0) \\ le_1(\langle m, i \rangle) & \text{otherwise} \end{cases}$$

3.2 Linking

The two linking functions are *link*, which adds a single link to a program, and *link-all*, which adds all non-conflicting, non-ambiguous links to a program.

$\textit{link} : p \times m_{export} \times i \times m_{import} \rightarrow\ p \text{ or } \textit{error}$

$\textit{link-all} : p \rightarrow\ p$

The arguments to *link* must be consistent: the program must contain the two modules, the importing module must import the interface, the exporting module must provide it, and, finally, the program's link environment must not map $\langle m_{imp}, i \rangle$ to a value other than m_{export}.

Given a program $\langle M, le \rangle$, *link-all* extends le with every $\langle m_{import}, i \rangle \mapsto m_{export}$ such that m_{export} is the only module in M that exports i, $\langle m_{import}, i \rangle \notin dom(le)$, and m_{import} imports interface i.

As an illustration of how *link* and *link-all* can be used, consider the following example (in which *i0*, *i1*, and *i2* are interfaces and *make-program* had been extended to take an arbitrary number of arguments):

$$
\begin{aligned}
&m0 = make\text{-}module([],\ [i0],\ "\ldots")\\
&m1 = make\text{-}module([i2],\ [i0],\ "\ldots")\\
&n0 = make\text{-}module([i0{,}i2],\ [i1],\ "\ldots")\\
&n1 = make\text{-}module([i0{,}i1],\ [],\ "\ldots")\\
&p0 = link\text{-}all(make\text{-}program(m0,\ m1,\ n0,\ n1))\\
&p1 = link\text{-}all\ (link\ (link\ (make\text{-}program\ (m0,\ m1,\ n0,\ n1),\\
&\qquad\qquad\qquad\qquad\qquad m0,\ i0,\ n0),\\
&\qquad\qquad\qquad\qquad m1,\ i0,\ n1))\\
&p2 = link\text{-}all\ (make\text{-}program\ (link\text{-}all\ (make\text{-}program(m0,\ n0))),\\
&\qquad\qquad\qquad\qquad\qquad\quad link\text{-}all\ (make\text{-}program(m1,\ n1))))
\end{aligned}
$$

All three programs contain the same four modules. The only link in `p0` is `n1` importing interface `j` from `n0`. There are two exporters of interface `i`, so `link-all` is unable to link either of them to the importers of `i`, modules `n0` and `n1`. In both `p1` and `p2`, `n0` imports `i` from `m0` and `n1` imports `i` from `m1`. For `p1` these two links are added explicitly, for `p2` they are added by using `link-all` on subprograms. Note that in all three cases modules the final programs are incomplete, as they have no implementation of *i2*. A module's exports may be linked to importing modules before the exporting module's own imports have been determined. This is not possible with functors.

3.3 Name resolution

The final part of the core language is a function, *resolve*, that maps uses of an identifier in one module to a definition in another. *Resolve* takes a module and a link environment and returns a function mapping names and to qualified names. A qualified name is a name annotated with uid of the module in which it is bound. Using just the core module system the name portion of the qualified name will be the same as the name passed to the lookup function. This is no longer true if modules are allowed to rename imported bindings; see Section 5.1 below. Once a program has been fully linked all uses of identifiers can be replaced with qualified names.

$$
\begin{aligned}
&qn \in \text{Qualified-Name} = \text{Module-Uid} \times \text{Name}\\
&resolve : (m \times le) \to n \to\ qn \text{ or } error\\
&resolve(m, le) =\\
&\quad \lambda n.lookup(m,n)\\
&\qquad\quad \text{where } lookup(m,n)\\
&\qquad\qquad\qquad = \begin{cases} uid_m : n & \text{if } I_n = \{\}\\ qn & \text{if } \exists!\ qn \mid \forall_{i\in I_n} lookup(le(m,i),n) = qn\\ error & \text{otherwise}\end{cases}\\
&\qquad\qquad\qquad \text{where }\ m = \langle uid_m, I_{imports}, I_{exports}, code\rangle\\
&\qquad\qquad\qquad\qquad\quad I_n = \{\langle uid_i, se\rangle \in I_{imports} \mid n \in dom(se)\}
\end{aligned}
$$

Within a module a name is either defined in the module's code or imported from some other module (but not both; see next paragraph). Any name not imported from some other module must be defined in the module's code and thus are qualified with the current module's uid. A name can be supplied by more than one interface as long as the name is resolved to the same qualified name in each case.

The module system imposes the restriction that imported names may not be shadowed by local definitions within a module. Our experience with Scheme 48, which does not have this restriction, has shown that allowing shadowing leads to errors that are difficult to detect automatically. Allowing imported names to be aliased within a module, as defined in Section 5.1 below, makes it easy for a programmer can to work around the restriction.

The linking functions can easily construct circularities in the module dependency graph, but the *resolve* function is not well-defined in their presence. For the moment we will prohibit circular dependencies and have the linking functions return errors instead. An alternative solution is presented in Section 5.3.

3.4 Compiling and Executing Programs

Compiling or executing a program requires an implementation of the module system tailored to the core language (or languages) used within the program's modules. From the module system's point of view a compiler is a function that annotates a program with object code for a subset of the program's modules.

$$\mathit{compile} : p \times M \rightarrow p$$

The option of compiling several modules at once allows the compiler to perform cross-module checking and optimizations beyond that enabled by the static information in the interfaces. Implementing *compile* requires a compiler for the core language. The module system and the core language have three points of contact: the static information in interfaces, the code in modules, and the *lookup* functions created for modules in a linked program.

One approach is to have the static information and the module code be source text. A parser for the core language can be used to apply the lookup functions to the names in the module code, replacing each use with an appropriately qualified one. The resulting code can then be written out and compiled as a core language program. In this instance the program is being linked as source code and linking must precede compilation.

Alternatively, the module system could add name qualifiers to object code instead of source. The code for each module would be merged with the static data from any imported interfaces and compiled as a separate core program. After linking, names in the object codes would be replaced with qualified ones. The resulting object codes would be merged into a single program using a traditional linker, such as Unix's `ld`.

If the capability exists to update names in both source and object code the module system can be used to control the extent to which static information is shared between modules. Linking modules together before compilation can allow the compiler to produce more efficient output, at the cost of increasing the recompilation needed after changes, either in the modules' code or in the program structure. For a language with macros, such as Scheme, it may be necessary to link modules before compilation to allow the compiler access to macro definition.

4 Problematic language features

In this section we discuss the ways in which various core language features, such as macros or a requirement for static type checking, interact with the module system. To make the discussion concrete we show how the module system can support ML's static type checking, Scheme's hygenic macros, and mutable module-level state as present in many languages.

4.1 Static type checking

The module system presented here could be used as an alternative to Standard ML's modules for programs written in Core ML. Interfaces and modules would correspond to ML structure signatures and functors. Because the module system is linguistically separated from the core language the interfaces would have to be "closed" and the modules would be full functors. This in turn allows separate compilation, as noted in [1].

The signatures of modules themselves would be parameterized signatures similar to those in [8]. The compiler would produce a signature for a module that gave concrete types for the opaque ones in the exported interfaces. This signature would be parameterized over types in the imported interfaces.

After linking these could be applied to the actual types and the sharing constraints checked. The concrete types in an exported interface might not be determined until the exporting module had been linked to appropriate providers of its imports. Once concrete types have been determined for all opaque types in interfaces, any sharing declarations can be checked.

4.2 Modules and state

Most programming languages allow programmers to modify the values of top-level variables, either by giving a variable a new value or by mutating the value it already has. For the module system this raises the issue of how many copies there are of each top-level variable. The rule is that each module within a program is instantiated only once, so there is exactly one copy of each top-level variable within a program. Including multiple copies of a single piece of source code in a program requires making multiple modules.

The following example defines modules each of which implements a single cell. Two procedures are exported, one for setting the value in the cell and the other for accessing it. Because a program is likely to want more than one cell a procedure is used to add cell modules to a program. When *add-cell* is invoked it creates a new cell module and adds it to the program. The new module is then linked to the available modules that import the *cell* signature, all of which will share a single cell instance. The core language used here is Scheme. The *cell* interface contains no static information.

$cell = make\text{-}interface([$"`cell-ref`", "`cell-set!`"$])$

```
cell-code = "(define value #f)
            (define cell-ref
              (lambda () value))
            (define cell-set!
              (lambda (new) (set! value new)))"
```

$add\text{-}cell = \lambda \,.\, program$

$\quad link\text{-}all(make\text{-}program(program,\ make\text{-}module([scheme],\ [cell],\ cell\text{-}code)))$

This example exposes a deficiency in the module system: using the above definition of *add-cell* no single module may make use of more than one cell. We will address this and other weaknesses in the next section.

The existence of side effects raises the issue of initialization order. In the absence of circularities in the dependency graph the modules can be initialized in a depth-first fashion, ensuring that no module is initialized before any of its imports. If circularities are permitted as described in Section 5.3, the initialization order for modules in a cycle can either be specified by the user or be arbitrarily fixed by the system.

$initial\text{-}order : p \times m_0 \times m_1 \rightarrow\ p$

Initial-order causes m_0 to be initialized before m_1 in the returned program.

4.3 Macros (a la Scheme)

Hygenic macros [2, 3] need a way of referring to bindings that are not necessarily in scope. This requires coordination between the renaming function and the macro expanders to rename macro-introduced names appropriately. This could take the form of a procedure passed to the expander for making non-local references and for examining the current top-level environment (the *lookup* procedure in [2]).

The one restriction that must be observed is that a module that exports syntax must be processed before any modules that make use of that syntax: if interface i includes syntax and module m_0 provides i to module m_1, then m_0 must be translated before m_1.

A module system based on first-class modules and interfaces fits well with Scheme's minimal syntax. Having the module specifications lexically separate from the program source makes sense for Scheme, which unlike most languages does not give its built-in syntax and procedures precedence over those of the programmer. This approach enforces a complete separation of the language within a module from the module language.

5 Extensions

In this section we present several extensions that increase the system's power and ease of use. They are all true extensions, in that they have no impact on programs which do not make use of them. At the end of this section is a new version of *resolve* that incorporates the extensions.

5.1 Aliases

Name clashes can be a problem even when using a module system. For example, a module may need to open two interfaces that coincidently include the same name. Opening both is an error in the system as defined above. Our solution is to allow bindings to be renamed when imported. We do this by including in an import specification a function that gives local aliases for the names in the imported interface.

$$m \in \text{Module} = \text{Module-Uid} \times \mathcal{P}(\text{Imports}) \times \mathcal{P}(\text{Interface})_{exports} \times \text{Code}$$
$$imp \in \text{Imports} = \text{Interface} \times \text{Aliases}$$
$$ali \in \text{Aliases} = \text{Name} \rightarrow (\text{Name} \cup \{\epsilon\})$$

The alias function is applied to the names in the interface to give the aliases under which they are visible in the importing module. If $ali(n) = \epsilon$ the binding of n is not available. The alias function is optional; it defaults to the identity function. *Resolve* uses the alias functions as follows: the import specification $\langle\langle uid_i, se\rangle, ali\rangle$ provides a definition of name n if there exists an $n' \in dom(se)$ such that $ali(n') = n$. When alias functions are used the name part of the qualified name returned by *resolve* is not necessarily the same as the name to which the lookup function was applied.

An alias function such as $\lambda n.append(\texttt{"m:"}, n)$ can be used to append a prefix to the names obtained from a particular interface. An imported interface can be restricted to a subset of its names using $\lambda n.n \in \{\texttt{"foo"}, \ldots\} \rightarrow n, \epsilon$.

5.2 Named imports

As mentioned above, the system does not allow a module to import multiple instances of the same interface. We can remove this prohibition by adding an optional tag to import specifications and then using the tags to disambiguate between multiple uses of the same interface in one module. Link environments now need to take a tag argument in addition to the importing module and the interface.

$$m \in \text{Module} = \text{Module-Uid} \times \mathcal{P}(\text{Imports}) \times \mathcal{P}(\text{Interface})_{exports} \times \text{Code}$$
$$imp \in \text{Import} = \text{Interface} \times \text{Tag}$$
$$tag \in \text{Tag} = \text{Name} \cup \{\epsilon\}$$
$$le \in \text{Link-Env} = (\ \text{Module}_{importing} \times \text{Interface} \times \text{Tag}\ ) \stackrel{fin}{\rightarrow} \text{Module}_{exporting}$$

The import tags are ignored by *link-all*. *Link* takes a tag as an additional argument.

$$link : p \times m_{export} \times i \times m_{import} \times tag \rightarrow\ p \text{ or } error$$

The importing module must have want the specified interface with the specified tag and the program's link-environment is extended using both the interface and the tag. The tag argument can be optional for both *make-module* and *link*, defaulting to ϵ.

A module that opens multiple copies of the same interface must use alias functions as described in the previous section to distinguish between the imported names in the module's code.

5.3 Circular dependencies

The linking functions can easily construct circularities in the module dependency graph, but the *resolve* function of Section 3.3 is not well-defined in their presence. The obvious solution is to prohibit circular dependencies. They can usually be avoided, sometimes at the cost of introducing a somewhat convoluted program structure. This has occasionally been a nuisance when using Scheme 48, which does not allow circularities.

An alternative is to modify *resolve*. *Resolve* can keep track of the names and modules that have been involved in the current lookup and return an error if an infinite loop is detected. The module dependency graph can contain loops so long as no loop is traversed while tracing a name back to its source.

5.4 Copying modules

There are several reasons why a programmer might want multiple instantiations of a single module specification. The cell example in Section 4.2 used multiple instantiations to get multiple copies of a module's state. Alternatively, a programmer might want to use two instances of a module linked to different providers of some interface. Because of its use of top-level state, it was obvious that multiple instantiations of the cell module would be required and its definition was embedded in a procedure for making copies. As this is not always going to be the case, a function for copying modules is useful.

$$\textit{copy-module} : m \to\ m$$

Copy-module takes a program and a module and returns a module that has a new unique identifier but is otherwise identical to original one.

5.5 A new *Resolve*

Here we present a new version of the *resolve* that encompasses the extensions discussed above (except for *copy-module*, which has no impact on *resolve*). Module import specifications now contain an interface, a tag, and an alias function. We also need to allow link environments to return *error* when a circular dependency is found.

$$\begin{aligned}
m &\in \text{Module} = \text{Module-Uid} \times \mathcal{P}(\text{Imports}) \times \mathcal{P}(\text{Interface})_{exports} \times \text{Code}\\
imp &\in \text{Import} = \text{Interface} \times \text{Tag} \times \text{Aliases}\\
tag &\in \text{Tag} = \text{Name} \cup \{\epsilon\}\\
ali &\in \text{Aliases} = \text{Name} \to (\text{Name} \cup \{\epsilon\})\\
le &\in \text{Link-Env} = (\ \text{Module}_{importing} \times \text{Interface} \times \text{Tag}\ ) \overset{fin}{\to} \text{Module}_{exporting} \cup \{\text{error}\}
\end{aligned}$$

Our strategy for handling circularities is to allow cycles in the graph of links but not to allow any cycle to be traversed during the lookup of any one name. This is done by restricting the link environment used in recursive calls to *lookup* to return *error* if the current module is referred to a second time.

$$
\begin{aligned}
&\mathit{resolve}(m, \mathit{le}) = \\
&\quad \lambda n.\mathit{lookup}(m, n, \mathit{le}) \\
&\qquad \text{where } \mathit{lookup}(m, n, \mathit{le}) \\
&\qquad\qquad = \begin{cases} \mathit{uid}_m : n & \text{if } \mathit{Hits} = \{\} \\ \mathit{qn} & \text{if } \exists \mathit{qn} \mid \forall_{\langle i, \mathit{tag}, n' \rangle \in \mathit{Hits}} \mathit{lookup}(\mathit{le}(m, i, \mathit{tag}), n', \mathit{le}') = \mathit{qn} \\ \mathit{error} & \text{if no unique } \mathit{qn} \text{ exists} \end{cases} \\
&\qquad\qquad \text{where } \; m = \langle \mathit{uid}_m, \mathit{Imp}, I_{\mathit{exports}}, \mathit{code} \rangle \\
&\qquad\qquad\qquad \mathit{Hits} = \{ \langle \langle \mathit{uid}_i, \mathit{se} \rangle, \mathit{tag}, n' \rangle \mid \langle \langle \mathit{uid}_i, \mathit{se} \rangle, \mathit{tag}, \mathit{ali} \rangle \in \mathit{Imp} \\
&\qquad\qquad\qquad\qquad \text{and } \exists n' \in \mathit{dom}(\mathit{se}) \text{ with } \mathit{ali}(n) = n' \} \\
&\qquad\qquad\qquad \mathit{le}'(m', i', \mathit{tag}') = \begin{cases} \mathit{error} & \text{if } \mathit{le}(m', i', \mathit{tag}') = m \\ \mathit{le}(m', i', \mathit{tag}') & \text{otherwise} \end{cases}
\end{aligned}
$$

6 Related work

The module system presented here is a mixture of three different approaches. Its relationship to ML-style functors and the classic object-code linker is discussed in the introduction. The third major influence, the module system used in Scheme 48 [9], is less well known. The main contribution of this work is to join the best parts of these existing systems into a single module system that is both simple and powerful.

The modules and interfaces described here are similar to Scheme 48's, as is the separation between the configuration and core languages. The Scheme 48 system is organized around structures: the user manipulates structures instead of modules and importing and exporting is done in terms of structures instead of interfaces. As a result, there is no separate linking phase: linking occurs when a module is created. Parameterized modules are simply procedures that take structures as arguments and return new structures, which in turn requires that the programmer write down the complete module dependency graph.

The module system for Scheme Xerox[4] influenced Scheme 48's system, and thus our system as well. The Scheme Xerox module system does not allow parameterized modules. They mention the need for some way to link modules together, but no details are given.

Ada [11] and Modula-3 [6] use explicit interfaces and a traditional linker, but in much more direct fashion than described here. Within an Ada or Modula-3 program is only one implementation of each interface (although it may be divided across several modules). Linking is implicit and module and interface constructors are declarative, and not generative In effect, an Ada or Modula-3 program consists of a set of modules which are linked using a single call to *link-all.* Modules in Ada and Modula-3 can be parameterized as *generics*, which are considerably more limited that the ML style of functor.

There are two other approaches that have been used to get cross-module type checking while using an existing object-code linker. One is to do the type checking independently of linking and compilation, using a completely separate program. *lint* [7] is a well-known example. The second is to encode type information into variable and function names [5], [12]. Neither of these provides the user with modules. Instead they are techniques for working around the absence of explicit modules and interfaces.

References

[1] Andrew W. Appel and David B. MacQueen. Separate compilation for standard ML. In *Proceedings SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 13–23, 1994.

[2] William Clinger and Jonathan Rees. Macros that work. In *Conf. Rec. 18 ACM Symposium on Principles of Programming Languages*, 1991.

[3] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3):1–55, 1991.

[4] Pavel Curtis and James Rauen. A module system for Scheme. In *Proc. 1990 ACM Symposium on Lisp and Functional Programming*, pages 13–19, 1990.

[5] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[6] Samuel P. Harbison. *Modula-3.* Prentis Hall, Englewood Cliffs, NJ, 1992.

[7] Stephen C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ, 1977.

[8] Mark P. Jones. Using parameterized signatures to express modular structure. In *Conf. Rec. ACM Symposium on Principles of Programming Languages*, pages 68–78, 1996.

[9] Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7:315–335, 1994.

[10] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[11] David J. Naiditch. *Rendezvous with Ada.* John Wiley & Sons, New York , NY, 1989.

[12] Diomidis Spinellis. Type-safe linkage for variables and functions. *SIGPLAN Notices*, 26(8):74–79, August 1991.