Separate Compilation for SCHEME

Matthias Blume
Department of Computer Science
Princeton University

September 2, 1997

Abstract

This paper presents the outline of a module system for the SCHEME programming language in order to solve several problems commonly encountered with contemporary implementations. These problems include the lack of support for separate compilation and cross-module type checking as well as the necessity for non-standard language extensions to aid certain compiler optimizations like open compilation and procedure in-lining.

Even though SCHEME needs to be augmented with concepts like *modules*, *types*, and *signatures*, this will not change the spirit of the language in a major way. Instead it turns out that other recent additions to the language interact smoothly with the proposed module system.

It will be emphasized that it is necessary to make a clean distinction between syntactic definitions and value definition — reflected by a textual separation of *modules* and their *signatures*.

1 Introduction

The performance of compiled SCHEME code suffers from several limitations imposed by the language itself, but most notably from two facts:

- SCHEME is a weakly typed language. In general even compiled code requires run-time type checks.
- An interactive top-level environment permits on-the-fly redefinition of next to everything. This discourages expensive compiler analysis. A highly optimizing compiler needs to know all top-level definitions at once but can deal with them only one at a time. The interactive nature of the top-level environment makes it impossible for a compiler to acquire any information about future definitions. Furthermore, even existing variables can be re-bound or assigned to. This would invalidate assumptions which already have been used in the past.

The first problem can be solved — at least partially — by sophisticated type inference mechanisms [WC93, CF91]. A compiler for SCHEME can take advantage of scoping constructs like `let` or `do` by virtue of the fact that they establish scopes for local variables, the boundaries of which are known at compile-time.
Type inference mechanisms are safe within such boundaries.

An attempt to cope with the second difficulty requires keeping track of all assumptions used by the compiler and of all the implications changes in those assumptions may have. Only very ambitious implementations can afford the costs of this solution. And even then it can happen that a simple assignment triggers a ripple effect which eventually causes a complete recompilation of everything. It can be questioned whether such a system still deserves the attribute "interactive".

The first duty of a SCHEME module system is to provide textually closed entities which contain the equivalent of top-level definitions. The result of evaluating such an entity will be an alternative top-level environment. In comparison with the interactive environment the important differences are:

- The compiler sees all definitions at once.
- The environment is not mutable.

Most previous attempts [Ree93, CR90] to add a module system to SCHEME stop here. Other proposals take a completely different approach. [Tun92] emphasizes interactive programming and distinguishes between *developing* and *developed* modules. *Developing* modules facilitate interactive program development, while *developed* modules are meant to be well-suited for separate compilation.

However, it must be noted that first-class alternative top-level environments alone do not provide support for separate compilation as long as efficiency is of concern. The reasons for this are:

- There is no support for inter-module type inference or inter-module type checking.
- There is no distinction between syntactic definitions and value definitions. The use of a macro embodies a textual alteration of the program text to be compiled and depends upon the actual definition of the macro. If the macro was defined in a different compilation unit, then it is impossible to compile the code where it is used. Thus, separate compilation requires the separation of syntactic definitions from value definitions.

The module system proposed here draws many ideas from other languages, most notably from STANDARD ML [MTH90]. In particular it distinguishes between *modules*[1] and *signatures*.

While modules are first-class objects, which can be thought of as being containers carrying a collection of values, signatures represent the compile-time knowledge about these modules including syntactic extensions and *types*.

2 Types

A type system for SCHEME must be non-intrusive. Many people like SCHEME for the fact that it does not insist in strong typing, and they would not tolerate the introduction of strong typing into the language. A good number of correct SCHEME programs could not even be typed in a traditional strong type system like the one used in SML. Many SCHEME implementations are intended to be inexpensive and small. Those implementations can not afford a complicated type inference mechanism.

Therefore, a type system for SCHEME should have the following properties:

- Type declarations are not mandatory.

  An application developer must always be free to choose the level of accuracy for the type information he or she gives to the compiler. The compiler must be able to produce correct (but perhaps slow) code even from source code with no type information.

- Type-checking and type inference are not mandatory.

[1] *structures* in SML jargon

> An implementation is encouraged but not required to verify or use the type information given by the programmer.

A programmer should provide type information — the more precise the better — because this is likely to speed up program execution in high-quality implementations.

An implementation should check the types and use them to its advantage, because given precise type information this will result in more type safety, earlier error detection, and faster code.

Types in SCHEME

In strongly typed languages it is possible to infer type information for other modules by observing how these modules are used. Therefore, the compilation of one module does not require extra information about any other module [SA93].

In SCHEME it is not possible to acquire reliable type information using this technique. This means that a module system should also be able to provide type information to the compiler. Type information must be separated from actual value definitions.

In some sense types are similar to syntactic definitions. Both of them provide information to the compiler. To be useful both of them must be known at compile-time.

With the emergence of hygienic macro systems [CR91] for SCHEME the language community has become conscious of the distinction between syntactic definitions and value definitions. Syntactic extensions[2] are second-class citizens of the language, and so are types. They can not be passed as arguments to procedures, they can not be stored into data structures.

A collection of types, which correspond to a collection of named values, is called a *signature*. A signature is the type of a module.

[2]macros

The SCHEME type system will be *polymorphic*, and different levels of accuracy will be modeled by different levels of polymorphism. Just like in SML it is acceptable for an imported module to be less polymorphic than the corresponding signature says.

3 An example

Every module consists of a sequence of definitions. An *export signature* describes the definitions which are *public*. Everything defined in the module but not mentioned in its export signature remains *private* and cannot be accessed directly from outside the module.

The `define-type` syntax is used to associate identifiers with types (and signatures).

```
(define-type streams-sig

  (signature
      (nil head tail stream-cons)

    (constant nil any)
    (constant head (proc any -> any))
    (constant tail (proc any -> any))

    (use scheme-sig)

    (define-syntax stream-cons
      (syntax-rules ()
        ((stream-cons x y)
         (cons x (delay y))))))
```

The `signature` specification is the counterpart of a `module` expression. It contains a number of *declarations* specifying the type and *access mode* of identifiers appearing in the module. The only access mode in the example above is `constant`. The other two possible access modes are `variable` and `read-only`.

An *export list* mentions the names of those identifiers, which shall be visible from outside. The sample signature exports the names `nil`, `head`, `tail`, and `stream-cons`.

Implicitly, the `streams-sig` signature also exports the names `cons` and `delay`, which are taken from the `scheme-sig` signature. However, these names will only be valid as part of the output of the macro transformer for `stream-cons`.

The type of `nil` is left unspecified, and the type of both `head` and `tail` is a procedure type with argument and result types left unspecified.

Another signature describes a module of utility procedures. One of them is `compose`. The type of `compose` is polymorphic. It uses three universally quantified type variables `a`, `b`, and `c` and describes a procedure, which takes two procedures as its arguments and returns another procedure as its result.

```
(define-type util-sig
  (signature (... compose ...)
    ...
    (constant compose
      (gen (a b c)
        (proc (proc b -> c)
              (proc a -> b)
           -> (proc a -> c))))
    ...))
```

In a more familiar notation the type of `compose` would be written as:

$$(\beta \to \gamma) \times (\alpha \to \beta) \to (\alpha \to \gamma)$$

A module in SCHEME is a first-class object, which can and will be created at run-time by evaluating a `module` expression. First it mentions its *export signature*. What follows is a sequence of definitions, a subset of which has to satisfy the export signature.

A definition can be an instance of the ordinary `define` syntax as reported in the current SCHEME standard. Furthermore, the new `use` syntax is used to *import* everything exported by another module.[3] The `use` clause mentions an *import signature* to tell the compiler about the imported module.

In the example `scheme` is a *variable* which will *hold* a module at run-time. The export signature of this module must satisfy the import signature `scheme-sig`. This requirement can not be checked before module creation time (i.e. it must be checked at run-time).

```
(define util
  (module util-sig
    (use scheme scheme-sig)
    ...
    (define (compose f g)
      (lambda (x) (f (g x))))
    ...))
```

The second form of the `use` syntax allows for a more selective way to import values from another module. It has an additional (third) parameter, which must be an identifier. Like the two-argument `use` form it verifies that the actual module (first argument) satisfies the import signature (second argument). But it also arranges for the third argument to act as special syntax from now on. After a successful `(use x y z)` one can write `(z w)` to refer to the variable `w` in module `x`. In the example this has been done with `(u compose)` to refer to the `compose` procedure of module `util`.

```
(define streams
  (module streams-sig
    (use scheme scheme-sig)
    (use util util-sig u)
    (define nil '())
    (define head car)
    (define tail
```

[3]Strictly speaking, this is not true. The `define` syntax is initially unknown, but an appropriate `use` clause can make it available.

```
    ((u compose)
     force cdr))))
```

The only free variables of a `module` expression are the free variables which appear in the first arguments of at least one of the various `use` clauses.

4 New bits and pieces

The module system adds the following new keywords to the language: `module`, `use`, and `define-type`. In addition, a sublanguage for the description of types is introduced. This language does not add new keywords to SCHEME, because its use is restricted to instances of `module`-, `use`-, or `define-type`-clauses, respectively. Types do not occur in the place of ordinary expressions. Therefore, the expression language can not be confused with the type language.

The `module` expression

Modules in SCHEME are objects in their own right, which can and must be created by expression evaluation.

⟨expression⟩ $\rightarrow$ ⟨module expression⟩

The `module` keyword is used to specify a `module` expression.

⟨module expression⟩
$\rightarrow$ (**module** ⟨signature⟩ ⟨program⟩)

The processing of definitions and commands in a `module` expression's ⟨program⟩ proceeds strictly from left to right. Evaluation takes place in an environment, which contains bindings for precisely those variables, for which there is a definition in the module's body. The exception to this rule is the `use` clause, the first argument of which will always be evaluated in the environment the current `module` expression appears in.

The only a-priori known syntax inside a module is the `use` form. Everything else must be defined. In practice this means that the first form in a `module` expression must be an instance of `use`, which usually imports some standard module in order to make the full syntax of SCHEME available. The definitions in a module's body must be consistent in the same way, in which local definitions at the beginning of a `lambda` expression's ⟨body⟩ must be consistent.

It is possible to have more than one definition for the same name in a module's body as long as the meaning of the resulting construct is uniquely defined (and not contradictory). This means that one can import a name via `use` and provide a different definition for the same name afterwards, thus providing a variation of the original. The occurrence of an identifier refers to the definition directly preceding this occurrence. If a name appears before its first definition, then the *last* definition for this name within the module is used. The mention of an identifier in a module's export signature counts as a use of that name. Therefore, the exported names of a module will always refer to their last definition.

The ⟨signature⟩ given as the first parameter to the `module` expression specifies those names defined in the module which can be made accessible from outside the module via `use`. However, a module might *implicitly* export more than that. This is, because the output of a macro transformer can refer to names defined in the module, which are *not* part of the export list. These names are exported as well. The implementation of the macro system has to make sure that they will not become part of the importing environment but will be valid in the output of the macro transformer only.

The `use` clause

Formally, an instance of the `use` clause is a *definition*.

⟨definition⟩ → ⟨use clause⟩

The clause has two different forms, the first of which is used to import everything exported by a module at once.

⟨use clause⟩
  → (**use** ⟨expression⟩ ⟨signature⟩)

Evaluation of ⟨expression⟩ takes place in the environment *surrounding* the current module and must yield a module object. Likewise, ⟨signature⟩ is analyzed in the lexical environment in which the current `module` expression is evaluated. The `use` clause then incorporates all definitions of the module, which are listed in ⟨signature⟩, into the current environment.

For the sake of convenience the interactive environment is treated as a special module, which *surrounds itself*. This makes the following program fragment valid when it is typed to the SCHEME top-level:

```
(define-type sig ...)
(define mod (module sig ...))
(use mod sig)
```

This fragment will *not* be valid inside any other module.[4]

The second form of the `use` clause imports elements from another module without unnecessarily cluttering the name space.

⟨use clause⟩
  → (**use** ⟨expression⟩ ⟨signature⟩
      ⟨identifier⟩)

[4]It must be noted that the interactive environment is special in several ways, because unlike all other modules it is mutable.

Here, ⟨expression⟩ and ⟨signature⟩ are evaluated just as with the first version of `use`, but no definition from the imported module gets incorporated into the current environment. Instead, ⟨identifier⟩ will be established as special syntax which allows to selectively access elements from the imported module.

Consider:

```
(define-type sig
  (signature (f) (constant f any)))
(define mod
  (module sig
    (use scheme scheme-sig)
    (define (f x) (+ x 1))))
(use mod sig m)
((m f) 2)
```

The expression on the last line will yield `3`, because `(m f)` refers to the variable `f` in module `mod`.[5]

The `define-type` declaration

Like `use` clauses, occurrences of `define-type` are definitions.

⟨definition⟩ → ⟨type definition⟩

Although it is possible to specify types explicitly using the type specification language described later, it is more convenient to give names to frequently recurring types. Type names are similar to syntactic definitions, because they add syntax to the type description language. It is possible to use a separate name space for types, because with the proposed syntax type names can never be confused with other names.

⟨type definition⟩
  → (**define-type** ⟨identifier⟩ ⟨type⟩)

[5]As mentioned before this example will only work in the interactive environment.

A type defined with `define-type` can also be parametric.

```
⟨type definition⟩
      → (define-type
            (⟨identifier⟩
                ⟨identifier⟩ ⟨identifier⟩ ...)
            ⟨type⟩)
```

Like other definitions, type definitions are analyzed in a syntactic environment where all definitions of a ⟨body⟩ or a ⟨program⟩ are in effect. Therefore, `define-type` can be used to define recursive or mutually recursive types.

The type language

This section of the paper is intended to give a brief overview of how the type language could be structured. A detailed specification certainly requires discussion in the SCHEME language community. Since it is intended to communicate type information which might be used by existing type inference mechanisms like [WC93], it must be coordinated with those mechanisms, should they ever become widespread in use.

The early motivation to design the module system presented in this paper was to support optimization of the procedure calling protocol. In particular, eliminating most of the arity-checking code could noticeably improve overall performance. Therefore, this presentation will restrict itself to a type system, the only "structured" types of which are procedure types and signatures. It is anticipated that in the future this will be extended to a full-fledged type system similar to the one used in strongly typed languages. Nevertheless, the momentary restriction does not limit the domain of values, which can be exported or imported by modules, even though it limits the accuracy of type information the programmer can provide. A type, which cannot be expressed with full precision, can always be replaced by a type variable.

One can distinguish between *signatures* and *ordinary types*.

```
⟨type⟩ → ⟨signature⟩
⟨type⟩ → ⟨ordinary type⟩
```

Simple type names

Simple type names are identifiers which either have been defined by a (non-parametric) `define-type` or which belong to a set of predefined built-in types such as `number`, or `string`. Simple type names are specified by writing down the corresponding identifier:

```
⟨ordinary type⟩
      → ⟨identifier⟩
```

Parametric type names

Unless they are built-in, parametric types have been defined by a parametric `define-type`. An example of a possibly built-in parametric type is `list`, which describes a strongly typed subset of SCHEME's proper lists. Parametric type names must be specified together with appropriate *type parameters*:

```
⟨ordinary type⟩
      → (⟨identifier⟩ ⟨type⟩ ⟨type⟩ ...)
```

Type variables

Type variables are identifiers, which are universally quantified. Therefore, the only syntactic distinction between simple type names and type variables is that for the latter there is an enclosing `gen` specification.[6]

[6]The necessity to explicitly generalize over type variables stems from the lack of *lexical* distinction between type variables and other symbols. Languages like SML use a separate set of symbols (lower-case greek letters) for type variables. The type generalization operator is implicit.

⟨ordinary type⟩
→ ⟨identifier⟩

It is an error for an identifier, which is used as a type, not to be defined as a simple type name and not to be universally quantified.

Universal quantification

The `gen` syntax is used to universally quantify a list of type variables.

⟨ordinary type⟩
→ (**gen** (⟨identifier⟩ ⟨identifier⟩ ...)
⟨type⟩)

Procedure types

⟨ordinary type⟩
→ ⟨procedure type⟩

A procedure type lists number and types of the arguments as well as the type of the result. Let's start with simple procedure types:

⟨procedure type⟩
→ (**proc** ⟨type⟩ ... `->` ⟨type⟩)

The type of `lambda` expressions, which use the familiar dot-notation to accept a variable number of arguments can be specified with an ellipsis:

⟨procedure type⟩
→ (**proc** ⟨type⟩ ... ⟨ellipsis⟩ `->` ⟨type⟩)

⟨ellipsis⟩ → `...`

Note that ⟨ellipsis⟩ is not used to repeat the preceding type. Instead, it tells that there can be an arbitrary number of extra arguments of arbitrary types.[7]

[7]This seems to be a restriction, and in fact it is. However, the repetition of a type would imply that all the extra arguments must *share* the same type. In many cases this is not adequate.

It is conceivable that the same kind of syntax used *in front of* `->` would serve the purpose of indicating procedures with multiple return values when used *after* `->`. For the sake of simplicity this paper refrains from adopting this idea.

Module types — Signatures

Just as with other types one can refer to signatures by using a name. A SCHEME implementation will have a number of predefined signatures (like `scheme-sig` in above's example) available.

⟨signature⟩
→ ⟨identifier⟩
⟨signature⟩
→ (⟨identifier⟩ ⟨type⟩ ⟨type⟩ ...)

Furthermore, it is possible to specify new signatures.

⟨signature⟩
→ (**signature**
(⟨identifier⟩ ⟨identifier⟩ ...)
⟨declaration⟩ ⟨declaration⟩ ...)

The list of identifiers specifies the names explicitly exported by the signature. When a module is created, then then the top-level environment of the module is "filtered" through its export signature. Later, when the module is `use`d, then it is "filtered" further, namely through the signature mentioned in the `use` clause. Only those names, which are listed in both import and export signatures will become available.

A signature can *declare* more names than it exports. This is necessary to allow for macro definitions to refer to names, which should not be accessible otherwise. A common method to define `delay` in terms of `lambda` uses an auxiliary procedure `make-promise`. The macro definition for `delay` must refer to `make-promise`.

However, the name `make-promise` should not be visible from outside, even in contexts, where `delay` was imported.

The declarations inside a signature's body can be type definitions, macro definitions, or *value declarations*.

⟨declaration⟩ → ⟨type definition⟩
⟨declaration⟩ → ⟨syntactic definition⟩
⟨declaration⟩ → ⟨value declaration⟩

Value declarations give access mode, name, and type of a variable symbol.

⟨value declaration⟩
    → (⟨access mode⟩ ⟨identifier⟩ ⟨type⟩)
⟨access mode⟩ → **constant**
⟨access mode⟩ → **variable**
⟨access mode⟩ → **read-only**

The `use` declaration

Modules import *values* from other modules via `use`. An almost identical syntax permits to include *declarations* from other signatures.

⟨declaration⟩ → ⟨use declaration⟩

Again, the `use` declaration has two forms: one which includes everything exported by the signature, and one which declares a handle to allow for selectively accessing individual elements from the signature.

⟨use declaration⟩
    → (**use** ⟨signature⟩)
⟨use declaration⟩
    → (**use** ⟨signature⟩ ⟨identifier⟩)

The purpose of the `use` declaration is to augment the syntactic environment with elements from another signature. However, while it is necessary for a module to first `use` some other module to make the SCHEME language itself available, this is not the case for signatures. Signatures inherit the syntactic environment in which they are declared, but only those names appearing in the signature's export list are relevant for their use in `module` expressions.

The main purpose of `use` declarations is to establish a well-defined syntactic environment even in situations, where it is not clear what the enclosing environment will look like.[8]

Access modes

constant

The `constant` access mode notifies the compiler that a variable's value will never be changed. It is the intention that this mode will be used most frequently in practice. The import of such a variable can be implemented by simply copying its value.

variable

The other extreme is the `variable` mode. It allows for variables which can be assigned to. This will work even outside the module where they are defined. The `variable` mode is similar to the `ref` type of SML. It is a well-known fact that there are some fundamental difficulties with references to polymorphic types [LW91].

Consider:

```
(define-type s1
  (signature (x)
    (variable x number)))
(define m
  (module s1
    ...
    (define x 1)))
```

[8]Signatures defined in separate files can be `load`ed into the interactive environment. But the interactive environment is mutable.

```
(define-type s2
  (signature (x)
    (variable x (gen (a) a))))
```

and now

```
  ...
  (module ...
    (use m s2)
    ...
    (set! x "Hello, world!")
    ...)
```

The assignment to `x` seems to satisfy the type checker, because `s2` lists `x` as being of unspecified type. However, according to `s1` `x` really is a number.

There are two possible solutions to this problem[9]:

1. The type of a name with access mode `variable` must always be `any`.[10]

2. The types in `variable` declarations can not be polymorphic.

read-only

The `read-only` access mode can be thought of as being a compromise between `constant` and `variable`. It indicates that the variable's value may very well change, but assignments are not allowed from outside the module where it was defined. A module, which imports a read-only variable, can rely on the type information given in the declaration. The problems with polymorphic references do not arise here.

Compatibility

In general it is possible that the import signature lists an access mode which differs from the one given in the export signature. At module creation time it is necessary to verify that they are compatible.

First of all, `variable` is only compatible with `variable`. Obviously, a module can not treat an imported name as `variable`, if it really is not. On the other hand, if the name is actually exported as `variable`, then other modules are entitled to modify it. While `constant` is out of question anyway, the problem with polymorphic references rules out the use of `read-only` in an import signature as well.
The `variable` access mode must be considered a kludge to allow for mutable variables. Its use should be minimized.

A rather straight-forward relationship exists between the other two modes. A `read-only` declaration matches another `read-only` just as `constant` is compatible with `constant`. Furthermore, a variable exported as `constant` can be imported `read-only`,[11] while obviously the converse can not be true.

The any specification

The `any` syntax adds a convenience notation to the type language. First of all, `any` can appear in place of an ordinary type:

$\langle\textsf{ordinary type}\rangle \rightarrow \textbf{any}$

In this case `any` acts like a fresh type variable, with an implicit universal quantifier in place. One can think of `any` as being equivalent to `(gen (a) a)`.

A parametric version of `any` is used as a shorthand for signatures.

$\langle\textsf{signature}\rangle$
$\quad\rightarrow (\textbf{any}\ \langle\textsf{identifier}\rangle\ \langle\textsf{identifier}\rangle\ \ldots)$

[9]It is possible to use less limiting restrictions, but this paper will not discuss them.

[10]At least, it must be treated that way by a compiler. For an explanation of `any` see below.

[11]This can be implemented by introducing a dummy reference cell at module creation time.

This notation serves as an abbreviation for the `signature` form, which declares and exports all the identifiers listed in the clause with type `any`[12] and access mode `read-only`.

For example, one can write:

```
(define m
  (module (any f g h)
    (use scheme scheme-sig)
    (define (f x) x)
    (define (g x) (x f))
    (define (h x) (g (x g)))))
```

or:

```
(define (functor input-module)
  (module (signature (val)
             (constant val int))
    (use scheme scheme-sig)
    (use input-module (any f))
    (define val (f 2))))
```

5 Signature matching

So far something has been said about the matching of *access modes* in signatures. This section deals with the problem of *type matching*.

The new proposal does not add strong typing to SCHEME. Therefore, it is legal for a module's implementation to be in conflict with the type declarations. In the case of doubt it is the compiler's responsibility to insert explicit run-time type checks into the object code. A low-quality implementation can always ignore all types.

There are three places, where different parts of a SCHEME program have to negotiate types:

1. a module's implementation vs. its export signature
2. an export signature of a module vs. the import signature of the `use` clause, where the module is imported
3. the import signature of a `use` clause vs. the use of the imported names in the importing module's implementation

[12]This refers to the first form of `any`.

Cases 1 and 3 are not important for the discussion of the module system itself. However, they are relevant, when it comes to implementing a compiler. It is anticipated that type inference mechanisms use the type information given by the programmer in order to remove unnecessary type checks in the object code.

In all cases there is always the possibility of *thinning*. A module can define more values than it exports. An import signature can select fewer elements for import than the export signature had listed. And finally, an importing module does not have to use everything mentioned by the import signature.

Syntactic definitions are a different matter. Only those syntactic definitions, which appear in a module's import signature, can be seen by the compiler. Therefore, the exporting module does not need to mention the same macros. In fact, it is not even necessary for the exporting module to define *any* macros. However, the exporting module must make those values available for implicit import, which are required by the import signature. Implicit import of a definition exported explicitly is permitted, but the converse of that is not true.

Everything mentioned *directly* or *indirectly* in the export signature's export list will be available for implicit import.[13] Macros exported by a module are of no further use, but they can serve the purpose of mentioning names, which shall be available for implicit export.

Example:

```
(define mod
  (module
    (signature
```

[13]An indirect mention of a name is the direct or indirect mention of a macro, which depends on it.

```
      (nil head tail stream-cons)
      (constant nil any)
      (constant head
        (proc any -> any))
      (constant tail
        (proc any -> any))
      (use scheme-sig)
      (define-syntax
        stream-cons
         (syntax-rules ()
           (_) (delay cons))))
    ...))
```

With this definition in effect one can write:

```
(module ...
  (use mod
    (signature
      (nil head tail stream-cons)
      (constant nil any)
      (constant head
        (proc any -> any))
      (constant tail
        (proc any -> any))
      (use scheme-sig)
      (define-syntax stream-cons
        (syntax-rules ()
          ((stream-cons x y)
           (cons x (delay y)))))))
  ...)
```

The macro definitions do not coincide. Instead, the exported `stream-cons` uses a fake definition in order to make `delay` and `cons` available.[14]

The compiler can expect that imported modules hold the promises given by the corresponding import signatures. Similarly, the compiler must make sure that a module faithfully implements the interface specified by its export signature. The module system is structured in a way that the compiler can do this by merely looking at textually closed entities.

It is case 2, which can not be statically analyzed by the compiler. Therefore, it must be implemented in a way such that no further type checks can ever become necessary due to signature mismatches.

To achieve this goal the Scheme module system adopts a rule, which is reminiscent of the rule governing the matching of structures and signatures in SML. In particular, a module must restrict its values to a domain, which is smaller[15] than the one specified by the import signature.

Consider:

```
(define m
  (module
    (signature (n)
      (constant n number))
    (define n 2)))

(module ...
  ...
  (use m (signature (n)
           (constant n any)))
  ...)
```

This code is allowed, because the importing module expected `n` to be anything, which includes the *possibility* of being a number. If one was to exchange the roles of the two signatures, then the fragment would be illegal, triggering a type error during creation of the second module.

Here is another example illustrating some more cases:

```
(define m
  (module
    (signature (f g h)
      (constant f
```

[14]This is not supposed to be common practice. It only serves the purpose of explaining the rules governing export and import of definitions.

[15]I.e., it must be a proper or improper subset.

```
        (proc any -> number))
      (constant g
        (gen (a) (proc a -> a)))
      (read-only h
        (gen (a)
          (proc (proc number -> a)
                -> a)))))
  ...)
```

With this definition in effect it is now legal to write:

```
(module ...
  ...
  (use m
    (signature (f g h)
      (constant f
        (proc number -> number))
      (read-only g
        (proc number -> number))
      (read-only h
        (gen (b)
          (proc (proc any -> b)
                -> b)))))
  ...)
```

A quite simple modified unification algorithm can be used to verify that an export signature satisfies the corresponding import signature. However, it must be noted that this unification must be carried out *at run-time* — during module creation.

6 Other considerations

It has been promised that the new module system attempts to solve a variety of problems. Until now the paper has mostly been concerned with the issue of separate compilation and type inference.

The remainder of this section will give a brief overview of which other problems can be tackled, and of how this is actually done.

Optimization

Type information and textually closed compilation units already provide a great deal of more freedom to an optimizing compiler. Type inference mechanisms now work within modules and across module boundaries. Furthermore, the compiler can perform traditional optimizations such as procedure in-lining on an intra-module basis.

What's left to be desired is the so-called "open compilation" of SCHEME's built-in procedures. For instance, many existing SCHEME systems tacitly assume that the symbol `car` always refers to the built-in procedure, which returns the first element of a list.[16]

The addition of a few standard signatures and modules can now cure the problem in a much more systematic way. The examples above already made use of the built-in signature `scheme-sig` and the corresponding module `scheme`. The idea is that `scheme-sig` refers to a "magical" built-in signature, which can only be matched by the `scheme` module. Indirectly this trick tells the compiler that the imported module *will* in fact be "standard SCHEME".[17] Open compilation of calls to `car`, `cdr`, or any other built-in procedure is obviously legal then.

Some people might argue that not the *signature*, but the *module* ought to be the "magical" thing, because then a different module with a, let's say, bookkeeping version of `car` can be "plugged in" easily. However, this is in direct contradiction with the main objective: support for separate compilation. Open compilation is something the compiler needs to know about. The signature is the language construct dedicated to carrying information relevant to the compilation process.

[16]Usually, there is a way to turn this "feature" off. The result is a semantically even more inconsistent system.

[17]whatever this might be

There can be more signatures, which allow to import `scheme`, and there can be other built-in modules:

- Signature `noc-scheme-sig` allows to import `scheme`, but it prohibits open compilation.
- Module `r4rs` and signatures `r4rs-sig` as well as `noc-r4rs-sig` refer to SCHEME as described in [Ce91].
- Pre-defined names like
  - `r3rs`,
  - `r3rs-sig`,
  - `noc-r3rs-sig`,
  - `ieee-scheme`,
  - `ieee-scheme-sig`,
  - and `noc-ieee-scheme-sig`

  do the same trick for a SCHEME conforming to other versions of the standard.
- ...

Preferring signatures over modules to carry information about open compilation still offers the opportunity to use different implementations for a module without requiring extensive changes to the program text. This will be illustrated in the following example.

First, consider the definition of a module residing in file `a.scm`:

```
(define module-a
  (module ...
    (use scheme-a scheme-sig-a)
    ...))
```

Note that `module-a` uses `scheme-sig-a` instead of `scheme-sig`. File `a.scm` can now be loaded (i.e. compiled) in different contexts — with different definitions for `scheme-sig-a` in effect.

During program development it could be desirable to write:

```
(define-type
  scheme-sig-a noc-scheme-sig)
```

The signature `noc-scheme-sig` allows module `scheme-a` to be a suitable substitute for `scheme`. Therefore, it is now easy to install a bookkeeping version of certain functions.

```
(define scheme-a
  (module noc-scheme-sig
    (use scheme scheme-sig)
    (define orig-car car)
    (use bookkeeper bookkeeper-sig)
    (define (car x)
      (let ((result (orig-car x)))
        (register-call!
          orig-car x result)
        result))))

(load "a.scm")
```

Note that this module contains *two* definitions for `car`, while only the second, modified version will be exported. The definition of `orig-car` refers to the one imported from `scheme`.

A fictitious "bookkeeping" module provides the definition of `register-call!`.

Later, in production quality code, both signature declaration and module definition can be replaced.

```
(define-type
  scheme-sig-a scheme-sig)
(define scheme-a scheme)
(load "a.scm")
```

The contents of file `a.scm` itself does not even have to be touched.

Functors

SML uses a feature called *functors* to allow for modules, which can be parameterized with other modules. In effect, functor application corresponds to the *link phase* of traditional systems like C compilers. Due to theoretical subtleties with the type system there is no notion of *higher order* functors in SML.[18]

Modules in SCHEME are first-class objects. They are created at run-time, where they are automatically parameterized via `use`. It is easy to create "functors" as ordinary SCHEME procedures, which return a module.

SCHEME's type system does not attempt to be *sound*. A program, which "satisfies" the type checker can still break at run-time. Consequently, the difficulties with the type system, which occur in the case of SML, do not arise here.

Even higher-order "functors" are covered by this proposal, because "functors" are just procedures, and procedures do not have to be first-order.

Modules, environments, and `eval`

User demand seems to force a re-introduction of `eval` into the language. There have been many arguments about the relative merits of such a construct. More importantly, there has been some discussion about its *semantics*.

In the fifth revision of the "Report of the Algorithmic Language Scheme", which has yet to appear [Ree92], `eval` will be a procedure. It takes two arguments: the symbolic expression to evaluate and an *environment specifier*. This second argument is used to denote the environment the expression is to be evaluated within.

There will be a few built-in environment specifiers, but no systematic way to create new specifiers is mandated by the new report. In particular, an implementation does not need to support the reification of local environments like the ones created by scoping constructs.

[18]Meanwhile, the theory of higher-order functors has been worked out, and there is an implementation in SML/NJ. The semantics of higher-order functors are explained in [MT].

It has been hinted that *modules* can be regarded as top-level environments, or rather as top-level environment *specifiers*. The new report will probably list three procedures, which can be used to get hold of certain built-in environments. This paper suggests that there should be a number of built-in modules. The relationship between these two things can hardly be overlooked:

```
(scheme-report-environment 4)
```

returns the same object, which is also the value of the variable `r4rs`. A similar relationship exists for

```
(scheme-report-environment n)
```

and `r`*n*`rs`. Furthermore, the value of

```
(interactive-environment)
```

would be the forementioned special module, which was said to implement the concept of SCHEME's (mutable) top-level environment.

The expression *e* in (`eval` *e m*) will be evaluated in the lexical environment specified by module *m*'s export signature.

References

[Ce91] William Clinger and Jonathan Rees (editors). Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):278–292, June 1991.

[CR90] Pavel Curtis and James Rauen. A Module System for Scheme. In *1990 ACM Conference on Lisp and Functional Programming*, pages 13–19, 1990.

[CR91] William Clinger and Jonathan Rees. Macros That Work. In POPL'91 [POP91], pages 155–162.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In POPL'91 [POP91], pages 291–302.

[MT] David B. MacQueen and Mads Tofte. A Semantics for Higher-order Functors.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[POP91] *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[Ree92] Jonathan Rees. The Scheme of Things: The June 1992 Meeting, June 1992.

[Ree93] Jonathan Rees. Another Module System for Scheme, January 1993.

[SA93] Zhong Shao and Andrew Appel. Smartest Recompilation. In *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450, 1993.

[Tun92] Sho-Huang Simon Tung. Interactive Modular Programming in Scheme. In *1992 ACM Conference on Lisp and Functional Programming*, pages 86–95, 1992.

[WC93] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. Technical Report TR93-218, Department of Computer Science, Rice University, December 1993.