Extending the Scope of Syntactic Abstraction*

Oscar Waddell
University of Kansas
owaddell@ittc.ukans.edu

R. Kent Dybvig
Indiana University
dyb@cs.indiana.edu

Abstract

The benefits of module systems and lexically scoped syntactic abstraction (macro) facilities are well-established in the literature. This paper presents a system that seamlessly integrates modules and lexically scoped macros. The system is fully static, permits mutually recursive modules, and supports separate compilation. We show that more dynamic module facilities are easily implemented at the source level in the extended language supported by the system.

1 Introduction

The benefits of module systems and lexically scoped syntactic abstraction (macro) facilities are well-established [1, 2, 3, 6, 7, 8, 12, 10, 11, 15, 16, 18]. Over the past several years there has been increasing interest in combining lexically scoped macros and modules [2, 16, 17]. Building on this work we present here the first fully implemented system that allows arbitrary composition of module and macro facilities, supports separate compilation, and supports fully general macro transformations while maintaining lexical scoping for all macros.

Our design derives from the philosophy that a programming language should be based on a small core language augmented by a powerful syntactic abstraction facility. The core language should have simple constructs and a straightforward semantics, and the syntactic abstraction facility should permit the definition of new language constructs whose meanings can be understood in terms of their static translation into the core language.

Our system extends the small core language and powerful syntactic abstraction mechanisms of the `syntax-case` system [4, 6] with support for modules. A module is a named scope that encapsulates a set of identifier bindings. Importing from a module makes these identifier bindings visible in the importing context. Modules control visibility of bindings and can be viewed as extending lexical scoping to allow more precise control over where bindings are or are not visible. Modules export identifier bindings, i.e., variable bindings, keyword bindings, or module name bindings. The primitive module and import forms are simple, may be extended via syntactic abstraction, and may appear at top level, within local scopes, or nested within other modules. A program containing module and import forms may be understood statically in terms of a straightforward translation to core language forms.

The remainder of this paper is organized as follows. Section 2 describes an extended language that supports both modules and syntactic abstraction. Section 3 demonstrates the synergy between modules and syntactic abstraction with a series of short examples. Section 4 describes the implementation. Section 5 addresses practical concerns such as separate compilation and supporting static program analysis in the presence of powerful macro systems. Section 6 discusses related work, and Section 7 presents our conclusions.

2 Language

2.1 Syntax

The core language of the `syntax-case` system is the core language of the Revised[5] Report on Scheme [9]. In the core language, a program consists of a sequence of definitions and expressions.

⟨program⟩ ⟶ ⟨form⟩`*`
⟨form⟩ ⟶ ⟨definition⟩ | ⟨expression⟩

A definition is a variable definition or a sequence of definitions.

⟨definition⟩ ⟶ `(define` ⟨variable⟩ ⟨expression⟩`)`
  | `(begin` ⟨definition⟩`*)`
⟨variable⟩ ⟶ ⟨identifier⟩

An expression is a constant, a quoted datum, a variable reference, an abstraction, an application, a conditional, or an assignment.

⟨expression⟩ ⟶ ⟨constant⟩
  | `(quote` ⟨datum⟩`)`
  | ⟨variable⟩
  | ⟨lambda expression⟩
  | `(`⟨expression⟩ ⟨expression⟩`*)`
  | `(if` ⟨expression⟩ ⟨expression⟩ ⟨expression⟩`)`
  | `(set!` ⟨variable⟩ ⟨expression⟩`)`
⟨lambda expression⟩ ⟶ `(lambda` ⟨formals⟩ ⟨body⟩`)`

*This material is based on work supported in part by the National Science Foundation under grants CCR-9623753 and CCR-9711269.

To appear in the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1999, San Antonio, Texas.

A `lambda` body consists of a sequence of of zero or more definitions followed by one or more expressions.

```
⟨body⟩ ⟶ ⟨definition⟩* ⟨expression⟩+
```

Thus, definitions may appear both at the top level of a program or nested within a `lambda` body.

Support for syntactic abstraction is provided by extending the set of definition forms to include both keyword definitions and definitions derived via syntactic abstraction, i.e., macro calls that expand into definitions.

```
⟨definition⟩ ⟶ (define ⟨variable⟩ ⟨expression⟩)
    | (define-syntax ⟨keyword⟩ ⟨expression⟩)
    | (begin ⟨definition⟩*)
    | ⟨derived definition⟩
⟨keyword⟩ ⟶ ⟨identifier⟩
⟨derived definition⟩ ⟶ ⟨macro call⟩
```

In addition, the set of expression forms is extended to include derived expressions.

```
⟨expression⟩ ⟶ ⟨constant⟩
    | (quote ⟨datum⟩)
    | ⟨variable⟩
    | ⟨lambda expression⟩
    | (⟨expression⟩ ⟨expression⟩*)
    | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
    | (set! ⟨variable⟩ ⟨expression⟩)
    | ⟨derived expression⟩
⟨derived expression⟩ ⟶ ⟨macro call⟩
```

As in the core language, definitions may appear at top level or nested within `lambda` bodies. They may also appear nested within the bodies of derived expressions that expand into `lambda` expressions, such as `let`, which expands as shown below.

```
(let ((x e) ...) form1 form2 ...) ⟶
  ((lambda (x ...) form1 form2 ...) e ...)
```

The language is further extended to support modules by the addition of two new definition forms, `module` and `import`.

```
⟨definition⟩ ⟶ (define ⟨variable⟩ ⟨expression⟩)
    | (define-syntax ⟨keyword⟩ ⟨expression⟩)
    | (begin ⟨definition⟩*)
    | ⟨derived definition⟩
    | (module ⟨module name⟩ ⟨interface⟩
        ⟨definition⟩* ⟨expression⟩*)
    | (import ⟨module name⟩)
⟨interface⟩ ⟶ (⟨identifier⟩*)
⟨module name⟩ ⟶ ⟨identifier⟩
```

A module consists of a (possibly empty) set of definitions and a (possibly empty) sequence of initialization expressions. Each identifier listed in a module's interface must be defined within that module. Because `module` and `import` are definitions, they may appear wherever other definitions may appear: at top level, nested within a `module` or `lambda` body, or nested within derived forms that expand into `module` or `lambda` forms.

Although the language does not provide primitive constructs for separating interfaces from implementations, importing only selected identifiers from a module, or renaming exported identifiers, Section 3 demonstrates how such constructs may be obtained via syntactic abstraction.

The syntax of macro calls traditionally subsumes that of variable reference only in the operator position of an application. We generalize the concrete syntax of macro calls to include other contexts in which an identifier reference may occur.

```
⟨macro call⟩ ⟶ (⟨keyword⟩ ⟨form⟩*)
    | ⟨keyword⟩
    | (set! ⟨keyword⟩ ⟨expression⟩)
    | (import ⟨keyword⟩)
```

Consequently, the syntax of macro calls subsumes that of variable reference, variable assignment, and module import, resulting in a significant increase in expressive power.

2.2 Scope

Except where shadowed, identifiers defined at top level are visible throughout the program, while identifiers defined within a body are visible only within that body.

A `module` definition introduces a named lexical scope. Module names occupy the same namespace as other identifiers and follow the same scoping rules. Unless exported, identifiers defined within a module are visible only within that module. Identifiers exported from a module are visible within the module and where the module is imported. A module is imported via an `import` of the module's name. An identifier made visible by `import` is scoped as if its definition appears where the `import` form appears. The following example illustrates these scoping rules.

```
(let ((x 1))
  (module M (x setter)
    (define-syntax x (identifier-syntax z))
    (define setter (lambda (x) (set! z x)))
    (define z 5))
  (let ((y x) (z 0))
    (import M)
    (setter 3)
    (list x y z)))  ⇒  (3 1 0)
```

The inner `let` expression binds `y` to the value of the `x` bound by the outer `let`. The import of `M` makes the definitions of `x` and `setter` visible within the inner `let`. In the expression `(list x y z)`, `x` refers to the identifier macro exported from `M`, while `y` and `z` refer to the bindings established by the inner `let`. The identifier macro `x` (defined using the syntactic abstraction `identifier-syntax`) expands the reference to `x` into a reference to the variable `z` defined within the module.

2.3 Semantics

Before it is compiled, a source program is translated into a core language program containing no syntactic abstractions, syntactic definitions, module definitions, or import forms. Translation is performed by a *syntax expander* that processes the forms in the source program via recursive descent. The semantics of the core language is defined in the Revised^5 Report on Scheme.

A `define-syntax` form associates a keyword with a transformer in a translation-time environment. When the expander encounters a keyword, it invokes the associated transformer and reprocesses the resulting form. A `module` form associates a module name with an interface. When the expander encounters an `import` form, it extracts the corresponding module interface from the translation-time environment and makes the exported bindings visible in the scope where the `import` form appears.

Definitions within a `lambda` or `module` body are processed from left to right so that a module's definition and import may appear within the same sequence of definitions. Expressions appearing within a body and the right-hand sides of variable definitions, however, are translated only after the entire set of definitions has been processed, allowing full mutual recursion among variable and syntactic definitions.

Module and import forms affect only the visibility of identifiers in the source program, not their denotations. In particular, variables are bound to locations whether defined within or outside of a module, and `import` does not introduce new locations.

Local variables are renamed as necessary to preserve the scoping relationships established by both modules and syntactic abstractions. Thus, the module program in Section 2.2 is equivalent to the following program in which identifiers have been consistently renamed as indicated by subscripts.

```
(let ([x_0 1])
  (define-syntax x_1 (identifier-syntax z_1))
  (define setter_1 (lambda (x_2) (set! z_1 x_2)))
  (define z_1 5)
  (let ([y_3 x_0] [z_3 0])
    (setter_1 3)
    (list x_1 y_3 z_3)))
```

Both programs are equivalent to the core-language program below in which syntactic abstractions have been expanded.

```
((lambda (x_0)
  (define setter_1 (lambda (x_2) (set! z_1 x_2)))
  (define z_1 5)
  ((lambda (y_3 z_3)
     (setter_1 3)
     (list z_1 y_3 z_3))
   x_0
   0))
 1)
```

The mechanisms by which expansion and renaming occur are described in Section 4.

Although definitions within a `lambda` or `module` body are processed from left to right by the expander, the order of evaluation of variable definitions is not specified. Initialization expressions appearing within a `module` body are evaluated in sequence after the evaluation of the variable definitions.

3 Examples

The examples in this section show that many useful module constructs can be derived from the primitive `module` and `import` forms via syntactic abstraction.

3.1 Qualified reference

It is often convenient to refer to one export of a module without importing all of its exports. While our system does not provide an explicit construct for this purpose, a qualified reference macro can easily be defined and used as follows.

```
(define-syntax from
  (syntax-rules ()
    ((_ M id) (let () (import M) id))))

(let ((x 10))
  (module m1 (x) (define x 1))
  (module m2 (x) (define x 2))
  (list (from m1 x) (from m2 x)))  ⇒  (1 2)
```

3.2 Anonymous modules

Often it is convenient to avoid naming a module that is imported only in the block where it is defined. The following macro supports *anonymous* as well as named modules. An anonymous module definition expands into a sequence containing the definition and immediate import of an equivalent module named `tmp`. Because the macro expander automatically renames introduced identifiers, the name `tmp` is visible only to the `import` form introduced by the macro.

```
(define-syntax module*
  (syntax-rules ()
    [(_ (id ...) form ...)
     (begin
       (module* tmp (id ...) form ...)
       (import tmp))]
    [(_ name (id ...) form ...)
     (module name (id ...) form ...)]))
```

3.3 Selective import and renaming

While the `import` construct of the core language does not directly support renaming of imported bindings or selective import of specific bindings, the following macro does provide these capabilities.

```
(define-syntax import*
  (syntax-rules ()
    [(_ M) (begin)]
    [(_ M (new old))
     (module* (new)
       (define-alias new tmp)
       (module* (tmp)
         (import M)
         (define-alias tmp old)))]
    [(_ M id) (module* (id) (import M))]
    [(_ M spec0 spec1 ...)
     (begin (import* M spec0)
            (import* M spec1 ...))]))
```

To selectively import an identifier from module `M`, the macro expands into an anonymous module that first imports all exports of `M` then re-exports only the selected identifier. To rename on import, the macro expands into an anonymous module that instead exports an alias bound to the new name. The alias is simply an identifier macro that expands into a reference to the old name visible where the alias is defined.

If the macro placed the definition of `new` in the same scope as the import of `M`, and `new` is also present in the interface of `M`, a naming conflict would arise. To prevent this, the macro instead places the import within a nested anonymous module and links `old` and `new` by means of an alias for the introduced identifier `tmp`.

The macro expands recursively to handle multiple import specifications. Thus the following construct imports `x`, `y` (renamed as `z`), and `z` (renamed as `y`) from module `m`.

```
(import* m x (y z) (z y))
```

The `define-alias` macro is given below. Since aliases are resolved at translation time, programs using `define-alias` and `import*` incur no run-time penalty.

```
(define-syntax define-alias
  (syntax-rules ()
    [(_ x y)
     (define-syntax x
       (identifier-syntax y))]))
```

3.4 Multiple views on modules

Because a module can re-export imported bindings, it is quite easy to provide multiple views on a single module, as

D and E provide for C below, or to combine several modules into a compound module, as C does.

```
(module A (x y)
  (define x 1) (define y 2))
(module B (y z)
  (define y 3) (define z 4))
(module C (a b c d)
  (import* A (a x) (b y))
  (import* B (c y) (d z)))
(module D (a c) (import C))
(module E (b d) (import C))
```

3.5 Mutually recursive modules

Mutually recursive modules can be defined in several ways (see also Section 3.8). In the following program, A and B are mutually recursive modules exported by an anonymous module whose local scope is used to statically link the two. For example, the free variable y within module A refers to the binding for y provided by the import of B in the enclosing module.

```
(module* (A B)
  (module A (x) (define x (lambda () y)))
  (module B (y) (define y (lambda () x)))
  (import A)
  (import B))
```

The following syntactic abstraction generalizes this pattern to permit the definition of multiple mutually recursive modules.

```
(define-syntax rec-modules
  (syntax-rules ()
    ((_ (module N (id ...) form ...) ...)
     (module* (N ...)
       (module N (id ...) form ...) ...
       (import N) ...))))
```

Recursive modules residing in different source files can be combined using the include macro. For example, if a program consists of two modules, m1 and m2, residing in files m1.ss and m2.ss, they can be combined as follows:

```
(module* (m1 m2)
  (include "m1.ss")
  (include "m2.ss")
  (import m1)
  (import m2))
```

As shown below, the rec-modules macro can be adapted to support modules located in different source files.

```
(define-syntax rec-modules
  (syntax-rules ()
    [(_ include (m file) ...)
     (module* (m ...)
       (include file) ...
       (import m) ...)]))
```

Using this version of rec-modules, the example given earlier can be written as follows.

```
(rec-modules include
  (m1 "m1.ss")
  (m2 "m2.ss"))
```

3.6 Separating interface from implementation

To allow interfaces to be separated from implementations, the following macros support the definition and use of named interfaces.

```
(define-syntax define-interface
  (syntax-rules ()
    [(_ name (export ...))
     (define-syntax name
       (lambda (x)
         (syntax-case x ()
           [(_ n defs)
            (with-implicit (n export ...)
              #'(module n (export ...) .
                  defs))])))]))

(define-syntax define-module
  (syntax-rules ()
    [(_ name interface defn ...)
     (interface name (defn ...))]))
```

define-interface creates an interface macro that, given a module name and a list of definitions, expands into a module definition with a concrete interface.[1] These macros can be used as follows.

```
(define-interface simple (a b))
(define-module M simple
  (define-syntax a (identifier-syntax 1))
  (define b (lambda () c))
  (define c 2))
(let () (import M) (list a (b)))  ⇒  (1 2)
```

3.7 Compound interfaces

It is sometimes convenient to combine several interfaces into a compound interface as shown below.

```
(define-interface one (a b))
(define-interface two (c d))
(define-interface both
  (compound-interface one two))
```

The separate interface abstraction defined in Section 3.6 can be extended to support compound interfaces by introducing a limited reflection mechanism. When expanded, the following define-interface macro defines a macro representing the interface as before.

```
(define-syntax define-interface
  (syntax-rules (compound-interface)
    [(_ name (compound-interface i0 i1 ...))
     (d-i-help name (i0 i1 ...) ())]
    [(_ name (export ...))
     (define-syntax name
       (lambda (x)
         (syntax-case x (expand-exports)
           [(_ n defs)
            (with-implicit (n export ...)
              #'(module n (export ...) . defs))]
           [(_ (expand-exports i-name mac))
            (with-implicit (i-name export ...)
              #'(mac i-name export ...))])))]))
```

For example, the definition (define-interface one (a b)) expands into the following macro definition.

```
(define-syntax one
  (lambda (x)
    (syntax-case x (expand-exports)
      [(_ n defs)
       (with-implicit (n a b)
         #'(module n (a b) . defs))]
      [(_ (expand-exports i-name mac))
```

[1] with-implicit, used here to ensure that the introduced export identifiers are visible in the same scope as the name of the module in the define-module form, is implemented in terms of datum->syntax-object [4, 6]. The reader syntax #'⟨form⟩ expands into (syntax ⟨form⟩).

```
        (with-implicit (i-name a b)
          #'(mac i-name a b))])))
```

The second clause of this macro provides a simple mechanism for reflecting on the interface. When invoked with the `expand-exports` auxiliary keyword, the name of the interface `i-name`, and the name of a macro `mac`, this interface macro now expands into the macro call `(mac i-name a b)`. By analogy with continuation-passing style (CPS), the macro `mac` supplied as an argument to the interface macro is a continuation that takes as input the set of exports in the interface.

To construct a compound interface, `define-interface` calls upon the `d-i-help` macro, defined below, to collect the exports of the constituent interfaces.[2]

```
(define-syntax d-i-help
  (syntax-rules ()
    [(_ name () (export ...))
     (define-interface name (export ...))]
    [(_ name (i0 i1 ...) (e ...))
     (begin
       (define-syntax tmp
         (syntax-rules ()
           [(_ name expt (... ...))
            (d-i-help name (i1 ...)
              (e ... expt (... ...)))]))
       (i0 (expand-exports name tmp)))]))
```

The `d-i-help` macro takes the interface name, a list of constituent interfaces, and a list of exports collected so far. When it has finished processing all constituent interfaces, `d-i-help` simply calls `define-interface` with the list of exports collected. In the recursive case, `d-i-help` uses the reflective facility of the first constituent interface, `i0`. The continuation macro, `tmp`, extends the list of exports collected so far and invokes `d-i-help` recursively to process the remaining interfaces.

It is not always convenient to define a compound interface explicitly. The following version of `define-module` allows a compound interface to be specified directly.

```
(define-syntax define-module
  (syntax-rules (compound-interface)
    [(_ name (compound-interface i ...) defn ...)
     (begin
       (define-interface tmp
         (compound-interface i ...))
       (define-module name tmp defn ...))]
    [(_ name interface defn ...)
     (interface name (defn ...))]))

(define-module M (compound-interface one two)
  (define a 1)
  (define b 2)
  (define c 3)
  (define d 4))
(let () (import M) (list a b c d))  ⇒  (1 2 3 4)
```

3.8 Satisfying interfaces incrementally

The abstract module facility defined below allows a module interface to be satisfied incrementally. This permits flexibility in the separation between the interface and implementation, and it supports separate compilation of mutually recursive modules.

```
(define-syntax abstract-module
  (syntax-rules ()
    ((_ name (ex ...) (mac ...) defn ...)
     (module name (ex ... mac ...)
       (declare ex) ...
       defn ...))))

(define-syntax implement
  (syntax-rules ()
    ((_ name form ...)
     (module* () (import name) form ...))))
```

The following example illustrates the use of abstract modules to define mutually recursive modules. We first define two abstract modules, `E` and `O`.

```
(abstract-module E (even?) ())

(abstract-module O (odd?) (pred)
  (define-syntax pred
    (syntax-rules () ((_ exp) (- exp 1)))))
```

We then define implementations of these modules, each of which imports from the other.

```
(implement E
  (import O)
  (satisfy even?
    (lambda (x)
      (or (zero? x) (odd? (pred x))))))

(implement O
  (import E)
  (satisfy odd?
    (lambda (x) (not (even? x)))))
```

`declare` and `satisfy` may simply be `define` and `set!`, although a single-assignment semantics may be more appropriate for `satisfy`.

The interfaces of these abstract modules can be separately compiled. Because the modules are mutually recursive, the compiled interfaces of both modules must be loaded in order to compile either implementation. Subject to this restriction, the implementations may be separately compiled. The compiled interface of an abstract module must be loaded before attempting to use its compiled implementation.

4 Implementation

The module system is implemented by extending the `syntax-case` macro system [6]. Section 4.1 describes the existing macro system, and Section 4.2 shows how the macro system is extended to support modules.

4.1 The `syntax-case` system

As described in Section 2.3, a source program is translated into the core language before it is compiled. To preserve lexical scoping during translation, the `syntax-case` macro system consistently renames bound variables (α-conversion). This substitution process is fully automated by the system and is transparent to the user.

During translation, identifier references are resolved via a two-level map consisting of a substitution environment and a store. The substitution environment maps symbols to labels in the store. It is represented as a sequence of sub-environments called *ribs*, each of which corresponds to a single lexical contour. When searching the substitution environment, the ribs corresponding to the innermost contours are consulted first in accordance with lexical scoping.

[2]The pattern `(... ...)` produces a single ellipsis in the output of the macro. Thus the input pattern specification of the `tmp` macro introduced in the expansion of `d-i-help` is `(_ name expt ...)`.

The store maps labels to translation-time bindings that describe the roles of the corresponding identifiers in the input program. For example, if the environment and store map a symbol to a macro transformer, the symbol represents a syntactic keyword. The store is used to detect attempts to reference run-time bindings within code that is evaluated at translation time.

Macro transformers operate on *syntax objects*. In addition to the usual list-structured source expression, a syntax object contains the substitution environment and marks, collectively referred to as the *wrap*, that apply to the entire expression.

Substitutions are introduced by core binding forms. For example, the `lambda` transformer creates a substitution rib mapping each formal parameter to a fresh label, and extends the store to map each label to a fresh variable. This rib is added to the substitution environment of the syntax object representing the `lambda` body before the body is processed recursively by the macro expander.

Marks are introduced when the expander invokes a macro transformer. A fresh mark is applied to the input of the transformer and the same mark is applied to the output of the transformer. Because identical marks cancel when they meet, marks adhere only to expressions introduced in the expansion of a macro. Substitutions are keyed with marks to prevent substitutions for identifiers introduced in the expansion of a macro from affecting identifiers not introduced by the macro. An identifier is renamed only when the set of marks on that identifier is identical to the set of marks associated with the symbol in the substitution environment.

Within programs, macro transformers are simply `lambda` expressions whose chief distinction is that they may be evaluated and applied at translation time. Syntax objects commonly appear in the expanded code of these transformers. For example, a syntax object representing the identifier `lambda` appears within the expanded code for the `let` transformer. This syntax object contains the substitution environment and marks in effect where `let` was defined. Therefore `let` expands into a reference to the binding for `lambda` visible in the scope where `let` was defined.

Using syntax objects as the input and output of macro transformers enables the system to perform α-conversion lazily. In addition, this representation allows the system to correlate program source and object code through arbitrary user-defined transformations. Macro transformers destructure their input using a pattern-matching facility that exposes list structure within a syntax object and propagates the wrap to the exposed subforms. Syntax objects are destructured only as far as is necessary for pattern matching.

The lazy substitution model is important for two reasons. First, applying substitutions on demand is far more efficient than repeatedly traversing subexpressions to apply substitutions eagerly. In fact, lazy substitution maintains lexical scoping with constant overhead on macro expansion. Second, by delaying substitution until it can determine the role of an input expression, the system avoids traversing (potentially cyclic) structured constants, and it avoids renaming identifiers used as symbolic data. For example, in the body of the following `lambda` expression, the two occurrences of `x` have different roles: the first refers to the formal parameter, and must be renamed, while the second is used as symbolic data, and must retain its original name.

```
(lambda (x)
  (define-syntax f
    (syntax-rules () ((_ e) (quote e))))
  (cons x (f x)))
```

As this example illustrates, the role of an input expression cannot be determined until macro expansion of surrounding forms is complete.

4.2 Adding modules

The module system uses the α-conversion machinery described in Section 4.1 to control the visibility of identifiers. To support modules, the rib structures contained in substitution environments are modified so that they may be extended incrementally, and an interface structure (described below) is added to the set of translation-time bindings to which the store maps.

When processing a `lambda` expression or `module` form, the expander creates an extensible rib in the substitution environment. This rib is part of the wrap pushed down on the form (and propagated to subforms) by the pattern matcher. Definitions within the `lambda` body or `module` form extend this rib with substitutions renaming the identifiers they bind. Internal definitions extend the store with translation-time bindings such as local macro transformers and module interfaces.

To determine the set of internal definitions, the system partially expands each subform in the body of the `lambda` or `module` form until it finds the first non-definition. When it encounters a macro call, the system expands the macro and processes any definitions in the resulting output. For `define`, `define-syntax`, and `module` forms, the system extends the substitution rib of the local scope with a fresh label for the defined variable, keyword, or module name. For a `define` form, the system defers expansion of the expression on the right-hand side and extends the store with a mapping from the fresh label to a translation-time structure representing the lexical variable. For `define-syntax` forms, the transformer expression is expanded and evaluated to obtain a procedure that is installed in the store as the binding for the fresh label. For `module` forms, the store is extended with a mapping from the fresh label to an interface structure listing the exports of the module. The sequencing construct, `begin`, is treated as a splicing form: its subforms are simply added in its place to the list of subforms to be processed.

The system processes `module` forms recursively to collect the set of internal definitions. Definitions inside and outside the module are identical except that the substitution rib extended by definitions inside the module is present only in the wrap pushed down on syntax objects representing expressions within the module. Because internal definitions extend a local substitution rib introduced by the module, substitutions for the defined identifiers are not visible outside the module. Bindings in the enclosing scope are visible within the module because the substitution environments inside and outside the module share a common tail.

Definitions within the module extend the store of the enclosing `lambda` or top-level `module` form. These bindings are accessible only where the substitutions introduced by definitions within the module are in effect. The `import` form makes available the substitutions introduced for identifiers exported from the module. An exported macro can introduce valid references to other bindings in the module where it is defined even when those bindings are not visible or are shadowed where the macro is used. This is possible because syntax objects inserted by the macro close over the substitution environment in effect where they are constructed. In particular, the module rib extended by internal definitions is present in the wrap of each syntax object constructed while processing the module.

When the expander encounters an `import` form, it looks up the module name in the substitution environment and then looks up the resulting label in the store to obtain the module interface. The module's interface structure contains syntax objects naming the exports. These syntax objects contain the module's substitution rib within their wraps. To import a module, the system extends the substitution rib of the enclosing lexical scope with the substitutions extracted from the exported identifiers.

After processing the internal definitions of a `lambda` or `module` form, the system expands the expressions on the right-hand sides of variable definitions and expands any non-definition forms following the definitions. For `lambda` it then wraps the body in a `letrec` form binding variables defined locally. For top-level `module` forms it produces a series of top-level definitions for the renamed identifiers. To aid static analysis we improve on this expansion for top-level `module` forms in Section 5.2.

5 Practical Considerations

This section discusses extensions to the language and translation mechanism to support separate compilation, static program analysis, and isolated scopes.

5.1 Separate compilation

The implementation described in Section 4 supports both internal and top-level modules. For internal modules, the new names generated by the expander must be locally unique, *i.e.*, not otherwise visible within the same top-level expression. For top-level modules within a single compilation unit, the names must be unique within the compilation unit. When multiple compilation units may be linked together, the names must be unique across compilation units.

Our system supports both incremental compilation and linking of compilation units compiled by different runs of the compiler. Since the compiler may be invoked simultaneously on the same or different physical hardware, global name generation factors in the host's network interface identifier, process identifier (PID), and compiler invocation time. Some operating systems provide system calls that generate unique names in much the same fashion.

If a module `m2` imports from a module `m1`, the compiler must have the interface for `m1` in order to compile `m2`. When a source file containing `m1` is compiled, the resulting object file contains the compiled code for `m1`, its interface, and compiled code for any macro transformers defined by `m1`. Before compiling `m2`, it is necessary to first `load` or `visit` `m1`'s object file to install its interface and transformers. While both `load` and `visit` install the interface and transformers, `load` also loads the compiled code and evaluates the initialization expressions.

5.2 Static program analysis

One benefit of modular program structure is that it may improve the results of static analysis. In the presence of separate compilation, global variables are subject to arbitrary manipulation by compilation units that may be unavailable for scrutiny by the compiler. Variables not exported from a module, however, are effectively local and may therefore present opportunities for analysis and optimization.

The ability to export macros complicates the situation, since exported macros may expand into references or assignments to nonexported variables. It is tempting to believe that a straightforward analysis of the transformer expression can determine the set of these *implicit* exports. Even with the constrained `syntax-rules` transformers, however, such an analysis is necessarily conservative. For example, consider the following definition for module `M`.

```
(module M (a b)
  (define-syntax a
    (syntax-rules ()
      [(_ e0 e1 ...) (e0 (quote c) e1 ...)]))
  (define b (lambda () c))
  (define c 3))
```

It appears that `c` is not referenced outside the definition of `b`. The only other occurrence of `c` is within the macro `a`, where it is apparently used as symbolic data. As the following program demonstrates, however, code outside the module may not only reference but also assign `c`.

```
(let ()
  (define-syntax f
    (syntax-rules ()
      [(_ (any id)) id]
      [(_ (any id) value) (set! id value)]))
  (import M)
  (let ([original (a f)])
    (a f 4)
    (list original (b))))  ⇒  (3 4)
```

Here `(a f)` expands to `(f (quote c))`, which further expands to the reference `c`. Similarly, `(a f 4)` expands to `(f (quote c) 4)`, which further expands to the assignment `(set! c 4)`. Because the compiler does not know how the macro `a` will be used in separately compiled code, it must conservatively assume that the variable `c` is assigned and that its value escapes.

Even conservative analysis is impossible if transformers are able to synthesize new identifiers from existing identifiers, *e.g.*, using `datum->syntax-object` [4, 6]. For example, consider the following module definition.

```
(module foo (access)
  (define a 123)
  (define-syntax b (identifier-syntax 456))
  (define-syntax access
    (lambda (x)
      (syntax-case x ()
        [(_ exp)
         (datum->syntax-object #'b
           (syntax-object->datum #'exp))]))))
```

The `access` macro returns its argument expression intact but for one key difference: the wrap on the input is replaced with the wrap from the identifier `b` bound inside the module.[3] Consequently, expanding `(access exp)` has the effect of expanding `exp` within the module scope. For example, `(access b)` constructs a reference to the private syntax binding `b` and `(access (set! a 3))` constructs an assignment to the private variable binding `a`.

In fact, the `access` macro need not be defined within the module that is violated. A module boundary can be breached if any syntax object constructed within the module may arrive at a call to the procedure `datum->syntax-object`. For instance, the `expose` macro below is virtually identical to the `access` macro defined earlier except that it transplants the wrap from an identifier in its input onto `exp`.

[3] The choice of `b` here is arbitrary. Each syntax object in the module carries in its wrap sufficient information to access any module binding.

```
(begin
  (cte-install! m8 'interface '#(interface #<import-token> (#<syntax-object a0> #<syntax-object c6>)))
  (cte-install! a0 'transformer (identifier-syntax b1))
  (cte-install! b1 'transformer (identifier-syntax c2))
  (letrec ([e3 (lambda (x4) (* x4 2))])
    (set! c2 (lambda (x5) (e3 x5)))
    (set! c6 (lambda (x7) (e3 (+ x7 1))))))
```

Figure 1: Sample expansion of a module that exports a macro with implicit exports.

```
(define-syntax expose
  (lambda (x)
    (syntax-case x ()
      ((_ (any n ...) exp)
       (datum->syntax-object #'any
         (syntax-object->datum #'exp))))))
```

Now consider the module defined below and observe that the macro `a` contains no occurrences of the identifier `b`. In fact, `b` is not even referenced within the module.

```
(module foo (a)
  (import scheme)
  (define-syntax a
    (syntax-rules ()
      [(_ e0 e1 ...)
       (e0 (quote 3) e1 ...)]))
  (define b 1))
```

The `expose` macro provides arbitrary access to bindings within the `foo` module, including `b`, as illustrated below.

```
(let ()
  (import foo)
  (a expose
    (let ((old b))
      (set! b 5)
      (list old b))))  ⇒  (1 5)
```

One partial solution is to disallow assignments outside the module to unexported variables. This limits the expressive power of exported macros and does not address the problem of escaping values. Another partial solution is to introduce a `define-constant` form that establishes immutable bindings. This does not inherently limit the expressive power of exported macros but leaves the problem of escaping values.

We have chosen instead to require that implicit exports be explicitly declared in the interface for top-level modules[4] using the following syntax.

$\langle\text{interface}\rangle \longrightarrow$ `(`$\langle\text{export}\rangle$`*)`
$\langle\text{export}\rangle \longrightarrow$ $\langle\text{identifier}\rangle$ | `(`$\langle\text{identifier}\rangle$ $\langle\text{export}\rangle$`*)`

Declarations of implicit exports are propagated to enclosing interfaces where an identifier is exported. For example, the module `m` defined below exports only `a` and `c`, but the export of `a` causes the implicit export of `b` and `c` from module `n`.

```
(module m ((a b) c)
  (define-syntax a (identifier-syntax b))
  (module n ((b c) (d e))
    (define-syntax b (identifier-syntax c))
    (define c (lambda (x) (d x)))
    (define-syntax d (identifier-syntax e))
    (define e (lambda (x) (* x 2))))
  (import n)
  (define c (lambda (x) (d (+ x 1)))))
```

[4] The declarations are permitted but not required for internal modules.

Because `e` is not exported, explicitly or implicitly, it is bound locally in the output of the expander. Thus the preceding program is equivalent to the code in Figure 1. Because `e` is not assigned, it can be integrated by a later pass of the compiler.

5.3 Isolated scopes

Expressions within a module can reference identifiers bound outside of the module. For example, the following expression evaluates to `8`.

```
(let ((x 3))
  (module m (plusx)
    (define plusx (lambda (y) (+ x y))))
  (import m)
  (let ((x 4)) (plusx 5)))
```

The system also supports a variant of `import`, called `import-only`, that creates an *isolated scope*, in which the only visible identifiers are those exported by the imported module. This is useful for static verification that a module does not access identifiers that are not explicitly provided by the import. For example, the following expression generates a translation-time error.

```
(let ((x 3))
  (module m ())
  (import-only m)
  x)
```

By using `import-only` to implement the qualified reference macro of Section 3, we ensure that it introduces a reference to the intended export and not to some local binding.

```
(define-syntax from
  (syntax-rules ()
    ((_ M id) (let () (import-only M) id))))
```

When a module is imported via `import-only`, subsequent imports are not possible unless this module exports the name of at least one module and `import` or `import-only`. To create an isolated scope containing the exports of more than one module without making `import` or `import-only` visible, it is necessary to create a single module that contains the exports of each of the other modules.

```
(module m2 (y) (define y 'y))
(module m1 (x) (define x 'x))
(module compound-module (x y cons)
  (import m1)
  (import m2)
  (import scheme))
(let ((x 3))
  (import-only compound-module)
  (cons x y))  ⇒  (x . y)
```

Like the `module` and `import` forms, the `import-only` form is a primitive language construct treated specially by the macro expander. The macro expander makes the exports of a module imported via `import-only` visible in the same manner as for `import`. In addition, the expander adds to the local substitution rib a `fail` token, keyed with the set of marks that apply to the `import-only` identifier. To determine the role of a source-program identifier, the expander consults the substitution environment and the store as before. An error is signalled if the expander encounters a matching `fail` token while searching the substitution environment for the symbol and marks representing that identifier. To match, the set of marks associated with the `fail` token must match the set of marks on the identifier.

6 Related Work

Our system is similar in some respects to systems described by Curtis and Rauen [2] and Rees [16]. Our system is based on simpler core module constructs, however, and derives its expressive power largely via syntactic abstraction. While the other systems build in support for separation of interface from implementation, for example, our system allows multiple interface separation mechanisms to be created via syntactic abstraction. Curtis and Rauen forbid the definition and import of modules in local scopes, a feature we have found quite useful. It is not clear whether Rees intends to permit local module definitions. Although the implementation (in Scheme 48) does not forbid the such definitions, it provides no apparent way to use them. In both systems, the use of syntactic abstractions expressed in terms of modules is limited. Neither system implements separate compilation.

Rees points out that prohibiting assignment to imported bindings permits the compiler to assume that any variable not assigned within a module is never assigned. The current Scheme 48 implementation, however, does not enforce this restriction. In particular, a Scheme 48 transliteration of the first example in Section 5.2 shows that code outside a module can assign an apparently unassigned variable, even if that variable is not exported. To avoid this, the implementation would have to prevent exported macros from expanding into assignments to either exported or nonexported variables, limiting expressiveness. Curtis and Rauen describe a similar restriction but give few details and no implementation.

Curtis and Rauen propose a *meta-module* facility that permits the modularization of the code used in transformer expressions. In particular, they allow macro transformers to share code. Meta-modules are similar to ordinary modules except that they export only variable bindings and they are evaluated at translation time. This is a useful feature that our system currently lacks, although nothing in our system precludes support for such a feature.

Queinnec and Padget describe a module language for controlling the visibility of sets of named locations [13, 14]. They describe a high-level macro expansion protocol intended to support various macro system implementations such as expansion-passing style [5] or hygienic systems based on syntactic closures [1]. Because their system is not tightly integrated with the macro expander they do not permit nested modules, nor do they permit macros to expand into module constructs.

The Dylan programming language [17] supports both modules and lexically scoped macros. Their macro system is more restrictive in its treatment of symbols and identifiers. Macro and module definitions can appear only at the top level of a compilation unit (library), and import is tied to the module syntax, limiting expressiveness. To determine the set of identifiers implicitly exported by macros, the body of each macro transformer is analyzed. Because it is generally impossible to compute the precise set of implicitly exported bindings, an upper bound is computed by assuming exported any binding with the same name as an identifier appearing in the right-hand side of a rewrite rule.

We considered implementing a similar analysis in our system but ultimately dismissed this approach for two reasons. First, by foiling the conservative analysis, seemingly innocent edits can have unanticipated performance consequences. For example, inserting a quoted symbol in the wrong place can cause the like-named identifier to be implicitly exported. Second, our mechanism for intentional capture, `datum->syntax-object`, is sufficiently powerful that all identifiers are potential implicit exports, as demonstrated in Section 5.2. While Dylan provides a limited form of intentional capture via the macro template modifier `?=`, this is insufficient to express macros such as `include` [4].

Flatt and Felleisen propose a system for dynamically linking separately compiled program *units* [7, 8]. While it does not permit the export of macros, the `unit` facility is interesting as representative of systems providing both higher-order modules and programmatic control over linking. They argue against module linkage via static import and instead propose a mechanism that provides programmatic control over module linkage. We believe that the interfaces should be specified statically whenever possible as an aid to static program analysis (both by the programmer and by the compiler). However, the static nature of our system does not preclude support for dynamic module linkage. As demonstrated in Figure 2, the `unit` module system can be implemented entirely at the source level in our language.

7 Conclusion

We have presented a language design that augments a small core language with simple module and import forms and a powerful syntactic abstraction facility that permits the construction of new language features, including richer module constructs. The module language is based entirely on manipulation of static scope. A program containing modules can therefore be understood in terms of a straightforward translation into the core language without module forms.

We have demonstrated how syntactic abstractions can be defined to support anonymous modules, qualified module access, mutual recursion, multiple views on a single module, arbitrary combination of modules, separate interfaces, compound interfaces, and incrementally satisfiable interfaces.

Module definitions and imports may appear in any context where variable or keyword definitions may appear, allowing modules to be nested or to be used locally. Modules are therefore useful not only for large-scale software development but also at a *micro-modular* level within expressions.

We have fully implemented the features described in this paper and incorporated the implementation into the front end of the *Chez Scheme* compiler. The system supports both incremental and separate compilation. A portable implementation of the module system is freely available via ftp[5].

Acknowledgements

Comments from Mike Ashley, Bob Burger, Dan Friedman, Brian Moore, and anonymous reviewers improved the presentation of the paper.

[5] `ftp.cs.indiana.edu/pub/syntax-case`

```
(module Unit ((unit make-unit) (compound-unit make-unit) invoke-unit)
  (define-syntax make-unit
    (syntax-rules ()
      [(_ ((ev eloc) ...) (iv ...) form ...)
       (values (list (cons 'ev eloc) ...) (lambda (iv ...) form ... (void)))]))
  (define-syntax unit
    (lambda (x)
      (syntax-case x (import export)
        ((_ (import iv ...) (export ev ...) form ...)
         (with-syntax (((eloc ...) (generate-temporaries (syntax (ev ...)))))
           #'(lambda ()
               (let ((eloc (box 'undefined)) ...)
                 (make-unit ((ev eloc) ...) (iv ...)
                            (module* () (set-box! eloc ev) ...)
                            (let-syntax ((iv (identifier-syntax
                                               (_ (unbox iv))
                                               ((set! _ val) (set-box! iv val))))
                                         ...)
                             form ...)))))))))
  (define-syntax compound-unit
    (lambda (x)
      (syntax-case x (import link export)
        ((_ (import iv ...) (link (ltag (expr linkage ...)) ...) (export (etag ev) ...))
         (with-syntax (((ulocs ...) (generate-temporaries (syntax (ltag ...))))
                       ((uimp ...) (generate-temporaries (syntax (ltag ...)))))
           #'(lambda ()
               (mvlet* ([(ulocs uimp) (expr)] ...)
                 (define-syntax ltag
                   (syntax-rules ()
                     ((_ id) (cdr (assq 'id ulocs))))) ...
                 (make-unit ((ev (etag ev)) ...) (iv ...) (uimp linkage ...) ...))))))))
  (define-syntax invoke-unit
    (syntax-rules ()
      [(_ expr) (mvlet* ([(ulocs uimp) (expr)]) (uimp))])))
```

Figure 2: An implementation of the UNIT_d module facility of Flatt and Felleisen. A `unit` is a procedure that returns an interface and an implementation procedure that is parameterized over its imports. A `unit` interface associates variable names with first-class reference cells. Within a `unit` implementation, references and assignments to imported variables are rewritten (via `identifier-syntax`) as explicit operations on the reference cells passed in as arguments. A group of units is linked by obtaining the interfaces and implementations of each and invoking each implementation procedure with an appropriate set of reference cells from the various interfaces as determined from the linkage specification. A fully linked unit is invoked by invoking its implementation procedure.

References

[1] William Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Seventeenth Annaual ACM Symposium on Principles of Programming Languages*, pages 155–162, January 1991.

[2] Pavel Curtis and James Rauen. A module system for Scheme. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 13–19, June 1990.

[3] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 113–120, 1994.

[4] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, second edition, 1996.

[5] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[6] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[7] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, September 1998.

[8] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[9] Richard Kelsey, William Clinger, and Jonathan A. Rees (Editors). Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[10] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[11] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, Indiana, 1986.

[12] David MacQueen. Modules for standard ML. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207, 1984.

[13] Christian Queinnec and Julian Padget. A deterministic model for modules and macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[14] Christian Queinnec and Julian Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[15] Jonathan Rees. Modular macros. Master's thesis, Massachusetts Institute of Technology, May 1989.

[16] Jonathan Rees. *Another Module System for Scheme*. Massachusetts Institute of Technology, 1994. Scheme 48 documentation.

[17] Andrew Shalit. *The Dylan Reference Manual*. Addison Wesley Longman, 1996.

[18] Sho-Huan Simon Tung and R. Kent Dybvig. Reliable interactive programming with modules. *Lisp and Symbolic Computation*, 9(4):343–358, 1996.