Syntactic Abstraction in Component Interfaces

Ryan Culpepper[1], Scott Owens[2], and Matthew Flatt[2]

[1] Northeastern University (ryanc@ccs.neu.edu)
[2] University of Utah ([sowens, mflatt]@cs.utah.edu)

Abstract. In this paper, we show how to combine a component system and a macro system. A component system separates the definition of a program fragment from the statements that link it, enabling independent compilation of the fragment. A macro system, in contrast, relies on explicit links among fragments that import macros, since macro expansion must happen at compile time. Our combination places macro definitions inside component signatures, thereby permitting macro expansion at compile time, while still allowing independent compilation and linking for the run-time part of components.

1 Introduction

Good programmers factor large software projects into smaller components or modules. Each module addresses a specific concern, and a program consists of a network of cooperating modules. First-order module systems provide name management, encapsulation, and control over separate compilation [1]. However, first-order module systems use *internal linkage*, in which modules refer directly to other modules.

A module system can support component programming [2] by separating module definition from linking. Thus, components use *external linkage*, in which a component refers indirectly to other components through a parameterization mechanism. Additionally, a component must be compilable and deployable by itself, apart from any linkages that use the component. In analogy to separate compilation, we call this property *independent compilation*. A single independently compiled component is therefore re-usable in various situations, linked with a variety of other components. Although many module systems [3–5] support component-style parameterization, we concentrate here on a system designed expressly for component programming: *Units* [6].

Units and other component systems allow a component to import and export values and types, but not macros. Macro support is desirable because macros allow the definition of domain-specific language extensions, and components may benefit from these extensions. Because Scheme [7] supports sophisticated, lexically-scoped macros [8], implementors have devised module systems that support the import and export of macros [5, 9–12], but these module systems do not support component-style parameterization with independent compilation.

This paper explains how to integrate macros and components while maintaining the desirable properties of both. In particular, our macros respect the lexical scope of the program and our components can be compiled before linking.

Section 2 introduces an example of a component-based program in PLT Scheme, our implementation substrate. Section 3 explains how the use of macros improves the program and introduces the macro system. Section 4 shows how we combine macros and components, and Section 5 shows more uses of macros in components. Section 6 discusses related work, and Section 7 concludes.

2 Programming with Units

Units are software components with explicit import and export interfaces. These interfaces serve as the canonical mechanism for designing systems and communicating documentation. Furthermore, since units are externally linked, they can be independently compiled.

Using the PLT Scheme unit system, programmers can organize programs as networks of components. Units are heavily used in the major applications distributed with PLT Scheme, including DrScheme and the PLT web server.

2.1 The unit system

Signatures are the interfaces that connect units. A signature specifies a set of bindings that a unit may either import or export. A unit specifies one signature that lists its exported bindings, but it can specify many signatures listing imported bindings to support importing from multiple different units.

Signatures are defined using the **define-signature** form:

(**define-signature** signature-id
  (variable-id*))

The **unit/sig** expression specifies an atomic unit as follows:

(**unit/sig** (**import** signature-id*) (**export** signature-id)
  definition-or-expression$^+$)

The export signature indicates which definitions in the unit body are exported, and the import signatures indicate what variables are bound in the unit body. Unit expressions need not be closed. Like procedures, unit values close over the unit expression's free variables.

Units are *externally linked*; that is, a unit expression cannot refer specifically to the contents of another unit. Thus, compilation of a unit body does not require knowledge of any other unit, so units are independently compilable. Unit compilation depends only on signatures to determine import and export variables. These variables are compiled to use an indirection that supports linking.

Programs use a separate linking form called **compound-unit/sig** to link units together, satisfying each unit's import signatures:

(**compound-unit/sig**
  (**import** (tag **:** signature)*)
  (**link** (tag **:** signature (unit-expr tag*))$^+$)
  (**export** (**var** tag **:** identifier)*))

The tags correspond to the nodes of the linkage graph, and the lists of tags in the link clauses specify the edges of the graph. The result is another unit whose imports are specified by the **import** clause and whose export signature is computed from the variables listed in the **export** clause.

The **invoke-unit/sig** form invokes a unit with no imports:

(**invoke-unit/sig** unit-expr)

An invocation evaluates all definitions and expressions in the unit's (atomic or compound) body in order.

2.2 An example

Throughout the rest of this paper, we use the example of a hotel registration system that uses a database for persistent storage. The business logic consists of the following code:[3]

```
(define-signature db-sig (query make-select-cmd))
(define-signature hotel-reg-sig (get-reservation get-empty-rooms))

(define hotel-reg-unit
  (unit/sig (import db-sig) (export hotel-reg-sig)
    ;; get-reservation : string date → reservation
    (define (get-reservation name date)
      (———— (query (make-select-cmd 'reservations
                                    (list 'room 'rate 'duration)
                                    (list (cons 'name name) (cons 'date date)))
                   ————)))
    ;; get-empty-rooms : date → (list-of (cons string number))
    (define (get-empty-rooms date) ————)))
```

The third definition binds *hotel-reg-unit* to a unit satisfying the *hotel-reg-sig* interface. It must be linked with a database unit that exports the *db-sig* interface before it can be used.

The *hotel-reg-unit* component can be linked with any database component that exports the *db-sig* interface. In turn, the *hotel-reg-unit* unit provides the functionality represented by the *hotel-reg-sig* signature. This functionality may be used by any number of different components in the system, such as a graphical user interface for finding reservations and vacancies.

Assembling these components—the database code, the business logic, and the user-interface code—creates a complete program:

```
(define hotel-program-unit
  (compound-unit/sig
    (import)
    (link [HOTEL-DB : db-sig (PrestigeInc-db-unit)]
          [HOTEL-REG : hotel-reg-sig (hotel-reg-unit HOTEL-DB)]
          [GUI : hotel-gui-sig (hotel-gui-unit HOTEL-REG)])
    (export)))

(invoke-unit/sig hotel-program-unit)
```

[3] In the code fragments, we use ———— to indicate elided code.

The hotel's programmers can also write a web interface and assemble a second view for the registration software:

```
(define hotel-servlet-unit
  (compound-unit/sig
    (import (SERVER : servlet-import-sig)
    (link [HOTEL-DB : db-sig (GargantuWare-db-unit)]
          [HOTEL-REG : hotel-reg-sig (hotel-reg-unit HOTEL-DB)]
          [WEB-UI : servlet-sig (hotel-webui-unit SERVER HOTEL-DB)])
    (export))))
```

The web server would then link *hotel-servlet-unit* against a unit providing controlled access to the web server's functionality and invoke the resulting unit.

Signatures not only control the linking process; they also guide programmers in writing the components that implement and use the interfaces they represent. The brief signature definition of *db-sig*, however, is insufficient as a guide to implementing or using the the database unit. The true interface consists of not only the set of imported names, but also the types of those names, descriptions of their meanings, and advice on their use. Figure 1 shows an improved description of the *db-sig* interface.

```
(define-signature db-sig
  [;; query : select-cmd ((list-of string) → α) → (list-of α)
   ;; Applies the procedure to the each record returned, accumulating the final result.
   query

    ;; a constraint is (pair-of symbol {string|number|boolean})

    ;; make-select-cmd : (list-of symbol) symbol (list-of constraint) → select-cmd
    make-select-cmd

    ;; Example: to return a list of the concatenations of fields
    ;; 'f1' and 'f2' in table 'tab1' where 'f3' equals 0:
    ;;   (query (make-select-cmd (list 'f1 'f2) 'tab1 (list (cons 'f3 0)))
    ;;          (lambda (fieldset)
    ;;            (string-append (list-ref fieldset 0) ":" (list-ref fieldset 1))))
   ])
```

Lines with ";;" are comments.

Fig. 1. Database interface

The comments in this revised signature specify function headers and contain a usage example for *query*, a complex function. The example shows how to use *query* to select two fields, *f1* and *f2*, from the table named *tab1*. Only the rows where field *f3* is zero are processed. The argument function extracts the fields

from *fieldset* by position and concatenates them. The call to *query* returns the accumulated list of concatenated strings.

The example exposes the awkwardness of the unadorned *query* function. The results come back in a list of 3-tuples (also represented with lists) that must be unpacked by position. This fragile relationship breaks when fields are reordered, inserted, or deleted. The programmer should be able to refer to field values by name rather than by position, and the database library should provide an abstraction that manages the connection between field names and their offsets in the result tuples. Ideally, the field names would be variables in the result-handling code. Creating a variable binding based on arguments is beyond the power of procedural abstraction, but it is a standard use of macros.

3 Programming with Macros

A declarative **select** form, specifically designed for expressing database queries, is more robust and less awkward to use than the *query* function. It can express the example query from Fig. 1 as follows:

```
;; Example: to return a list of the concatenations of fields
;; 'f1' and 'f2' in table 'tab1' where 'f3' equals 0:
(select [(f1 f2) tab1 with (f3 = 0)]
  (string-append f1 ":" f2))
```

The **select** form is implemented with a macro that compiles a **select** expression into a call to the *query* function. In Scheme, a **define-syntax** expression binds a compile-time function as a macro.

```
(define-syntax (macro-name stx)
  macro-body)
```

A macro takes in and produces annotated s-expressions called *syntax objects*. In a macro body, the **syntax-case** form matches a syntax object against patterns and binds *pattern variables*, and the #' operator creates a syntax object from a *template*. Each pattern variable used in a template refers to the portion of the input expression matched by that variable. The postfix ellipsis operator in a pattern matches the previous pattern zero or more times (similar to Kleene star); in a template it repeats the previous template for every match.

Figure 2 presents the **select** macro in the syntax-case [8] macro system. For the above **select** example, this macro generates code equivalent to the *query* example in Fig. 1.

The **select** macro uses *field*, *table*, *f1*, *v2*, and *body* as pattern variables, but **with** and = are matched as literals. The macro uses the name *field* both as a symbol (by putting it inside a **quote** form) passed to *make-select-cmd* and as a variable name. Because *body* is placed within that **lambda** expression, *field* is bound in *body*, and the evaluation of *body* is delayed until the procedure is applied.

The macro expander invokes the **select** macro when it encounters an expression of the form (**select** ————). The macro's parameter, here *stx*, is bound

```
(define-syntax (select stx)
  (syntax-case stx (with =)
    [(select [(field ...) table with (f1 = v2) ...] body)
     ;; table, all field, and all f1 are identifiers
     ;; each v2 can be any expression
     ;; body is an expression
     #'(query (make-select-cmd (list (quote field) ...)
                               (quote table)
                               (list (cons (quote f1) v2) ...))
              (lambda (fieldset)
                (apply (lambda (field ...) body)
                       fieldset)))]))
```

Fig. 2. A simple **select** macro

to the **select** expression. Macro expansion replaces the **select** expression with the result of the **select** macro and continues processing the program.

3.1 Respecting lexical scope

Unlike macros in LISP or C, Scheme's macros respect lexical scoping.[4] Variable references introduced in a macro's template are bound in the environment of the macro's definition (i.e. to the lexically apparent binding occurrence), instead of the environment of the macro's use. Hence, the macro's user can use local variables that coincide with those chosen by the macro implementer without altering the macro's behavior. In addition, templated identifiers that become binding occurrences never capture references received through the macro's argument. This property protects the meaning of the macro user's code from the macro definition's choice of temporary variables.

For example, consider the definition of the **select** macro in Fig. 2. The template contains a use of the *query* variable. Since the macro is defined at the top-level, the expanded code always refers to the *top-level* variable of that name, even if **select** is used in a context where *query* is shadowed by a local binding.

Scheme's macro system stores information about the lexical context of identifiers in the syntax objects that macros manipulate. The following superscript annotations are intended to illustrate the information that the syntax objects carry:

```
(define query^top ————)
(define-syntax (select^top stx)
  (syntax-case stx (with =)
    [(select [(field ...) table with (f1 = v2) ...] body)
     #'(query^top (make-select-cmd^top (list^top (quote^top field) ...) ————)
                  ————)]))
```

[4] Historically, a lexically-scoped macro system has also been called *hygienic* [13] and *referentially transparent* [14].

Thus, the following code

```
(let ([query^loc "What is the meaning of life?"])
  (select^top [(f1 f2) tab1 with (f3 = 0)]
    (string-append^top f1 ":" f2)))
```

expands into the following:

```
(let ([query^loc "What is the meaning of life?"])
  (query^top (make-select-cmd^top (list^top (quote^top f1) (quote^top f2)) ————)
             ————))
```

The macro system uses the lexical context information to ensure that the use of $query^{top}$ is not bound by $query^{loc}$, but refers to the top-level binding.

The **select** macro also relies on the other guarantee of lexically-scoped macro expansion. The macro expander uses another form of annotation on syntax objects to track identifiers introduced by macros. When those identifiers become binding occurrences, such as *fieldset* in the template of **select**, they bind only uses also generated by the same macro expansion. Consider this use of the **select** macro:

```
(let ([fieldset (lambda (new-value) (set! last-field-seen new-value))])
  (select [(f1 f2) tab1 with (f3 = 0)]
    (fieldset (+ f1 f2))))
```

This expands into the following:

```
(let ([fieldset (lambda (new-value) (set! last-field-seen new-value))])
  (query ————
         (lambda (fieldset_1)
           (apply (lambda (f1 f2) (fieldset (+ f1 f2)))
                  fieldset_1))))
```

Again, the binding of $fieldset_1$ does not capture the use of *fieldset*.

These two guarantees made by lexically-scoped macro systems are crucial to writing and using reliable macros. Without static knowledge of the binding structure of macros, reasoning about them becomes impossible.

However, consider the implications for programming with units. A macro such as **select** which uses the *query* function must be defined in a context where *query* is bound. Since *query* is a unit variable, it is only available to other units, through an import clause. Thus the macro definition must occur within the body of the importing unit:

```
(define hotel-reg-unit
  (unit/sig (import db-sig) (export hotel-reg-sig)
    (define-syntax select ————)
    ————))
```

Putting the macro definitions in the client code, however, defeats the component abstraction. The database component should provide everything necessary to write client code—both dynamic and static constructs—so that every application component can use these facilities.

One attempt at a solution might define **select** in the database component and export it to the client components. With this solution, the client component

could not be compiled until it is linked with a particular database component, for two reasons. First, compilation of a unit depends on the definition of every syntactic extension used in its body. Second, the contents of a unit are not available to other units until link-time. Thus if the definition of a macro resides in a unit, then its clients cannot be compiled until link-time. Furthermore, the same client unit might be compiled differently for every linkage it participates in. Thus, this proposal violates the important property of component programming that a component be independently compilable.

4 Signatures and static information

Our solution integrates macros and components by including macro definitions in signatures. Since signatures specify the static properties of units, they are a natural place to put syntactic extensions. In this section we present the design, pragmatics, and implementation of the new unit system.

4.1 Extended signatures

A signature contains the set of names that make up a unit's interface. In the extended system, it also contains macro definitions that can refer to the signature's names.

The extended syntax of signature definitions is:

```
(define-signature signature-id
  (variable-id*
   macro-definition*))
```

With this extension, it is possible to put the **select** macro in the *db-sig* signature, rather than manually copying it into every client unit:

```
(define-signature db-sig
  [query
   make-select-cmd
   (define-syntax (select stx) ————)])
```

The syntax for units remains the same. When the *db-sig* signature is used in an import clause, however, it also inserts the **select** macro into the unit:

```
(unit/sig (import db-sig) (export application-sig)
  (select [(room rate) rooms with (available = true)]
    (format "~a, available for just $~a" room rate)))
```

Macro expansion of the unit body produces the desired target code:

```
(unit/sig (import db-sig) (export application-sig)
  (query (make-select-cmd (list 'room 'rate)
                          'rooms
                          (list (cons 'available true)))
         (lambda (fieldset)
           (apply (lambda (room rate)
                    (format "~a, available for just $~a" room rate))
                  fieldset))))
```

The expansion of the macro does not depend on the particular version of the database unit linked at run-time. In fact, the above unit may be linked to many database units during the course of the same program execution.

4.2 Preserving lexical scope

As discussed in Sect. 3.1, respect for lexical scope is a critical property of Scheme's macro systems. It ensures that variable references behave in the natural way. In particular, variable references inserted by a macro refer to the variables in the macro definition's context, not those in the context of the macro's use.

The problem becomes more complex when we allow macros to occur inside of signatures. When a signature macro refers to a signature variable, there is no definite binding of the variable for the macro to refer to. Rather, when the signature is *instantiated*, that is, when it is used in a unit's import clause, the instance of the macro will refer to the instance of the imported variable.

This observation suggests a natural extension of the principle of lexical scoping for macros. In a signature, a free occurrence of a name in a #'-form should have the same meaning as it does in the context of the signature definition, unless the name is also a signature element. In the latter case, the name should refer to the variable linkage created when the signature is used in a unit **import** clause.

To illustrate this principle, consider the example from earlier:

```
(define-signature db-sig
  [query
   make-select-cmd
   (define-syntax (select stx)
     (syntax-case stx (with =)
       [(select [(field ...) table with (f1 = v2) ...] body)
        #'(query (make-select-cmd (list (quote field) ...) ————)
                 (lambda (fieldset) (apply ————)))]))])
```

In the template for **select**, *query* and *make-select-cmd* must denote the unit import variables. In contrast, *list* and *apply* must refer to the standard Scheme procedures, because those were their meanings where the macro was defined. Any other interpretation of macro definitions would violate the scoping principle of Scheme. It is the task of the implementation to make sure that these properties hold.

4.3 Implementation

In order to compile units, the compiler must be able to completely expand the definitions and expressions in a unit's body and identify all imported and exported variables. The information necessary to do this is statically bound to the signature names, and the compilation of a **unit/sig** form consists of fetching this static information, eliminating the signatures from the code, and compiling the resulting core unit form.

Signature definition When the compiler encounters a **define-signature** form, it builds a catalog consisting of the variable names and macro definitions that the signature contains. The catalog contains syntax objects, which are closed in the syntactic environment in which the macro signature occurs. This ensures that when the macro definitions are imported into a unit, lexical scoping is preserved.

Finally, it statically binds the signature name to a signature structure containing the information above. Thus,

```
(define-signature db-sig
  [query
   make-select-cmd
   (define-syntax (select stx) ————)])
```

binds *db-sig* to the static information

```
(make-signature 'db-sig
                ;; Variables
                (list #'query #'make-select-cmd)
                ;; Macro names
                (list #'select)
                ;; Macro definitions
                (list #'(define-syntax (select stx) ————)))
```

When *db-sig* appears in the import clause of a **unit/sig** form, the compiler looks up the static binding of *db-sig* and uses the catalog to eliminate the use of the signature.

Signature elimination The compiler translates **unit/sig** forms into an intermediate core unit form, replacing signatures with their contents. In the resulting code, imported and exported variables are explicitly enumerated, and all signature-carried macro definitions are inlined into the core unit body. This elimination process respects scoping, as outlined in Sect. 4.2.

Consider the following example:

```
(define-signature db-sig
  [query make-select-cmd (define-syntax select ————)])

(let ([query^loc ————])
  (unit/sig (import db-sig) (export application-sig) ————))
```

The **select** macro definition that occurs in the signature refers to a variable whose name appears in the **let** binding. The macro definition must be closed in the environment of the signature definition so that those names are not captured by the **let** binding. By using syntax objects in the signature catalog, we retain the correct syntactic environment for the macro definition.

There are two problems to overcome in implementing **unit/sig** correctly. First, the environment of the macro definitions is not quite complete; it lacks bindings for the signature's variables. Second, the names in the signature, being closed in a different environment, will not bind the names in unit body.

To illustrate, we write $query^{sig}$ for the identifier closed in the signature's environment and $query^{unit}$ for the identifier occurring in the unit body. Given the **unit/sig** expression

(**unit/sig** (**import** *db-sig*) (**export** *application-sig*)
  ($query^{unit}$ ————)
  (**select**$^{\mathbf{unit}}$ [(*room rate*) *rooms* **with** (*available* = *true*)] (———— *rooms rate*)))

if the variable names and macro definitions from *db-sig* were copied into the unit body, their identifiers would still have *sig* superscripts.

(**unit/sig** (**import** ($query^{sig}$ $make\text{-}select\text{-}cmd^{sig}$)) (**export** *application-sig*)
  (**define-syntax** (**select**$^{\mathbf{sig}}$ *stx*) ———— #'$query^{sig}$ ————)
  ($query^{unit}$ ————)
  (**select**$^{\mathbf{unit}}$ [(*room rate*) *rooms* **with** (*available* = *true*)] (———— *rooms rate*)))

Note that $query^{sig}$ is bound in the resulting unit body, so the environment for the definition of **select**$^{\mathbf{sig}}$ is complete. Unfortunately, $query^{unit}$ and **select**$^{\mathbf{unit}}$ are still unbound in the unit body.

To solve this problem, the compiler must identify the unit's local identifier for each signature identifier. For example, given $query^{sig}$, the compiler must determine that the local name is $query^{unit}$. Each such pair of foreign and local identifiers must have the same meaning. The compiler extends the environment of the unit form with bindings that alias each local identifier to the corresponding imported identifier.

(*make-signed-unit*
  ;; Metadata (intentionally omitted)
  ————
  ;; Core unit
  (**unit** (**import** $query^{sig}$ $make\text{-}select\text{-}cmd^{sig}$)
        (**export** ————)
    ;; Imported macro definitions
    (**define-syntax** (**select**$^{\mathbf{sig}}$ *stx*) ———— #'$query^{sig}$ ————)
    ;; Alias code
    (**define-alias** (**select**$^{\mathbf{unit}}$ = **select**$^{\mathbf{sig}}$)
                ($query^{unit}$ = $query^{sig}$)
                ($make\text{-}select\text{-}cmd^{unit}$ = $make\text{-}select\text{-}cmd^{sig}$))
    ;; Code from unit/sig begins here
    ($query^{unit}$ ————)
    (**select**$^{\mathbf{unit}}$ [(*room rate*) *rooms* **with** (*available* = *true*)] (———— *room rate*))))

Fig. 3. Translation into core unit

In summary, the compiler copies each macro definition from its imported signatures into the unit body, explicitly enumerates the imports and exports (using the foreign names), and creates aliases for the local names. This pass

eliminates the signatures and produces an intermediate core unit form in which all imported and exported variables are explicit, and the body consists of macro definitions, definitions, and expressions.

Figure 3 shows the result of this translation, using **unit** as the syntax for core unit forms and a **define-alias** instruction to create aliases. The **define-alias** form can be implemented with the low-level primitives of the PLT Scheme macro system.

Core units Once the signatures have been eliminated, all import and export variables are explicitly enumerated and the unit body contains all imported macros definitions.

Compiling a core unit involves expanding the body—a simple process once the macro definitions have been brought into the proper context—and compiling imported and exported variables as cells managed by the unit linker rather than primitive Scheme variables.

5 Static Programming

Many constructs can be expressed with a macro whose output depends only on the macro's input. The **select** macro (Fig. 2) is an example of such a construct. However, the compilation of some constructs relies on information from other locations in the program. For example, the pattern matching construct for algebraic datatypes relies on the definition of the datatype being matched. PLT Scheme's macro system supports the implementation of these kinds of constructs by letting macros communicate using compile-time variables.

A database has a statically-known structure called its *schema* that determines what queries are valid. We could add a mechanism to *db-sig* that lets clients specify the schema. Then **select** would be able to check at compile time that every query will succeed when run.

The **database-table** form lets a programmer express information about the database's relevant tables, fields, and types, and make the information accessible to the **select** macro.

;; database-table's syntax: (database-table table-name (field-name field-type) ...)
(**database-table** *rooms* (*room number*) (*rate number*) (*available boolean*))
(**database-table** *reservations* (*room number*) (*name text*) (**when** *date*))

Using these table declarations, the **select** macro can ensure that the table in question is declared, and that it contains the given fields. It can also emit run-time checks for the field types.

The **database-table** and **select** macros communicate through a compile-time variable named **schema**, which contains a mutable box. Compile-time variables are introduced using the **define-for-syntax** form, and they exist during macro expansion. See Fig. 4 for the implementation of **schema** and **database-table**. The **begin-for-syntax** form in the expansion of the **database-table** form indicates that the enclosed expression should execute during macro expansion. The **database-table** form is similar to examples from previous work [12].

```
(define-for-syntax schema (box null))

(define-syntax (database-table stx)
  (syntax-case stx ()
    [(database-table table (field-name field-type) ...)
     #'(begin-for-syntax
         (let ([table-record '(table (field-name field-type) ...)])
           (set-box! schema (cons table-record (unbox schema)))))]))
```

Fig. 4. Implementation of **database-table**

```
(define-syntax (select stx)
  ;; get-table-info : identifier → table-info
  ;; Given a table name, returns the list of fields and types associated with it,
  ;; or raises an exception if the table is not known.
  (define (get-table-info table-stx)
    (let ([table-entry (assoc (syntax-object→datum table-stx) (unbox schema))])
      (if table-entry
          (cdr table-entry)
          (raise-syntax-error 'select "unknown table" table-stx))))

  ;; check-field : identifier table-info → void
  ;; Checks that the given field name belongs to the table.
  (define (check-field field-stx table-info)
    (let ([field-record (assoc (syntax-object→datum field-stx) table-info)])
      (unless field-record
        (raise-syntax-error 'select "field not declared" field-stx))))

  (syntax-case stx (with =)
    [(select [(field ...) table with (f1 = v2) ...] body)
     (let ([table-info (get-table-info #'table)])
       (for-each (lambda (field-id) (check-field field-id table-info))
                 (syntax→list #'(field ...)))
       ;; The resulting code is the same as before (see Fig. 2)
       #'————)]))
```

Fig. 5. Implementation of **select** with static checking

The new **select** macro checks the table name and fields it receives against the information in the stored schema. The revised implementation shown in Fig. 5 produces the same code as the previous **select** macro, but it additionally uses helper procedures to check that the table and fields are declared. If they are not, the macro signals a compile time error.[5]

Consider the following unit that uses the new *db-sig* facilities:

```
(unit/sig (import db-sig) (export application-sig)
  (database-table rooms (room number) (rate number) (available boolean))
  (database-table reservations (room number) (name text) (when date))
  (select [(room rate) rooms with (available = true)] (cons room rate))
  (select [(room duration) reservations with] (——— duration ———)))
```

The first use of **select** is correct, but the second use triggers a compile-time error, notifying the programmer that *duration* is not a valid field in the *reservations* table.

The *db-sig* signature contains the definitions of the **schema** variable and **database-table** macro alongside the **select** macro. Thus, when a unit's import specifies the *db-sig* signature, **schema**'s and **database-table**'s definitions are copied into the unit's body. Consequently, each unit with a *db-sig* import receives its own **schema** box, keeping table definitions from mixing between different units.

The **select** and **database-table** macros present other opportunities for static extensions that we do not explore here. For example, the **database-table** form could produce code to dynamically verify the accuracy of the static schema description, and the **select** macro could add additional code to check whether constant field specifications have the correct type.

6 Related Work

Bawden [22] has proposed a system of lexically-scoped, "first-class" macros based on a type system. The "first-class" macros are statically resolvable, which preserves compilability, but the values for the bindings used in a macro's expansion can be passed into and returned from functions. A "first-class" macro is defined in a template that includes macro definitions and a listing of variables that the macro is parameterized over, similar to our signatures. His system uses types to statically track macro uses, whereas in our system signatures are statically attached to units, avoiding the need for a type system.

Krishnamurthi's *unit/lang* construct [21] allows programmers to specify the programming language of a component in addition to its external interface. A language contains new macro-like extensions (in fact, languages are more powerful than macros) and run-time primitives. A unit/lang component internally specifies which language it imports, similar to how our units specify their signatures. However, the internally specified language position does not coordinate

[5] A programming environment such as DrScheme [15] can use the information provided to *raise-syntax-error* to highlight the source of the problem.

with the externally linked component parameters, so lexically-scoped macros cannot refer to these parameters. Our system also makes it simpler to mix together orthogonal language extensions, since there is no need to manually define a language that contains the desired combination of extensions.

The Scheme community has formalized the principles of scope-respecting macros. Kohlbecker et al. [13] first introduced the hygiene property for macros, and subsequent papers developed referential transparency [8, 14]. Recent papers have addressed macros and internally-linked modules [11], including a notion of phase separation for macros and modules [12]. Other work in macro semantics from MacroML [16, 17] has not addressed the full complexity of Scheme macros. In particular, these systems do not model macros that can expand into macro definitions.

A different line of research has developed module systems to support component programming. Components are parameterized using collections of code that obey static interfaces, but the information carried by these interfaces is generally limited. In ML module systems [4, 18, 19], signatures contain types (and kinds, i.e., types for types) and datatype shapes (used for pattern matching). Similarly, the signatures in the original model of units [6] contain types and kinds. In the Jiazzi unit system [20], signatures contain class shapes, which play a type-like role.

The Scheme48 [5] module system provides some support for modules with parameterized imports. However, the signatures for the imports only contain the information that a binding is a macro, and the not macro itself. Consequently, parameterized modules that import macros cannot be independently compiled. We believe that our techniques could correct this problem.

7 Conclusion

We have designed and implemented an extension of the PLT Scheme unit system that allows programmers to attach language extensions to signatures, thus enriching the interfaces available to client code. Our extension preserves the essential properties of the unit system, such as independent compilation and external linking, as well as the lexical scoping principles of the macro system.

References

1. Wirth, N.: Programming in MODULA-2 (3rd corrected ed.). Springer-Verlag New York, Inc., New York, NY, USA (1985)
2. Szyperski, C.: Component Software. Addison-Wesley (1998)
3. MacQueen, D.: Modules for standard ml. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. (1984) 198–207
4. Leroy, X.: Manifest types, modules, and separate compilation. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1994) 109–122
5. Kelsey, R., Rees, J., Sperber, M.: Scheme48 Reference Manual. 1.1 edn. (2005) http://s48.org/1.1/manual/s48manual.html.

6. Flatt, M., Felleisen, M.: Units: Cool modules for HOT languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (1998) 236–248
7. Kelsey, R., Clinger, W., Rees (Editors), J.: Revised^5 report of the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76
8. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and Symbolic Computation **5** (1993) 295–326
9. Queinnec, C.: Modules in scheme. In: Proceedings of the Third Workshop on Scheme and Functional Programming. (2002) 89–95
10. Serrano, M.: Bigloo: A "practical Scheme compiler". 2.7a edn. (2005) http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html.
11. Waddell, O., Dybvig, R.K.: Extending the scope of syntactic abstraction. In: Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, New York, NY (1999) 203–213
12. Flatt, M.: Composable and compilable macros: You want it *when?* In: ACM SIGPLAN International Conference on Functional Programming. (2002)
13. Kohlbecker, E.E., Friedman, D.P., Felleisen, M., Duba, B.F.: Hygienic macro expansion. In: ACM Symposium on Lisp and Functional Programming. (1986) 151–161
14. Clinger, W., Rees, J.: Macros that work. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1991) 155–162
15. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A programming environment for Scheme. Journal of Functional Programming **12** (2002) 159–182 A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.
16. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In: International Conference on Functional Programming. (2001) 74–85
17. Taha, W., Johann, P.: Staged notational definitions. In: Proceedings of the second international conference on Generative programming and component engineering. (2003) 97–116
18. Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1994) 123–137
19. Harper, R., Pierce, B.C.: Design issues in advanced module systems. In Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2004) To appear.
20. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned Java. In: Proc. conference on Object oriented programming, systems, languages, and applications, ACM Press (2001) 211–222
21. Krishnamurthi, S.: Linguistic Reuse. PhD thesis, Rice University (2001)
22. Bawden, A.: First-class macros have types. In: Proc. symposium on Principles of programming languages, ACM Press (2000) 133–141