Advanced Macrology and the Implementation of Typed Scheme

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

Sam Tobin-Hochstadt
Northeastern University
samth@ccs.neu.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Abstract

PLT Scheme provides an expressive programming language implementation framework in order to enable experimentation with language design. This framework is rooted in PLT Scheme's hygienic macro system, but it has grown to encompass features that extend its capabilities beyond that of traditional macro systems.

In this paper we describe the features of PLT Scheme's language framework and demonstrate their use with a case study. Specifically, we present the design and implementation of Typed Scheme using the advanced language construction features of PLT Scheme.

1. Defining Languages

Since their creation, Lisp and Scheme macros have been used by programmers to extend their programming languages with notational abbreviations and domain-specific syntactic forms. Macros thus make it easier to read and write programs by bringing the programming language closer to the problem domain.

Discussions of macros often leave out the challenges that arise when macros need to work with other macros [6]. These challenges also appear in the construction of tools such as debuggers and static analyzers for a language defined via macros. The language tools should operate at the language's level of abstraction, not at the level of the underlying Scheme code. In general, collaborating macros and proper language abstraction require features from the macro system beyond those needed for isolated abstractions.

Collaborating macros must share information. For example, many syntactic forms in PLT Scheme deal with named structure types. The macro that defines new structure types publishes relevant information for other forms to consume. The pattern matching form [23] uses that information to check the arity of structure patterns and compile the pattern matching code.

Ideally, a similar sort of communication should take place between macros and language tools. When a new language can implement its language constructs in terms of corresponding constructs in the host language—for example, lexical scoping in the new language can be translated to lexical scoping in the host language—then the tools that analyze those constructs are reusable as well. For example, DrScheme's Check Syntax and debugging tools [3] work with the Algol60 [14], Lazy Scheme [1], Typed Scheme [20, 21], and other translation-based languages [8] because of this kind of linguistic reuse.

Even in the ideal case, though, language tools require hints from the macro expansion process to interpret expanded programs and correlate the expanded code with the original source. Virtually every part of DrScheme [10], from its debugging support to its graphical analysis displays, benefits from the automatic source location information stored in syntax objects, adapted from the recommendations of the **syntax-case** report [7]. Source tracking is mostly automatic. The Stepper [2], on the other hand, must reconstruct or "unexpand" terms in the supported languages. For example, it must determine whether a chain of **if** expressions came from a **cond** expression, a **case** expression, an **or** expression, or something else. This reconstruction requires communication between the supported languages and the Stepper.

This paper presents the features of the language infrastructure that support collaborating macros, language definitions, and analysis tools. It illustrates the design and the pragmatics of these features with the implementation of Typed Scheme, a new member of the PLT Scheme language suite.

The paper is organized as follows. Section 2 introduces Typed Scheme, an extension of the PLT Scheme language with types. Section 3 presents the syntax framework that provides the foundation for macro programming and language implementation in PLT Scheme. Section 4 sketches the Typed Scheme implementation approach at the level of single definitions and expressions. Section 5 shows how the implementation scales up to multi-module programs. Finally, Section 6 discusses related work.

2. Typed Scheme

Typed Scheme is an explicitly-typed dialect of PLT Scheme designed to support the addition of types to existing programs one module at a time [21]. Typed modules interact with untyped modules safely and intuitively. The type system accommodates the idioms of Scheme programming, minimizing the need to change the structure of the program when adding types.[1] It supports the features of PLT Scheme, most importantly macros, modules, and structures.

2.1 The language

The syntax of Typed Scheme differs from untyped Scheme only in its binding and definition forms. These require type annotations for all bound variables, structure fields, etc. A Typed Scheme program is executed by erasing the types after type-checking and running the residual Scheme program.

The following Typed Scheme program introduces many of the relevant language constructs:

(**define:** (*double* [*nums* : (*Listof Number*)]) : (*Listof Number*)
  (*map* (**lambda:** ([*n* : *Number*]) (∗ 2 *n*)) *nums*))

[1] The particular features of the type system are beyond the scope of this paper.

The **define:** and **lambda:** forms parallel the traditional versions but include type information. The rest of the code is identical to the equivalent PLT Scheme code.

Since PLT Scheme programs use structures extensively, Typed Scheme provides a typed structure definition form:

```
(define-typed-struct rectangle
      ([width : Number] [height : Number]))
(define-typed-struct circle ([radius : Number]))
```

The programmer can then define new types based on those types:

```
(define-type-alias shape (∪ rectangle circle))
```

Here, the *shape* type is the union of *circle*s and *rectangle*s. The following program illustrates how to manipulate values of this type:

```
(define: (area [sh : shape]) : Number
  (cond [(circle? sh)
         (* 3.1416 (circle-radius sh) (circle-radius sh))]
        [else (* (rectangle-width sh) (rectangle-height sh))]))
```

Programmers can also define polymorphic functions such as *foldr*:

```
(pdefine: (a b) (foldr [combine : (a b → b)]
                       [base : b]
                       [items : (Listof a)]) : b
  (cond [(pair? items)
         (combine (car items) (foldr combine base (cdr items)))]
        [else base]))
```

An important goal of Typed Scheme is to support programs that use macros. Typed Scheme programs can define and use macros, such as the following:

```
(define-syntax shape-case
  (syntax-rules (circle rectangle)
    [(shape-case sh
       [(circle rad) circle-expr]
       [(rectangle w h) rectangle-expr])
     (cond [(circle? sh)
            (let ([rad (circle-radius sh)])
              circle-expr)]
           [(rectangle? sh)
            (let ([w (rectangle-width sh)]
                  [h (rectangle-height sh)])
              rectangle-expr)])]))
```

Using **shape-case** simplifies our implementation of *area*:

```
(define: (area [sh : shape]) : Number
  (shape-case sh
    [(circle r) (* 3.1416 r r)]
    [(rectangle x y) (* x y)]))
```

The type-checker can type-check the function definition even though it uses a macro.

Typed Scheme programs can also use some—but not all—of PLT Scheme's standard macro libraries, even though those libraries know nothing about Typed Scheme. For example, we can use the **match** macro to simplify the *area* function:[2]

```
(define: (area [sh : shape]) : Number
  (match sh
    [(struct circle (rad)) (* 3.1416 rad rad)]
    [(struct rectangle (w h)) (* w h)]))
```

[2] To make supporting macros such as **match** easier, Typed Scheme includes a trivial form of type inference.

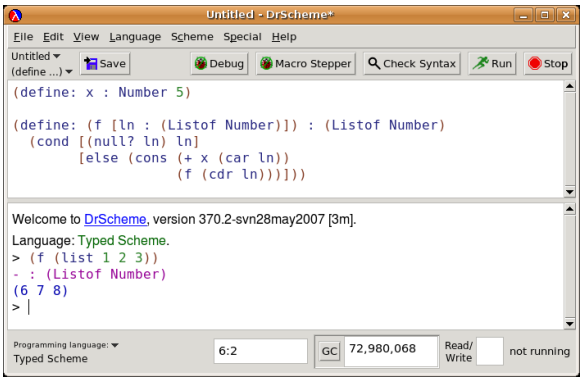

Figure 1. Typed Scheme in DrScheme

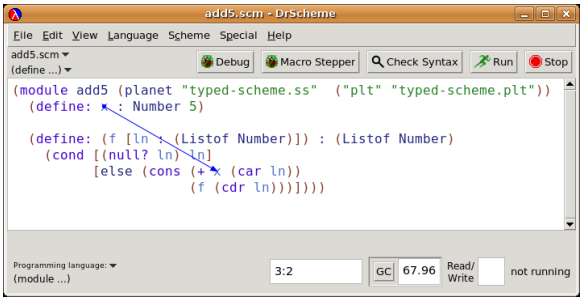

Figure 2. Typed Module Programs

More complicated macro libraries require special cooperation with Typed Scheme.

2.2 Using Typed Scheme

A programmer can use Typed Scheme in two different modes. First, the programmer can select the Typed Scheme language level in DrScheme, depicted in Figure 1. This provides the standard interactive top-level semantics, but with the type-checking previously described. When expressions are evaluated, their types print out as a part of the read-eval-print loop.

Second, the programmer can write a *typed module*, i.e., a PLT Scheme module specifying Typed Scheme as the module's language. Figure 2 shows such a typed module. The body of the module is Typed Scheme code, and the compiler type-checks the module during compilation; if type-checking fails, compilation fails and the module cannot be loaded.

A program may contain a mixture of typed and untyped modules (and even modules written in other macro-based languages). This mixture supports the process of gradual migration to Typed Scheme. One typed module can import values (and macros) from another using PLT Scheme's standard **require** form, as follows:

```
(require "some-typed-module.ss")
```

A typed module can also be used by untyped code, again using the standard **require** form. The typed module's exports are automatically protected by dynamically-checked contracts that maintain the typed module's invariants [21]. Finally, a typed module can require an untyped module by annotating the require specification with types:

```
(require/typed (lib "file.ss")
    [find-files ((Path → Boolean) Path → (Listof Path))])
```

The programmer-supplied types are automatically converted to contracts and checked at the module boundary, again preserving the typed module's invariants. These features mean that typed modules can be seamlessly integrated into larger programs.

3. Syntax Framework

The evolution of PLT Scheme has been guided by two major goals:

- implementing DrScheme and its teaching languages
- supporting research on programming tools and languages

These goals have pushed PLT Scheme to develop a powerful framework for representing and manipulating programs.

This section presents a summary of the standard and nonstandard features of PLT Scheme's language framework.[3] The purpose of this section is to show how these features are a part of a comprehensive language implementation framework, even those that appear to be of marginal interest when considered in isolation.

3.1 Macros

PLT Scheme's macro system is based on the hygienic [4, 17] **syntax-case** system [7], named after the syntactic form it provides for destructuring the syntax of macro occurrences. A distinguishing aspect of this system is its use of a syntax object system, a rich datatype for representing program fragments.

Unlike the **syntax-rules** facility [16], **syntax-case** permits *procedural* macros, i.e., macros that use Scheme code to compute the macro's transformation. Procedural macros have several major advantages over R5RS's restricted pattern-rewriting macros:

- Procedural macros can perform computation at compile time using Scheme's natural computation mechanisms. In contrast, many computations with **syntax-rules** require writing macros in a contorted style, using the macro expander as a trampoline, resulting in many artificial macro transformation steps [15]. Furthermore, many computations are simply impossible with **syntax-rules** because the pattern-rewriting language is insufficiently expressive.
- Procedural macros allow the programmer to detect and report syntax errors. Macro writers can enforce constraints on legal syntax, such as a given list of identifiers must not contain duplicates; detect when those constraints are violated; and report errors in an appropriate, context-specific fashion. In contrast, pattern-rewriting macros have only two kinds of errors: failure to match any pattern and misuse of a primitive form.
- The **syntax-case** macro system allows macros to construct identifiers that capture references not introduced by the macro—often called "breaking hygiene."

The macro definition in Figure 3 demonstrates the major capabilities of **syntax-case** macros. Its purpose is to create procedures that access and update a shared, hidden variable. For example, a programmer can write (**define-getter**+**setter** *balance*) to create definitions for *get-balance* and *set-balance!*.

The macro defines a procedural abstraction (*symbol-append*) to help construct names. Within the **syntax-case** clause, the macro checks that the given name is an identifier (a syntax object containing a symbol); otherwise, it raises an error. Then it uses the macro system's *datum→syntax-object* procedure together with its own *symbol-append* abstraction to construct the names of the getter and setter procedures. This macro breaks hygiene, because the hygiene principle states that introduced names only capture references to the same name that are introduced by the same macro transformation.

[3] While some of the features are PLT specific, many are supported in some form by other Scheme implementations.

```
(define-syntax (define-getter+setter stx)
  ;; symbol-append : symbol ... → symbol
  (define (symbol-append . syms)
    (string→symbol
     (apply string-append (map symbol→string syms))))
  (syntax-case stx ()
    [(define-getter+setter name init-value)
     ;; constraint checking:
     (unless (identifier? #'name)
       (raise-syntax-error 'define-get+set
                           "expected identifier"
                           #'name))
     ;; transformation:
     (with-syntax
         ([getter
           (datum→syntax-object
            #'name
            (symbol-append 'get-
                           (syntax-object→datum #'name)))]
          [setter
           (datum→syntax-object
            #'name
            (symbol-append 'set-
                           (syntax-object→datum #'name)
                           '!))])
       #'(define-values (getter setter)
           (let ([name init-value])
             (values (lambda () name)
                     (lambda (new-value)
                       (set! name new-value)))))])))
```

Figure 3. A **syntax-case** macro

3.2 Modules, or You Want it When, Again?

The PLT Scheme module system [13] allows programmers to group definitions, use imports and exports to control the scope of names, and specify the dependencies between modules. The presence of macros complicates the notion of dependence between modules.

In the presence of procedural macros, a compiler must execute parts of a program in order to deal with the remainder of the program. This blurs the line between compilation and execution. In particular, an interpreter may draw the line in a different place than the compiler, requiring programmers to debug their compiled program after they have already debugged their interpreted program. To eliminate this potential for inconsistency, the PLT Scheme module system provides uniform behavior in both interactive and batch-compilation mode by making module dependencies explicit.

Modules are compilation units, and every module must be compiled before it can be used. Modules contain declarations of their direct dependencies. When a module is compiled, the module system uses those declarations to determine the portions of existing modules that must be executed to support the compilation of the current module. If a macro transformer depends on a value definition, the macro's module must declare a "for-syntax" dependency on the value definition's module. Scoping rules prevent access from macros to undeclared run-time dependencies, and the compiler creates separate instantiations of declared dependencies to prevent interference across separate compilations.

```
(module macro-util mzscheme
  (provide check-for-duplicate-identifier)
  (define (check-for-duplicate-identifier ids) omitted ))

(module rec mzscheme
  (require-for-syntax macro-util)
  (define-syntax (recur stx)
    (syntax-case stx ()
      [(recur name ([var init] ...) . body)
       (begin
         (check-for-duplicate-identifier #'(var ...))
         #'(letrec ([name (lambda (var ...) . body)])
             (name init ...)))]))
  (define (build-list n f)
    (recur loop ([i 0])
      (if (< i n)
          (cons (f i) (loop (+ i 1)))
          null))))
```

Figure 4. Four kinds of references

3.2.1 Split environments

Syntactically, a module declaration contains the module's name, a module reference specifying the language that the module is written in, and a sequence of definitions and expressions:

```
(module module-name initial-language
  module-contents ···)
```

Denotationally, a module consists of two code parts: a compile-time component and a run-time component. The compile-time part consists of the syntax definitions. The run-time part consists of ordinary definitions and expressions. In addition, a module has a set of dependencies on other modules.

The compiler keeps separate environments for the compile-time expressions and run-time expressions. If a module defines a procedure as a run-time value, a macro transformer in the same module cannot *use* that procedure; the binding is unavailable in the compile-time phase. The macro can, of course, expand into code that *refers* to the procedure. Likewise, a binding in the compile-time phase cannot be used in the run-time phase. This phase separation permits the compiler to compile a module without also executing its entire contents.[4]

The two environments yield two kinds of module dependence and thus two distinct module import forms. The **require** form imports bindings into the run-time environment, and the **require-for-syntax** form imports bindings into the compile-time environment.

Macros bridge the gap between the two phases. The implementation of a macro is a compile-time expression, but the macro definition extends the environment for run-time expressions. To understand this idea, it is important to distinguish between the notions of macro versus value bindings from the notions of compile-time versus run-time environments.

The modules in Figure 4 illustrate the four different possibilities. In the context of the *rec* module, *check-for-duplicate-identifier* is a value binding in the compile-time environment; thus, it is available for use in the body of the **recur** macro definition. Even though *check-duplicate-identifier* is a "compile-time procedure," it is not a macro. In fact, it cannot be used in run-time expressions at all. In contrast, **recur** is a macro binding in the run-time environment. It is bound to a compile-time value, but the binding is available to run-time expressions such as the definition of *build-list*. The occurrence of **syntax-case** refers to a macro binding in the compile-time environment. Finally, the definition of *build-list* creates a value binding in the run-time environment.

Compilation of a module involves executing its dependencies[5] and expanding uses of macros in the module's body. The dependencies include the compile-time part of the module's initial language module, the compile-time part of every module imported with **require**, and both compile-time and run-time parts of every module imported with **require-for-syntax**.

The rules for compilation (and also for invoking a module's compile time part) are as follows:

- For every **require** import, including the initial language module, invoke that module's compile-time part in the same phase.
- For every **require-for-syntax** import, invoke that module's compile-time and run-time parts in the next higher phase.

If a module imported twice, once with **require** and once with **require-for-syntax**, the two corresponding invocations of the module are separate. They do not share mutable state. The module system uses phase numbers to distinguish the different instances. Finally, a module is only invoked once per phase, per compilation. Multiple modules that depend on a single module in the same phase share a single invocation of that module and its state.

3.2.2 Compilation independence

True separate compilation is impossible in a module system that supports the import and export of macros. Instead, the module system has a principle of compilation independence:

> The compilation of a module depends only on the compiled forms of the modules that it (transitively) requires.

This principle has two consequences:

- The compilation of two modules, neither of which transitively requires the other, should produce the same two results no matter which is compiled first, or whether they are compiled in parallel.
- The compilation of a module does not depend on side effects that occurred during the compilation of modules that it transitively requires. This has important implications for the use of side-effects at compile time.

The compiler effectively creates a new store for each module that it compiles. Each compilation gets a new execution of all supporting module code. Since the result of the compilation process is nothing but a body of code, the states of mutable variables and objects created during the compilation process of any module are discarded at the end.

The following set of modules illustrates the interaction between side-effects and compilation:

```
(module storage mzscheme
  (define storage '())
  (define (add! x) (set! storage (cons x storage)))
  (provide storage add!))
(module memory mzscheme
  (require-for-syntax storage)
```

[4] The same name may have meanings in both phases simultaneously. For example, modules written in the *mzscheme* language automatically import all primitive bindings into both phases.

[5] If the module depends on modules that are not already compiled, they are automatically compiled when the dependency is detected.

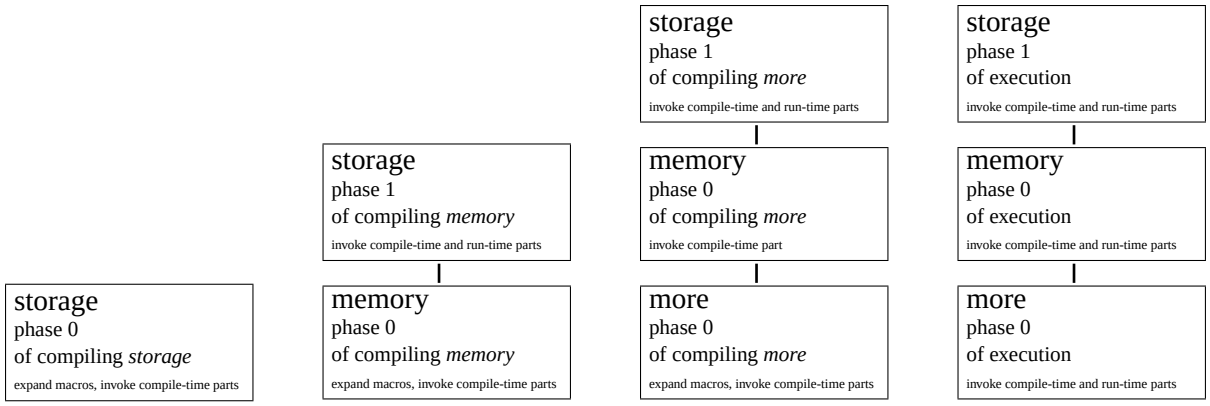

Figure 5. Module invocations

```
(define-syntax (remember stx)
  (syntax-case stx ()
    [(remember sym)
     (begin (add! (syntax-object→datum #'sym))
            (with-syntax ([syms storage])
              #'(begin (display (quote syms))
                       (newline))))]))
(remember a)
(remember b))
```

The first module defines two variables. The second module accesses the variables at compile time, so it imports the first module via **require-for-syntax**. It defines a **remember** macro that adds a symbol to the remembered list and generates code to print out the updated list of remembered symbols. Then it uses the macro twice. At the end of compiling the *memory* module, the *storage* variable has the value `(b a)`. Executing the *memory* module prints out the lists `(a)` and `(b a)`, as expected.

Consider the following addition to the program:

```
(module inspect-storage mzscheme
  (require storage)
  (require memory)
  (display storage) (newline))
```

When this module is executed, the last line it prints out is `()`, not `(b a)`, because the *run-time* instance of the *storage* module is distinct from the *compile-time* instance. That is, side-effects do not cross phases.

Now consider this further addition to the program:

```
(module more mzscheme
  (require memory)
  (remember c))
```

When this module is executed, the last line it prints out is `(c)`, not `(c b a)`. This result often surprises macro programmers. Many of them expect the final line to be `(c b a)`. It seems to them as if the effects in *memory* occur and are subsequently unwound behind their backs. Programming with compile-time side effects can result in unexpected behavior—or lack of behavior—unless programmers recognize the forgetful nature of the compilation process.

The reason that the (**remember** *c*) in *more* prints just `(c)` is that *more* was compiled with a fresh instance of *storage* (initially the empty list), and because executing the compile-time part of *memory* does not change that value. The variable is updated during *macro expansion*; the side-effects are not present in the compiled form of *memory*:

```
(compiled-module memory
  (require mzscheme)
  (require-for-syntax storage)
  (define-syntax (remember stx) omitted )
  (begin (display '(a)) (newline))
  (begin (display '(b a)) (newline)))
```

Figure 5 shows all of the module invocations involved in compiling and executing this program. Each box represents a module invocation, and the text at the bottom of each box indicates what parts of the module are executed. Each column represents a shared store; effects in one column are not visible in another column.

3.2.3 Persistent effects

The compilation rules of the module system yield a design pattern for expressing persistent effects. Compile-time side effects are transient. Only the code in the compiled module is permanent. Thus, the way to express a persistent effect is to make it part of the module. Here's one way to do it:

```
(module memory mzscheme
  (require-for-syntax storage)
  (define-syntax (storage-now stx)
    (syntax-case stx ()
      [(storage-here)
       (with-syntax ([syms storage])
         #'(quote syms))]))
  (define-syntax (remember stx)
    (syntax-case stx ()
      [(remember sym)
       #'(begin (define-syntax _ (add! (quote sym)))
                (display (storage-now))
                (newline))]))
  (remember a)
  (remember b))
```

The effect is not executed within the macro, but the macro expander executes the resulting **define-syntax** form when it continues expanding the module body, so the effect of the first addition to the list still occurs before the second **remember** is expanded. This version introduces a helper macro, **storage-now**, to retrieve the value of *storage* after the update.

Since the compile-time part of a compiled module includes all of the macro definitions, the side-effect is preserved:

```
(compiled-module memory
  (require mzscheme)
  (require-for-syntax storage)
  (define-syntax (storage-now stx) omitted )
  (define-syntax (remember stx) omitted )
  (define-syntax _1 (add! 'a))
  (display '(a)) (newline)
  (define-syntax _2 (add! 'b))
  (display '(b a)) (newline))
```

The calls to *add!* are executed whenever *memory* is required for the compilation of another module. Thus they are executed when *more* is compiled (refer back to Figure 5), so the storage is already set to (*b a*) when the use of **remember** in *more* is expanded. Thus, executing the new version of the program prints (*c b a*), as expected.

As a matter of readability, the **begin-for-syntax** form accomplishes the same effect as the awkward use of **define-syntax** with a throw-away name. Using **begin-for-syntax** also explicitly signals the programmer's intent to generate an expression that creates a persistent effect.

3.3 Macro protocols

Some language extensions involve not just a single macro definition, but a collection of collaborating macros, or one macro whose multiple uses collaborate. Those collaborating macros need ways to share information at expansion time.

For example, any datatype created with **define-struct** can be recognized and destructured using **match**, as follows:

```
(define-struct posn (x y))

(define (dist-to-origin p)
  (match p
    [(struct posn (a b))
     (sqrt (+ (sqr a) (sqr b)))]))
```

The **define-struct** macro gives **match** access to the names of *posn*'s predicate and accessor functions, and **match** uses those names in the expansion of the pattern to test the value, extract its contents, and bind the results to the pattern variables *a* and *b*.

PLT Scheme provides three mechanisms for communication between macros: static bindings, side-effects, and syntax properties. Each mechanism fits a particular form of communication.

3.3.1 Static binding

PLT Scheme generalizes **define-syntax** to bind names to arbitrary compile-time data. The definition of the *posn* structure above produces something similar to the following:

```
(begin
  (define-values (make-posn posn? posn-x posn-y) omitted )
  (define-syntax posn
    (list #'make-posn
          #'posn?
          (list #'posn-x #'posn-y))))
```

Despite the use of **define-syntax**, the definition of **posn** is not a macro, as its value is not a transformer procedure. The static information it carries is accessible from other macros (such as **match**) via the *syntax-local-value* procedure.

With static binding, the availability of information is tied to the name it is bound to. Static binding also relies on the ability to define the name; it cannot attach information to a name that is already bound. Still, static binding is the most common mechanism for defining macro protocols in the PLT Scheme libraries, including protocols for structs and component signatures [5].

3.3.2 Side-effects

Side-effects are commonly used to provide implicit channels of communication between collaborating run-time components. They are just as capable of providing such channels at compile time for macros, provided the programmer recognizes the difference between ephemeral and persistent effects and uses the appropriate technique.

3.3.3 Syntax properties

Dybvig et al.'s [7] syntax datatype extends S-expressions with the hygienic binding information and source location tracking. PLT Scheme adds *syntax properties*, key-value pairs of arbitrary associated data, as a way of attaching information to particular terms. By default, syntax properties are simply preserved by macros and primitive syntactic forms, so protocols defined via syntax properties generally do not interfere if they choose different keys. Accessing information contained in syntax properties requires only access to the term that carries it and the key to the property. Syntax properties are available even to observers that cannot access the expansion environment (necessary to access static bindings and compile-time variables).

For these reasons, syntax properties are well-suited to conveying information from macros to code analyzers that examine programs after they have been expanded to core Scheme. DrScheme's Check Syntax tool examines expanded programs to graphically display the program's binding structure, and it uses syntax properties to supplement binding information. For example, the expansion of **match** uses the information bound to the **posn** name, but the structure name does not occur in the expansion. The **match** macro leaves a 'disappeared-use syntax property on its expansion telling Check Syntax to color the occurrence of **posn** as a reference and connect it to the corresponding definition. DrScheme's teaching languages also use syntax properties to communicate with tools like the Stepper.

Macros can introduce and examine syntax properties in their arguments using the *syntax-property* procedure.

3.4 Local expansion

Some special forms must partially expand their bodies before processing them. For example, the primitive forms such as **lambda** handle internal definitions by partially expanding each form in the body to detect whether it is a definition or an expression. The definitions are collected and transformed into a **letrec** expression with the remainder of the original expressions in the body.

Macros can perform the same kind of partial expansion via the *local-expand* procedure.

3.5 Compilation-unit hooks

There are two basic compilation scenarios in PLT Scheme. In interactive mode, the compiler receives expressions from the read-eval-print loop. In module mode, the compiler processes an entire module at once. For each mode, the compiler provides a hook so the macro system can be used to control compilation of that body of code.

3.5.1 Top-level transformers

The read-eval-print loop automatically wraps each interaction with the **#%top-interaction** macro. By defining a new **#%top-interaction** macro, a programmer can customize the behavior of each interaction.

```
(module typed-scheme mzscheme
  (provide (rename top-interaction #%top-interaction))
  (define-syntax (top-interaction stx)
    (syntax-case stx ()
      [(top-interaction . term)
       (let ([expanded-term
              (local-expand #'term 'top-level null)])
         (type-check-top-level expanded-term)
         expanded-term)]))
  omitted )
```

Figure 6. *typed-scheme* module

3.5.2 Module transformers

The macro expander processes a module from top to bottom, partially expanding to uncover definitions, **require** and **require-for-syntax** forms, and **provide** forms. It executes syntax definitions and module import forms as it encounters them. Then it performs another pass, expanding the remaining run-time expressions. The module system provides a hook, called **#%module-begin**, that allows language implementations to override the normal expansion of modules.

The module transformer hook is typically used to constrain the contents of the module or to automatically import modules into the compile-time environment. For example, the *mzscheme* module transformer inserts (**require-for-syntax** *mzscheme*) at the beginning of the module body, so they automatically get the *mzscheme* bindings in the compile-time phase.

The module hook technique has been used before in language experimentation. Specifically, Pettyjohn et al. [19] prototyped a language for programming continuation-based web servlets. This prototype was the first evidence that the module transformer is useful for general-purpose language experimentation.

4. Typing Terms

The implementation of Typed Scheme illustrates like no other language design experiment the power of PLT Scheme's macro system. In this section, we explain the process for type-checking a single definition or expression, with a focus on type annotations and the use of type environments. The following section extends the implementation to handle modules.

Typed Scheme was designed to interoperate with PLT Scheme's existing macro and module systems. In particular, typed programs should be able to use existing macros (provided they produce type-correct code) and define and use new macros. Since it is generally impossible to derive type rules for arbitrary macros, the type-checker must analyze the program after expansion has eliminated all occurrences of macros and reduced the program to core syntax.

The type-checker hooks into the compilation process as a macro using the **#%top-interaction** interface described in Section 3.5. The type-checking macro receives the original unexpanded program, and it calls *local-expand* to fully expand the program for analysis. The type-checker then either approves the expanded program or raises an error, aborting compilation. Figure 6 shows the beginning of the *typed-scheme* language module.

The type-checker has rules for each primitive syntactic form. It knows how to assign types to Scheme constants. It also knows the types of the Scheme primitive operators. When it encounters a programmer-introduced variable, however, it needs to find the type of the variable, and although that information is present in the original program, type information is not part of fully expanded, core Scheme code. The rest of this section discusses the treatment of variable types and the communication between Typed Scheme's binding forms and its type-checker.

```
(define-syntax (lambda: stx)
  (syntax-case stx (:)
    [(lambda: ([formal : formal-type] ...) . body)
     (with-syntax ([(typed-formal ...)
                    (map
                     (lambda (id type)
                       (syntax-property id 'type-label type))
                     (syntax→list #'(formal ...))
                     (syntax→list #'(formal-type ...)))])
       #'(lambda (typed-formal ...) . body))]))

(define-syntax (define: stx)
  (syntax-case stx (:)
    [(define: var : type expr)
     (identifier? #'var)
     (with-syntax ([tvar (syntax-property #'var 'type-label #'type)])
       #'(define #,tvar expr))]
    [(define: (f [formal formal-type] ...) : result-type . body)
     #'(define: f : (formal-type ... → result-type)
         (lambda: ([formal formal-type] ...) . body))]))
```

Figure 7. Typed definition and binding forms

4.1 Variables

Typed Scheme requires type annotations on bound variables; type-checking depends on that information. Consequently, the typed binding forms and the type-checker employ a protocol regarding the communication of variable types.

Type annotations are local to the terms where they appear. They must be robust in the face of local expansion and re-expansion. Since the type-checker works on the fully-expanded program, it makes sense to put the type annotations in the program. At the same time, the result of expansion is a core Scheme term, and Scheme's primitive syntactic forms are unaware of types and do not accept Typed Scheme's typed binding syntax. Syntax properties provide an appropriate method for implementing the protocol by attaching type information to terms.

> **The Variable Protocol:** Every typed binding form decorates its declared variables with a type attached to the 'type-label syntax property of the bound identifiers.

Typed Scheme implements the variable protocol by defining the typed binding forms such as **lambda:** as macros that convert the [*variable* : *type*] variable syntax into primitive binding forms with the types attached to the 'type-label syntax property of the variable names. Figure 7 shows the definition of typed binding macros. The **lambda:** macro expands into the primitive **lambda** form. For each formal parameter name, it creates a new syntax object with a 'type-label property holding the type. Likewise, the **define:** macro handles typed definitions. The first clause handles the simple case with just a name being bound to a value. The second clause handles the function definition syntax by desugaring it to a **define:** form with an explicit **lambda:** form. It also synthesizes the function type from the argument types and the result type, adding it to the expanded definition.

The type-checker, at the other end of the protocol, consumes the syntax properties produced by the typed binding forms. When the type-checker encounters a binding form, it scans the bound variables and extracts their types with the *get-id-type* procedure:

```
(module env mzscheme
  (provide (all-defined))

  ;; An environment is a (list-of binding).

  ;; A binding is (make-binding identifier type).
  (define-struct binding (id type))

  ;; the-type-env : environment
  ;; Associates global variables with their types.
  ;; Initially contains types for the mzscheme primitives.
  (define the-type-env omitted)

  ;; declare-type : identifier type → void
  ;; Add a type association to the global type environment.
  (define (declare-type! id type) omitted)

  ;; empty-env : environment
  ;; The empty lexical environment.
  (define empty-env null)

  ;; extend-env : environment (list-of binding) → environment
  (define (extend-env env bindings) omitted)

  ;; lookup-type : lexical-env identifier → type
  ;; Searches the lexical environment, then the global environment.
  (define (lookup-type env var) omitted))
```

Figure 8. Type Environment

```
;; get-id-type : identifier → type
(define-for-syntax (get-id-type id)
  (let ([type (syntax-property id 'type-label)])
    (unless type (raise-missing-type-error id))
    type))
```

The type-checker maintains a two-part type-environment. One part holds the types of global variables, including variables defined via **define:** and all primitive variables. The other part holds the lexical variables, such as those bound by **lambda:** and other local binding forms. Figure 8 shows the outline of the environment module. The *declare-type!* operation updates the global type environment; *extend-env* extends the local type environment; and *lookup-env* finds the type of an identifier, searching first the local bindings then the global bindings.

The type-checker consumes the information attached to bound variables. Figure 9 lists the code for the type-checker. When the type-checker encounters a definition, it extracts the type annotations from the bound identifiers and extends the type environment with the new type association. It finally checks that the declared type matches the type computed for the right-hand side expression. When the type-checker encounters an expression, it switches to expression mode.

The *type-check-expr* procedure computes the type of the expression. In the simplest case, variable reference, the type-checker just looks up the type in the type environment. If the variable is not present, the *lookup-env* procedure raises an error. When the type-checker sees a **lambda** form, it gathers the types of the bound variables and extends the type environment before checking the body in the extended environment. It also uses the types of the formals, in addition to the computed type of the body, to create the type of the function. Finally, the application case involves finding the type

```
;; type-check-top-level : syntax → void
(define-for-syntax (type-check-top-level form)
  (syntax-case form (define)
    [(define var expr)
     (let* ([var-type (get-id-type #'var)])
       (declare-type! #'var var-type)
       (let ([expr-type (type-check-expr #'expr empty-env)])
         (check-type var-type expr-type form)))]
    [expr
     (type-check-expr #'expr empty-env)]))

;; type-check-expr : syntax lexical-env → type
(define-for-syntax (type-check-expr expr env)
  (syntax-case expr (lambda #%app omitted)
    [var
     (identifier? #'var)
     (lookup-type env #'var)]
    [(lambda (formal ...) body)
     (let* ([formal-types
             (map get-id-type (syntax→list #'(formal ...)))]
            [formal-bindings
             (map make-binding
                  (syntax→list #'(formal ...))
                  formal-types)]
            [body-type
             (type-check-expr #'body
                              (extend-env env formal-bindings))])
       (make-function-type formal-types body-type))]
    [(#%app op arg ...)
     (let ([op-type (type-check-expr #'op env)]
           [arg-types
            (map (lambda (arg) (type-check-expr arg env))
                 (syntax→list #'(arg ...)))])
       (check-function-type op-type #'op)
       (check-types (function-type-params op-type)
                    op-types
                    expr)
       (function-type-result op-type))]
    [omitted]))
```

These functions are defined using **define-for-syntax**, which creates a value binding in the compile-time environment, so the **top-interaction** macro can use the procedures.

Figure 9. The type-checker

of the operator, verifying that it is a function type of the right arity, and checking the expected parameter types against the actual parameter types. If the application is valid, the result is the function's result type.

5. Typing Modules

Type-checking a typed module is more complicated than type-checking an isolated definition or expression. Module bodies may refer to variables that are neither primitive nor locally-defined, but imported from other modules. Furthermore, module exports must be protected from misuse in other modules, both typed and untyped.

As with a single definition or expression, type-checking a module involves fully expanding the contents of the module and then analyzing the expanded contents. Typed Scheme uses the module transformer hook to type-check the contents of the module.

The variable protocol handles variables whose definitions or bindings occur within the body of the module, but typing imported variables requires additional communication between typed modules. The revised protocol affects the way a typed module's exports are compiled.

There are three kinds of module interactions that typed modules can participate in:

1. A typed module requires an untyped module.
2. A typed module requires another typed module.
3. An untyped module requires a typed module.

The first case simply requires a method of importing untrusted code in such a way that it cannot break the type system's invariants, which demands appropriate input from the programmer. The other two cases determine the behavior of a typed module's exports. Those two cases essentially demand different behaviors from a typed module depending on the context it is imported into.

This section explains how Typed Scheme interacts with the module system. We begin with the simplest case, a typed module importing untyped code. This case can be explained in terms of just the import statement. Then we consider the case of a typed module importing another typed module, and we develop the basic typed-module framework. Finally, we show how to extend the behavior of exports to support the case of importing a typed module into an untyped context.

5.1 Untyped to Typed

Typed modules cannot use untyped modules without additional protection.[6] Instead, typed modules use a special **require/typed** form to import names at specific types. The **require/typed** form wraps the untyped imports with contracts [9] that enforce the programmer-supplied types. It also adds the name to the type environment with the specified type.

For example, the following use of **require/typed** imports the *find-files* procedure from a standard library module:

```
(require/typed (lib "file.ss")
  [find-files ((Path → Boolean) Path → (Listof Path))])
```

It is equivalent to the following code fragment:

```
(require (rename (lib "file.ss") unsafe-find-files find-files))^TRUST
(define: find-files : ((Path → Boolean) Path → (Listof Path))
  (contract (type→contract
             ((Path → Boolean) Path → (Listof Path)))
            unsafe-find-files
            'find-files
            '<typed-scheme>)^TRUST)
```

The TRUST annotation indicates a syntax property that directs the type-checker to accept the labeled expression as-is. The **contract** expression wraps the unsafe version of the *find-files* procedure with a contract derived from the given type. The last two arguments indicate the parties involved in the contract; if something goes wrong, one of the parties is blamed.

The *find-files* contract checks the procedure's arguments and result. If the untyped version of *find-files* returns a non-path result, the contract catches it and blames 'find-files before the faulty value can interfere with the typed program. The first argument contract is itself a higher-order contract, so the contract system wraps the function passed to *find-files* with a contract corresponding to the (*Path* → *Boolean*) type. This contract prevents the untyped *find-files* from calling the function with faulty arguments; if it does so, the contract system raises an error and blames 'find-files for the violation. The second argument contract is a first-order contract. It can only be violated if typed code supplies an argument of the wrong type, which cannot happen if the type system is sound. Finally, if *find-files* were to return something other than a list of paths, the contract system would stop the program and thus protect the typed code that expects to process the result.

[6] However, typed modules can safely import untyped *macro* libraries (such as **match**) if the macros do not expand into untyped, non-primitive variables.

5.2 Typed to Typed

Typed Scheme installs a **#%module-begin** macro that first performs the normal module expansion (using *local-expand*), analyzes the result, and produces a module body that follows a new *module variable protocol* that provides the type-checker with the types of module variables:

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(module-begin form ...)
     (type-check-module-body
      (local-expand #'(#%plain-module-begin form ...)
                    'module-begin
                    null))]))
```

Unlike the type-checking procedure for top-level forms, *type-check-module-body* not only type-checks the module body; it also transforms the code to produce the module body.

When one typed module requires another typed module, type-checking the first module requires knowing the types associated with the all of the definitions of the second module. The type-checker needs the types for all of the definitions, even the unexported ones, because an imported macro can expand into references to the unexported variables of the module it was defined in. This requires a new protocol, the module variable protocol.

Let us consider the protocol mechanisms introduced in Section 3. An imported identifier does not carry any syntax properties, so syntax properties alone are insufficient. Static binding provides a partial solution: instead of directly providing a variable, a typed module could instead provide a macro that expands into a use of the actual variable. The macro would place a type annotation on the reference as a syntax property. The problem with the static binding approach is that it annotates only the references that cross the public import/export boundary. Variable references introduced by imported macros, however, do not go through the static binding mechanism; they refer directly to the module variables. Since Typed Scheme aims to support macros, static binding is not a viable approach.

That leaves compile-time side effects. We extend the type environment table to include all known typed-module definitions instead of just primitives and local definitions. A typed module relies on the global type environment to contain types for all variables that appear within its body, and it guarantees that its client modules have access to its own type associations.

> **The Module Variable Protocol:** During the compilation of a typed module, the global type environment contains bindings for all definitions in all typed modules transitively required by the module being compiled.

Since a module's contributions to the global type environment need to be present during the compilation of every module that depends on it, we use the persistent effect pattern described in Section 3.2.3.

In addition to verifying the correctness of the module's contents, the *type-check-module-body* procedure also appends compile-time type declarations to the end of the module. We illustrate the effect of the module transformer on the following modules:

```
(module one typed-scheme
  (provide one)
  (define: one : number 1))

(module plus typed-scheme
  (provide plus1)
  (define: (plus1 [n : number]) : number
    (+ n one)))
```

The first module passes the type-checker, which also adds a type declaration for *one* to the end of the compiled module:

```
(compiled-module one
  (require typed-scheme)
  (provide one)
  (define one 1)
  (begin-for-syntax
    (declare-type! #'one (type number))))
```

The reference to *declare-type!* was inserted by a macro from the *typed-scheme* module. Even though *one* does not import the *env* module directly, the procedure is available indirectly through *typed-scheme*. Since *typed-scheme* imports *env* via **require-for-syntax**, it is correct to use *declare-type!* within the compile-time part of *one*.

When the compiler encounters the *plus* module, the module system invokes the compile-time part of *typed-scheme*, initializing the global type environment with the primitive bindings only. Then when the compiler encounters the import of *one* in the module body, it invokes the compile-time part of the *one* module, which then loads its type declaration for *one* into the type environment.

The *plus* module includes just one new definition, and the module transformer adds the corresponding declaration to the module:

```
(compiled-module plus
  (require typed-scheme)
  (provide plus)
  (define plus (lambda (n) (+ n 1)))
  (begin-for-syntax
    (declare-type! #'plus (type (number → number)))))
```

The two modules are able to communicate using *typed-scheme*'s type environment because the compile-time parts of the *one* module and the *plus* module share a single invocation of *typed-scheme* and thus a single invocation of the *env* module.

Figures 10 and 11 show the implementation of typed modules and the module variable protocol.

5.3 Typed to Untyped

When a typed module is imported into another typed module, it must provide its definitions and load the type declarations into the global type environment. The type-checker ensures that the exported values are used safely, so there is no need for run-time checking or wrapping.

In contrast, when a typed module is imported into an untyped module, it should protect its exports so that the untyped context cannot destroy its invariants. As in the "untyped to typed" case, we use contracts to enforce the type constraints of the definitions. For any defined variable, it is a simple matter to generate a definition that wraps the variable in the protection of the appropriate contract.

```
(module typed-scheme mzscheme
  (require-for-syntax type-check)
  (provide (rename module-begin #%module-begin)
           (rename top-interaction #%top-interaction)
           (all-from-except mzscheme
             #%module-begin #%top-interaction)
           define:
           lambda:)
  (define-syntax (module-begin stx)
    (syntax-case stx ()
      [(module-begin form ...)
       (type-check-module-body
        (local-expand #'(#%plain-module-begin form ...)
                      'module-begin
                      null))]))
  (define-syntax top-interaction omitted)
  (define-syntax define: omitted)
  (define-syntax lambda: omitted))
```

Figure 10. The *typed-scheme* module

For example, the *plus* module above has a *plus1* procedure with type (*Number* → *Number*). Given that information, we can generate *defensive-plus1*:

```
(define/contract defensive-plus1
    (type→contract (Number → Number))
  plus1)
```

The **define/contract** form is like a definition that uses **contract** explicitly, except that it automatically computes the blame parties.

A typed module, then, needs to provide one set of definitions to typed contexts and another set of definitions to untyped contexts. Of course, no module can actually change the contents of its **provide** clauses once it is compiled. Instead, it can provide a set of *indirection* macros that choose whether to expand into the trusting or defensive versions of exported names, assuming they can determine whether the importing context is typed or untyped. PLT Scheme provides *rename transformers* as a convenient way of writing such identifier-to-identifier translations.

Continuing the *plus* module example, the module transformer rewrites

```
(provide plus1)
```

into the following indirection definition and renamed-provide clause:

```
(define-syntax export-plus1
  (if omitted ;; Will it be used in a typed context?
      (make-rename-transformer #'plus1)
      (make-rename-transformer #'defensive-plus1)))
(provide (rename export-plus1 plus1))
```

The indirection definitions depend on some way of determining whether the context they are imported into is typed or untyped. The context that matters is the module currently being compiled. If the require chain includes intervening modules, they have already been compiled, and references within the compiled modules are already resolved to the right version of the exports. Thus, the problem boils down to determining whether the module currently being compiled is a typed module.

The property that distinguishes a typed module is that it specifies *typed-scheme* as its language module, and thus its module body is

```
(module type-check mzscheme
  (require env)
  (provide (all-defined))

  ;; type-check-top-level : syntax → void
  (define (type-check-top-level form) omitted)

  ;; type-check-module-body : syntax → syntax
  (define (type-check-module-body form)
    (syntax-case form ()
      [(module-begin top-level-form ...)
       (let ([definition-types
              (get-definition-types
               (syntax→list #'(top-level-form ...)))])
         (for-each (lambda (def)
                     (declare-type! (binding-id def)
                                    (binding-type def)))
                   definition-types)
         (for-each type-check-module-level-form
                   (syntax→list #'(top-level-form ...)))
         ;; Generate declarations to reload types into the
         ;; global type environment
         (with-syntax ([(type-declaration ...)
                        (map binding→type-declaraction
                             definition-types)])
           #'(module-begin top-level-form ...
                           type-declaration ...)))]))

  ;; type-check-module-level-form : syntax → void
  (define (type-check-module-level-form form) omitted)

  ;; type-check-expression : syntax environment → type
  (define (type-check-expression expr env) omitted)

  ;; get-definition-types : (list-of syntax) → (list-of binding)
  (define (get-definition-types forms)
    (if (null? forms)
        null
        (syntax-case (car forms) (define)
          [(define name rhs)
           (cons (make-binding #'name (get-id-type #'name))
                 (get-definition-types (cdr forms)))]
          [_ (get-definition-types (cdr forms))])))

  ;; get-id-type : identifier → type
  (define (get-id-type id) omitted)

  ;; binding→type-declaration : binding → syntax
  (define (binding→type-declaraction b)
    (with-syntax ([id (binding-id b)]
                  [type-expr
                   (type→type-expression (binding-type b))])
      #'(begin-for-syntax (declare-type! #'id type-expr))))

  ;; type→type-expression : type → syntax
  (define (type→type-expression type) omitted))
```

Figure 11. Type Checker

under the control of the typed module transformer. Given that, it is critical to understand the exact order of events in the compilation process:

1. The compiler invokes the initial language module's compile-time part.
2. Then, it executes the initial language module's module transformer on the body of the module being compiled.
3. As the compiler encounters **require**s in the module's body, it invokes the compile-time parts of the relevant modules.

In particular, the execution of the module transformer precedes the execution of any of the indirection definitions in compiled typed modules. The Typed Scheme module transformer can therefore set a flag indicating that the module being compiled is a typed module, and the indirection definitions can simply check the value of the flag.

Here is the modified *typed-scheme* module:

```
(module context mzscheme
  (provide typed-context?)
  ;; typed-context? : (box-of boolean)
  ;; True when the module being compiled is a typed module.
  (define typed-context? (box #f)))

(module typed-scheme mzscheme
  omitted
  (require-for-syntax context)
  (define-syntax (module-begin stx)
    (syntax-case stx ()
      [(module-begin form ...)
       (begin
         (set-box! typed-context #t)
         (type-check-module-body
          (local-expand #'(#%plain-module-begin form ...)
                        'module-begin
                        null)))]))
  omitted)
```

The *type-check* module also adds (**require** *context*) so that the indirection definitions it inserts can refer to *typed-context?*.

The following program illustrate how the flag works. We add an untyped *main* module to the *one* and *plus* modules from our earlier examples.

```
(module one typed-scheme
  (provide one)
  (define: one : number 1))

(module plus typed-scheme
  (require one)
  (provide plus1)
  (define: (plus1 [x : number]) : number
    (+ x one)))

(module main mzscheme
  (require plus)
  (display (plus1 41)) (newline))
```

The compiler processes the typed *one* module first, creating the context-dependent indirection definition for the exported variable *one*. When the compiler encounters the typed *plus* module, it first invokes the compile-time part of *typed-scheme*. That, in turn, causes the invocation of the *context* module, including a new *typed-context?* box initialized to false. Executing the Typed Scheme

#%module-begin macro sets the value in the *typed-context?* box to true. Subsequently, when the compiler encounters the (**require** *one*) form in the module body, it invokes *one*'s compile-time part. Since the *typed-context?* variable is set to true, the indirections are set to the typed variants, and the compiler resolves uses of the imported names to the unwrapped definitions.

The compilation of the *main* module proceeds differently. When the compiler encounters the (**require** *plus*) form, it invokes *plus*'s compile-time part, which invokes *typed-scheme*'s compile-time part and invokes *context*. This creates a fresh *typed-context?* box initialized to false, just as before. The box's value is never changed to true, however, because Typed Scheme's **#%module-begin** macro is not used in the expansion of the *main* module. Thus when *plus*'s indirection definitions are executed, they point to the contract-wrapped variants. Thus the occurrence of *plus1* in the *main* module is wrapped in code to verify the type of its argument.

6. Related Work

Experiments with adding types to Scheme go back more than twenty years. Wand's Semantic Prototyping System [22] uses macros to type-check implementations of denotational semantics. Typed Scheme scales up the approach of SPS to a modern macro and module system, and it improves the interaction between typed and untyped code.

Flanagan's static analyzer for DrScheme [12] analyzed expanded programs and used syntax source information to display the analysis results in the program editor. The analyzer included a macro protocol that let programmers annotate their programs with hints to the analyzer. Meunier's followup explored the use of type-imitating contracts to modularize static analysis [18]. Rather than analyzing the entire program, the analysis uses contracts placed at module boundaries to approximate the imported values.

The Ziggurat project [11] has investigated alternate approaches to static analysis and other program observations in the presence of macros. Analyses in Ziggurat are implemented as methods on expression nodes, and derived expression forms (that is, macros) may either override analysis methods with special behavior or defer to the default analysis of the macro's expansion. Ziggurat represents a new approach to defining macro protocols, and it is as yet unclear how the Ziggurat approach compares with the mechanisms described here.

References

[1] Eli Barzilay and John Clements. Laziness without all the hard work: combining lazy and strict languages for teaching. In *FDPE '05: Proceedings of the 2005 workshop on Functional and declarative programming in education*, pages 9–13, New York, NY, USA, 2005. ACM Press.

[2] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. 10th European Symposium on Programming Languages and Systems*, pages 320–334, 2001.

[3] John Clements, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Little languages and their programming environments. In *Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 1–18, June 2001.

[4] William Clinger and Jonathan Rees. Macros that work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[5] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proc. Fourth International Conference on Generative Programming and Component Engineering*, pages 373–388, 2005.

[6] R. Kent Dybvig. Writing hygienic macros in scheme with syntax-case. Technical Report TR 356, Indiana University, June 1992.

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

[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. Building little languages with macros. *Dr. Dobb's Journal*, 2004.

[9] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[10] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In Hugh Glaser, Pieter Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 369–388, Southampton, UK, September 1997. Springer.

[11] David Fisher and Olin Shivers. Static analysis for syntax objects. In *ACM SIGPLAN International Conference on Functional Programming*, 2006.

[12] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[13] Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

[14] Matthew Flatt. Algol 60 implementation, 2007. Available from http://www.plt-scheme.org/.

[15] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming*, 2000.

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

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

[18] Philippe Meunier, Robby Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2006.

[19] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *ACM SIGPLAN International Conference on Functional Programming*, September 2005.

[20] Sam Tobin-Hochstadt. Typed Scheme. `http://www.ccs.neu.edu/~samth/typed-scheme.html`, 2007.

[21] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 964–974, New York, NY, USA, 2006. ACM Press.

[22] Mitch Wand. A semantic prototyping system. In *ACM SIGPLAN Symposium on Compiler Construction*, 1984.

[23] Andrew Wright and Bruce Duba. Pattern matching for Scheme, 1995. Unpublished manuscript.