First-class Macros Have Types

Alan Bawden*

Boston University
bawden@cs.bu.edu

Abstract

In modern Scheme, a macro captures the lexical environment where it is defined. This creates an opportunity for extending Scheme so that macros are first-class values. The key to achieving this goal, while preserving the ability to compile programs into reasonable code, is the addition of a type system. Many interesting things can be done with first-class macros, including the construction of a useful module system in which modules are also first-class.

Clams got legs!
—*B.C.*

1 Introduction

A revolution in macro technology has taken place over the last 15 years in the Scheme community. Classical Lisp macros operate as pure source-to-source transformations on S-expressions [Pit80, Ste90], just as `C`'s macros operate on tokens, or other macro systems operate on characters. Such macros have no understanding of the structure of the language they are generating, and in particular, they are blind to the scoping of variables. As a result, classical Lisp macros sometimes have bugs caused by inadvertent variable captures.

Modern Scheme macros [KFFD86, BR88, CR91, Cli91, DHB92, KCR98] have the ability to avoid capture problems by generating output that specifies the environment in which each variable is to be resolved. The programmer can easily write macros that behave properly with respect to the rules of lexical scoping. The details of this "namespace management" technology vary from implementation to implementation, but always the idea is that a macro should somehow capture the environment where it is defined.

This environment capture bears a resemblance to the way environments are captured in "closures" in order to implement first-class procedures. This is not surprising since in both cases the goal is to respect lexical scoping. It suggests that perhaps some closure-like mechanism might give us *first-class macros*, in the same way that ordinary closures give us first-class procedures.

This paper describes some extensions to Scheme which I have implemented that make macros into first-class values. In this system, macros can be passed to procedures as arguments, returned as values, and stored in data structures. And this is done *without* sacrificing the ability to compile into reasonable code.

The key idea is that a macro, considered as a first-class value, has a kind of type. Programs without type errors are programs that can be reasonably compiled.

Although this paper is not primarily about module systems, the extensions presented here can be thought of as low-level primitives for constructing module systems. One of the sample applications presented below is a fully functional module system that supports first-class modules and separate compilation.

Section 2 develops a small example that will serve as motivation for what follows. Section 3 describes the extensions to the Scheme language that support first-class macros. Section 4 explains the simple type system used. Section 5 presents two examples of what can be done in this system: a record structure package, and a module system. Section 6 describes the prototype compiler. Section 7 presents an interesting unexpected natural feature of the system that is of unknown utility. Section 8 presents some loose ends and conclusions.

2 Motivation

In the code that follows, I will write macros using the "explicit renaming" technology for namespace management [Cli91]. I will also only do the renamings that are strictly necessary for expository purposes—I will not clutter macro definitions with renamings that guard against possible shadowings of standard keywords and procedures such as `lambda` and `car`. In practice, you would do a lot more renaming, or you would use something like Scheme's `syntax-rules` that does renaming for you automatically.

Consider the following implementation of the standard Scheme `force` and `delay` ("promises") facility:

*This work was supported by NSF grant EIA-9806718.

```
(define delay
  (macro
    (lambda (form rename)
      `(,(rename 'make-promise)
        (lambda () ,@(cdr form))))))

(define make-promise
  (lambda (thunk)
    thunk))

(define force
  (lambda (promise)
    (promise)))
```

This implementation fails to meet the language specification [KCR98] because it does not cache the result of forcing a promise, but the programmer may know that the caching is wasted effort. She may prefer this implementation for certain applications.

Imagine that we enclose this implementation in a module using some module system such as Waddell and Dybvig's [WD99], and that the module exports `force` and `delay`, but keeps `make-promise` private. Recall that modern scheme macro namespace management technology will insure that the reference to `make-promise` inserted when a `delay` expression is expanded will resolve to the `make-promise` defined in this module—that is what the call to `rename` accomplishes.

Further imagine that the following alternate implementation of promises was placed in a second module:

```
(define delay
  (macro
    (lambda (form rename)
      `(,(rename 'make-promise)
        (lambda () ,@(cdr form))))))

(define make-promise
  (lambda (thunk)
    (cons #F thunk)))

(define force
  (lambda (promise)
    (if (car promise)
        (cdr promise)
        (let ((val ((cdr promise))))
          (if (car promise)
              (cdr promise)
              (begin (set-car! promise #T)
                     (set-cdr! promise val)
                     val))))))
```

This implementation fully implements the language specification, at the expense of some additional overhead.[1]

Although this definition of `delay` is identical to the definition of `delay` in the first module, they differ in that the the reference to `make-promise` inserted by *this* version of `delay` will resolve to the `make-promise` defined in *this* module.

The programmer may prefer either of these promise implementations for a given application. Using the module system, she has the option of opening and using the appropriate one.

But that is not the end of this software engineering story. The programmer uses promises a lot in her programs, and she often finds the following procedure for making lazy lists useful:

[1] The second `(car promise)` test is required by the standard.

```
(define lazy-map
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l))
              (delay
                (lazy-map f (cdr l)))))))
```

The version of `delay` being used here depends on which version of the promise module was opened in the environment where this definition appears. When the programmer wants to use `lazy-map` with the *other* implementation of promises she must *repeat* the definition of `lazy-map` in an environment were she opens up the other promise module.

The programmer would rather not duplicate the code for `lazy-map` like this. Instead, she would like to put *one* copy of it in a module of useful promise utilities, and use it with *both* promise implementations. If `delay` were a procedure, instead of being a macro, she could accomplish this by giving `lazy-map` an additional parameter:

```
(define lazy-map
  (lambda (delay f l)
    (if (null? l)
        '()
        (cons (f (car l))
              (delay
                (lazy-map delay f (cdr l)))))))
```

Then when she called `lazy-map`, she would pass it the version of `delay` from the implementation of promises that she wanted to use. But unfortunately macros are not first-class values, they cannot be passed as arguments to procedures, and so this solution will not work in Scheme.

The rest of this paper describes an extension to Scheme that will let the above solution work almost exactly as written above. Macros will become first-class values that *can* be passed as arguments, returned as values, and stored in data structures. And we will be able to do this *without* sacrificing the ability to compile Scheme into efficient code.

The reason we might worry about compilation, is that a naïve interpretation of what it means to pass a macro (or other keyword) as an argument naturally concludes that:

```
(lazy-map quote list '(1 2))
```

should evaluate to:

```
((1) lazy-map delay f (cdr l))
```

That is, the `delay`-expression in the body of `lazy-map` should become a `quote`-expression if the keyword `quote` is passed as the value of the `delay` parameter. In effect, the body of `lazy-map` would become:

```
(if (null? l)
    '()
    (cons (f (car l))
          (quote
            (lazy-map delay f (cdr l)))))
```

It is obviously quite difficult to generate reasonable compiled code for `lazy-map` if the compiler must anticipate that `delay` (or any of the other parameters!) might be a macro with an arbitrary definition.

But our imagined programmer does not need such excessive generality. All she wants is to be able to switch between definitions of `delay`. And the two different versions of `delay`

will, in fact, expand into nearly identical code—the only difference being the environment where the (renamed) variable `make-promise` will be found. This should not be hard to compile. The programmer is not really interested in passing in *arbitrary* macros as the first argument to `lazy-map`, she only wants to be able to pass in macros that are (in some sense we need to make precise) of the correct type.

Given an appropriate notion of the *type* of a macro, our programmer will be able to write:

```
(define lazy-map
  (lambda (delay f l)
    (declare (delay <delay>))
    (if (null? l)
        '()
        (cons (f (car l))
              (delay
                (lazy-map delay f (cdr l)))))))
```

Where `(declare (delay <delay>))` is a type declaration that indicates that the variable `delay` is to have the type `<delay>`.[2] In the next section we will see how to go about constructing a suitable `<delay>` type that describes exactly what the compiler needs to know in order to compile this code.

3 The Extensions

The scenario described in the previous section assumed the presence of a module system. We are actually going to solve the more general problem of first-class macros defined in an arbitrary environment. In section 5.2, I will show how to construct a module system on top of this more general foundation.

3.1 Templates

A `template`-expression describes everything a compiler would need to know about the environment where a macro (or a set of macros) is defined. For example:

```
(define promise-template
  (template
    (delay (macro
             (lambda (form rename)
               `(,(rename 'make-promise)
                 (lambda () ,@(cdr form))))))
    (make-promise (value <plain>))
    (force (value <plain>))))
```

defines `promise-template` to be a template that describes an environment where a macro named `delay` and two other values named `make-promise` and `force` are defined. The definition of the macro is given, as well as the types of the other two values. (The type `<plain>` contains all of the ordinary, dynamically typed, Scheme values.)

This template does not describe a *complete* environment, it describes just the scope that will immediately surround the definitions of those three names. In operational terms, it describes the top-most frame in an environment chain—it tells the compiler exactly what it needs to know in order to design a runtime representation for that frame.

When a macro defined in a template expands, any identifiers it renames (using the `rename` procedure it will be passed) will resolve in an environment constructed from the environment where the `template`-expression was written, extended with the names listed in the template itself. The runtime value of such a macro will just be an environment frame of the form described by the template.

A template is not itself an environment. An `instantiate`-expression (described below), must be used in order to actually make an instance of the template.[3] A template is a bit like the "interfaces" found in some module systems [Mac84, CR90, Ree93], but without any name hiding mechanisms.

Having defined a template, now we can define the type that describes the different `delay` macros we might get from various different instantiations of the template:

```
(define <delay>
  (type-of promise-template delay))
```

A `type-of`-expression obtains the type of any of the values described in a template.

This two-step process to actually arrive at the type definition is necessary because a template might define several different macros at the same time, and we might need to obtain types for all of them. Alternatively, we could refrain from naming the type, and just directly write

```
(declare
  (delay (type-of promise-template delay)))
```

in our procedure definitions—but this gets tedious.

The definitions of `promise-template` and `<delay>` are sufficient to allow the code at the end of the previous section to compile. Knowing that `delay` is of type `<delay>`, the compiler will be able to use the macro definition given in the template to expand the `delay`-expression into a call to some `make-promise` procedure. At runtime, when `lazy-map` is called, the value passed as its first argument will be an environment frame, and the appropriate `make-promise` procedure will be found at a known location within that frame.

3.2 Instantiation

In order to make an instance of a template, we use `instantiate`. For example:

```
(instantiate promise-template
  (set! make-promise
        (lambda (thunk)
          thunk))
  (set! force
        (lambda (promise)
          (promise)))
  ...)
```

`instantiate` is a binding form (like `let`) that extends the environment where the `instantiate`-expression was written by binding the variables named in the template (`delay`, `make-promise` and `force` in this case), and then executes its body. The value of the last expression in the body is returned. At runtime the actual environment frame created

[2] `declare` is not a new special form, but is rather an extension to the syntax of `lambda`-expressions. This is a bit ugly, but all the alternatives seem worse to me. I find the obvious alternative of placing the types in the list of bound variables to be very cluttered and hard to read.

[3] For the moment, do not worry about whether a template is itself a first-class value—we will return to that issue.

by `instantiate` will use the representation specified by the given template.

Variables that were specified using `value` in the template (`make-promise` and `force` in this case) are bound, and given their specified types, but are left unassigned. The notion of a bound-but-unassigned variable is not found in standard Scheme, so there is no precedent for how to specify an initial value for such a variable. I have chosen to simply use `set!`, which is perfectly clear semantically, but which does give the code an unfortunate imperative flavor. A purely functional language would probably adopt some other solution.

After the code in the body of an `instantiate`-expression has finished the job of initializing the template instance, it will ordinarily want to return some useful value. In the case of `promise-template`, we want it to return the `force` procedure, *and* the `delay` macro. (The ability to return the latter value being the point of this entire exercise!) We *could* get both these values out by using Scheme's multiple values feature. A solution that will prove more useful to us in the long run, is to go back and add a second macro definition to the promise template as follows:

```
(define promise-template
  (template
    ...
    (self (macro
            (lambda (form rename)
              (rename (cadr form)))))))
```

This defines `self` as a macro such that `(self foo)` expands into `foo`, where `foo` is to be resolved in the environment where `self` was defined. So `(self force)` will be the `force` procedure, and `(self delay)` will be the `delay` macro. So now we can return *both* those values by just returning the value of `self`:

```
(define uncached-promises
  (instantiate promise-template
    (set! make-promise ...)
    (set! force ...)
    self)
```

Now `(uncached-promises delay)` expands into the `delay` macro that was created when the `instantiate`-expression was entered. When that `delay` macro is used, it will generate code that includes a reference to the variable `make-promise` from that same environment. So finally, our promise-loving programmer can write:

```
(lazy-map (uncached-promises delay)
          expensive-operation
          huge-list)
```

This trick of defining a macro like `self` is a useful technique that we will employ several times in the following sections.

The way `self` works suggests an interesting way to think about first-class macros: First-class macros can be viewed as a limited form of first-class environments. True first-class environments would allow programmers to manipulate environments as first-class values, and to access arbitrary variables bound inside them. In contrast, first-class macros allow programmers to (in effect) manipulate environments as first-class values, but with a controlled form of access to the enclosed variables: access is only available via code generated by expanding the macro. A macro like `self` is willing to grant access to any variable in its environment at all, but other environments can be protected by less permissive guardians.

4 About The Types

Having introduced static types for macros into an otherwise dynamically typed language, I need to answer a few questions about the nature of this type system. Where do new types come from and how does the compiler reason about them?

In fact, this is an extremely simple type system (it is very similar to the way types work in traditional C):

- New base types are generated by macro definitions appearing in `template` expressions.
- The types of all procedure arguments must be explicitly declared by the programmer. (Arguments not mentioned in a `declare` clause are always of type `<plain>`.)
- A procedure has a type determined by the type of its return value and the types of its arguments.[4]
- Type equality is determined by a simple recursive comparison. Base types are only equal to themselves. Procedure types are equal if their return values are equal and corresponding arguments are equal.

The compiler performs only type checking. No type inference is needed. There are no polymorphic types. The only purpose of the type system is to ensure that: when an environment frame serving as the representation of a macro is accessed by code generated by the expansion of a macro, the macro being represented at runtime is the same as the macro that was expanded at compile time.

There is no fundamental reason why this macro type system needs to be static. The same safety could be achieved using dynamic typing. Type declarations would still be needed for macro valued procedure arguments. The runtime representation for a macro value would be an environment frame, *plus* a tag that identifies the macro. The compiler would emit code to check the tags at runtime to make sure that a value used as an operator always had the type that the programmer promised the compiler it would have.

There would be many advantages to using dynamic typing. The main advantage would be that the built-in Scheme primitives could manipulate macro values directly. For example, under static typing the arguments and return value of the built-in `cons` procedure are all of type `<plain>`, so a program that tries to use `cons` to build a list of values of type `<delay>` will fail to type check, making it impossible to build a list that contains a value of type `<delay>`. But if we use dynamic typing, we can pass any value at all to `cons`, because runtime type checks will ultimately prevent any macro values from being misused.[5]

Despite the advantages of dynamic typing (and despite the fact that I am normally an advocate of dynamically typed languages) my prototype implementation uses static types for macro values because:

- Simple static typing is easier to implement than dynamic typing. The compiler checks the programmer's type declarations at compile time and that is the end of it. There is no need to design a tagging scheme or figure out where to insert type checks in compiled code.

[4] The type of `lazy-map` can be written:

`(procedure <plain> <delay> <plain> <plain>)`.

[5] A polymorphic type system would also partly address this shortcoming.

- If I had built a dynamically typed prototype, some readers might not have realized that this idea would be applicable in languages that were not dynamically typed. This way it should be clear that this idea is perfectly applicable in a statically typed language.
- Traditionally, macro expansion leaves no residue behind in generated code. Thus the runtime type checks required for dynamic typing of macro values would violate some people's expectations of what it means for something to be a "macro."

5 Applications

This section presents two illustrative examples of using first-class macros in practice. The first example, perhaps the more surprising of the two, demonstrates how first-class macros can be used to define a complete record structure package. The second example shows how templates and their instances can be used to implement interfaces and modules.

5.1 Structures

We have got types introduced by macro definitions and procedure types constructed from them. Will we also need to add tuple types in order to be able to store first-class macros in data structures? As we noted at the end of section 4, one way to avoid problems like this is to switch to dynamic typing and then just use the existing Scheme list and vector types. But if a switch to dynamic typing is not an option (as would be the case if we were adding first-class macros to C), will it be necessary to add more machinery for defining record types that can hold first-class macro values? Fortunately not—we already have all the tools we need to define record types ourselves.

Our goal is to be able to write structure definitions such as:

```
(define-structure <kons> kons
  (car <delay>)
  (cdr <kons>))
```

Which declares a structure of type `<kons>`. A `<kons>` can be constructed by calling the `kons` procedure. It contains a `car` slot of type `<delay>` and a `cdr` slot of type `<kons>`. A chain of `<kons>`'s could be used to build a list of values of type `<delay>`, should that prove necessary...

Here is how we begin:

```
(define define-structure
  (macro
    (lambda (form rename)
      (let ((type (list-ref form 1))
            (make (list-ref form 2))
            (ids (map car
                      (list-tail form 3)))
            (types (map cadr
                        (list-tail form 3)))
            (template (rename 'template))
            (self (rename 'self)))
        `(begin
           (define ,type
             (type-of ,template ,self))
           ,(def-struct-template template self
                                 type ids
                                 types)
           ,(def-constructor template self
                             type ids types
                             make rename))))))
```

This tears the `define-structure`-expression apart into its components; creates two new identifiers to be used internally to name a template (`template`) and to be a self macro defined within that template (`self`); and generates a definition for the given type variable. Two sub-procedures are called to generate the template definition (`def-struct-template`) and the constructor definition (`def-constructor`).

```
(define (def-struct-template template self
                             type ids
                             types)
  `(define ,template
     (template
       (,self (macro (struct-self ',type
                                  ',ids)))
       ,@(map (lambda (i t)
                `(,i (value ,t)))
              ids
              types))))
```

All `def-struct-template` does is use the given slot names and types to generate a template definition, with an additional self macro constructed by `struct-self`:

```
(define (struct-self name ids)
  (lambda (form rename)
    (if (memq (cadr form) ids)
        (rename (cadr form))
        (error "Unknown slot" name form))))
```

A call to `struct-self` creates a macro transformer that functions like the `self` macro we defined in section 3.2, except it does some error checking to make sure it is only used to access the slots of the structure.

Finally, `def-constructor` builds a definition for the constructor procedure:

```
(define (def-constructor template self
                         type ids types
                         make rename)
  (let ((bvl (map rename ids)))
    `(define ,make
       (lambda ,bvl
         (declare (returns ,type)
                  ,@(map (lambda (v t)
                           `(,v ,t))
                         bvl
                         types))
         (instantiate ,template
           ,@(map (lambda (i v)
                    `(set! ,i ,v))
                  ids
                  bvl)
           ,self)))))
```

It generates a list of variables for the arguments, a declaration for the type of the procedure, an `instantiate` form to make an instance of the template, a sequence of initializations to the variables in the template instance, and finally returns the self macro.

That is all there is to it. Now we can create a new `<kons>`:

```
(define x (kons a b))
```

examine its `car`:

```
(x car)
```

and even change its `car`:

```
(set! (x car) new)
```

The code generated when accessing and modifying a `<kons>` is about as good as you could ask for. We will take a look at it in section 6.

There is no runtime type checking when using these structures—the typing is all statically checked at compile time. So there might be applications where a Scheme programmer would use this `define-structure` facility to avoid the overhead of dynamic type checking. But the interesting thing about this example is *not* the obvious fact that a statically typed record facility can be more efficient than a dynamically typed one. The interesting thing is that first-class macros are sufficiently powerful to construct a complete record structure package.

5.2 Modules

As I remarked before, templates strongly resemble the interfaces found in other module systems [Mac84, CR90, Ree93], but without any name hiding mechanism. In this section we will use templates and their instances to represent the interfaces and modules in a fully functional module system. Since this module system is constructed on top of first-class macros, our *modules* will be first-class as well. This module system also easily supports separate compilation.

Continuing with our promises example, here is how we would like to define the promises interface:

```
(define-interface promise-interface
      <promise-module>
      (force delay)
      ((<delay> delay))
  (delay (macro (lambda (form rename)
                  `(,(rename 'make-promise)
                    (lambda ()
                      ,@(cdr form))))))
  (make-promise (value <plain>))
  (force (value <plain>)))
```

This defines `promise-interface` to be the interface to modules implementing promises. `<promise-module>` is defined to be the type of such modules. The list `(force delay)` specifies which names in a promise module are to be exported. Next is a list of types to be defined—in this case `<delay>` is to be defined as the type of `delay` in a promise module. There then follows a list of variable specifications exactly as they would appear in a plain `template`-expression.

After seeing this definition, the compiler will be able to compile any program that uses a promise module. For example, the definition of `lazy-map`.

To instantiate a promise module, we will use a `module`-expression:

```
(define uncached-promises
  (module promise-interface
    (set! make-promise
          (lambda (thunk)
            thunk))
    (set! force
          (lambda (promise)
            (promise)))))
```

Here `promise-interface` is the interface we are instantiating, the body initializes the module, and the value returned is the module itself (a value of type `<promise-module>`).

Finally, to import the exported definitions of a module into the current environment, we open it as follows:

```
(open uncached-promises)
```

So we have three things to define: `define-interface`, `module` and `open`. `define-interface` looks a lot like `define-structure` did:

```
(define define-interface
  (macro
    (lambda (form rename)
      (let ((name (list-ref form 1))
            (type (list-ref form 2))
            (exported (list-ref form 3))
            (typedefs (list-ref form 4))
            (specs (list-tail form 5))
            (template (rename 'template))
            (self (rename 'self)))
        `(begin
           (define ,type
             (type-of ,template ,self))
           ,@(map
               (lambda (typedef)
                 `(define ,(car typedef)
                    (type-of ,template
                             ,(cadr typedef))))
               typedefs)
           (define ,name
             (macro (interface-self ',template
                                    ',self)))
           (define ,template
             (template
               (,self (macro (module-self
                              ',exported)))
               ,@specs)))))))
```

It tears the `define-interface`-expression apart, generates new `template` and `self` identifiers, and creates a pile of definitions. All but the last two definitions define type names requested by the user. The second-to-last definition defines the interface name to be a macro with a transformer built by `interface-self`—this macro represents the interface. The last definition defines the template, adding a definition for a `self` macro (by now a familiar cliché) whose transformer will be constructed by `module-self`—instances of this macro will represent modules.

The only thing you can do with the instance itself is to use `module` to instantiate it, so here is its definition:

```
(define module
  (macro
    (lambda (form rename)
      `(,(cadr form) ,@(cddr form)))))
```

All this does is pass the buck to the interface macro, which was created by `interface-self`:

```
(define (interface-self template self)
  (lambda (form rename)
    `(instantiate ,template
       ,@(cdr form)
       ,self)))
```

This just plugs the body of the `module`-expression into an `instantiate`-expression and returns the self macro.

The only thing you can do with a module is to use `open` to open it, so here is its definition:

```
(define open
  (macro
    (lambda (form rename)
      `(,(cadr form)))))
```

This just passes the buck to the module macro, which was created by `module-self`:

```
(define (module-self exported)
  (lambda (form rename)
    `(begin ,@(map (lambda (id)
                     `(define-alias ,id
                        ,(rename id)))
                   exported))))
```

This creates a bunch of aliases in the current environment for the variables exported from the module. (`define-alias` is borrowed from Waddell and Dybvig [WD99] who use it for a similar purpose in their module system.)

Note that if you are not using first-class macros or first-class modules, you do not need to use types anywhere in your program in order to use this module system.

6 Implementation

The prototype implementation works by compiling Scheme plus first-class macros into standard Scheme. At first this sounds like an easy task: just expand all the macros and you are done. But additional code must be generated to manipulate the extra environment frames needed to represent instantiated templates (and hence first-class macros themselves), so the process is more than just a simple macro expansion.

These extra frames are represented using Scheme vectors, while all the standard environment frames are left implicit in the generated Scheme code. This compilation technique has the advantage that the reader can easily find the code that was generated to support the first-class macros.

I will demonstrate the compiler by walking through some examples based on the record structure code from section 5.1, and in particular, assuming the following structure definition:

```
(define-structure <kons> kons
  (car <delay>)
  (cdr <kons>))
```

This `define-structure` expression will expand into the following three definitions, where `template_1`, `self_2`, `car_3` and `cdr_4` are the identifiers generated by the `rename` procedure when the macro wanted fresh identifiers:

```
(define <kons>
  (type-of template_1 self_2))

(define template_1
  (template
    (self_2 (macro (struct-self
                     '<kons>
                     '(car cdr))))
    (car (value <delay>))
    (cdr (value <kons>))))

(define kons
  (lambda (car_3 cdr_4)
    (declare (returns <kons>)
             (car_3 <delay>)
             (cdr_4 <kons>))
    (instantiate template_1
      (set! car car_3)
      (set! cdr cdr_4)
      self_2)))
```

When the compiler processes the `template` expression for `template_1`, it must design an environment frame in which variables `self_2`, `car` and `cdr` are defined.

`self_2` will be defined as a macro—all the compiler knows about that macro is that any identifiers it renames will be looked up first in this frame, and if they are not found there, in the environment where the `template` expression occurred. You might expect that in order to implement this name lookup, such a frame would need to contain a pointer to the environment frame for the surrounding environment. But in fact this is not necessary.

The reason for this is subtle: roughly, a top-level `template` expression, such as this one, is not itself a first-class value.[6] Therefore the macro types it creates are only available within the scope where this template is defined. So it is impossible to *use* those macros outside of that scope. And thus the variables in the environment surrounding the `template` expression can always be located somewhere up the chain of ordinary environment frames.

You might also expect that the compiler would need to allocate a slot to contain the value of `self_2`. But the run-time representation of the `self_2` macro will be the frame *itself*, so no such slot is needed.

Thus the compiler designs a frame containing two slots for instances of `template_1`—the first slot will contain the value of the variable `car` and the second will contain the value of the variable `cdr`.

We now consider the compilation of the definition of the constructor function `kons`. The only interesting issue is the compilation of the `instantiate` expression. The compiler allocates a frame of the appropriate size, and then arranges that inside the body of the `instantiate` expression, `car` and `cdr` will refer to the first and second slots of that frame, and that `self_2` will refer to the frame itself. The generated code is:

```
(define kons
  (lambda (car_3 cdr_4)
    (let ((instance_5 (make-vector 2)))
      (vector-set! instance_5 0 car_3)
      (vector-set! instance_5 1 cdr_4)
      instance_5)))
```

[6] But see the next section.

Finally, consider the compilation of the definition:

```
(define kar
  (lambda (x)
    (declare (x <kons>)
             (returns <delay>))
    (x car)))
```

The interesting issue here is the compilation of `(x car)` in the presence of a declaration that `x` has type `<kons>`. The type declaration allows the compiler to locate the expander procedure (generated by `struct-self`). The compiler calls this expander, passing it the expression `(x car)` and a specially constructed `rename` procedure that knows that `car` and `cdr` should refer to the first and second slots of the frame that is the value of `x`, and that `self_2` should refer to `x` itself. The result is:

```
(define kar
  (lambda (x)
    (vector-ref x 0)))
```

Following the same pattern, the code generated for `lazy-map`, as defined in sections 2 and 3, is:

```
(define lazy-map
  (lambda (delay-env f l)
    (if (null? l)
        '()
        (cons (f (car l))
              ((vector-ref delay-env 0)
               (lambda ()
                 (lazy-map delay-env
                           f
                           (cdr l))))))))
```

Here the compiler has designed a frame for `promise-template` where the procedure `make-promise` is located in the first slot.

The algorithm for compiling an expression is to first determine the type of its first sub-expression (the operator, its `car`). That type determines everything about how to compile the entire expression. In the case where the type is `<plain>` (the type of ordinary Scheme values), the expression is compiled as a procedure call, where the rest of the sub-expressions are argument expressions. In the case where the type is a macro type, the expression is compiled as follows:

```
(let ((env_17 compute-operator))
  result-of-expansion)
```

Where *compute-operator* is the code generated when the operator expression was compiled, `env_17` is a freshly generated identifier, and *result-of-expansion*, is the result of expanding the macro using a specially constructed `rename` procedure that knows where the various variables that might be inserted by the macro are located inside `env_17`.

7 First-Class Templates

Templates were introduced in order to make macros into first-class values. Until now, we have not considered the possibility that templates themselves might be first-class values. As it happens, we can obtain first-class templates with very little additional work. The same trick that worked to make macros first-class, will also work to make templates first-class: we simply allow templates to appear inside `template` expressions![7]

Unfortunately, it seems to be very hard to generate a plausible example of why one would *need* templates to be first-class. A likely scenario might be where a module interface exports a template that users of the module are expected to instantiate. Something like:

```
(define-interface promise-interface
      <promise-module>
      (force delay kons-tmplt)
      ((<delay> delay)
       (<kons-tmplt> kons-tmplt)
       (<kons> kons-tmplt self))
  (kons-tmplt (template
                (car (value <delay>))
                (cdr (value <kons>))
                (self (macro ...))))
  (delay (macro ...))
  (make-promise (value <plain>))
  (force (value <plain>)))
```

All of the examples I have been able to construct that have this structure have some at-least-as-good solution that does not involve first-class templates. Often simply exporting a procedure that instantiates a non-first-class template works just as well. I would be interested in hearing from anybody who thinks they can construct a really compelling case for first-class templates.

8 Conclusions

An interpreted system with first-class macros was presented by Jonathan Rees as the first step in explaining the development of a module system for Scheme [Ree89]. His final system, in which code can actually be compiled, no longer supported first-class macros. The system presented here results from my search for a way to save first-class macros from Jonathan's trash can.

This system also owes a debt to the module system proposed by Oscar Waddell and Kent Dybvig [WD99]. At the conclusion of the presentation of their system at POPL'99, an audience member asked why no additional runtime environment structures were needed in order to support their system. Thinking about when you *would* need additional environment structures in a system such as theirs helped lead me to the system here.

Compilers have always generated different code depending on the types of variables in expressions. In a `C` program, an expression like "`x + 1`" compiles into different machine instructions depending on whether `x` is an integer, floating point, or pointer variable. First-class macros deliver this ability to do type-driven code-generation directly into the programmer's hands. With first-class macros, `(x car)` compiles into a structure reference if `x` is of the type `<kons>` defined in section 5.1, but if `x` has a different type it may compile quite differently—the programmer is in complete control.

I am sure that I have only scratched the surface of the the interesting things that can be done with first-class macros. Consider the simple "syntactic protocol" employed by the

[7] Only one small change in the compiler is required in order to make this work: environment frames built for inner templates will require a pointer to the environment frame for the containing template.

`open` macro define in section 5.2. It can be used to open a module, as intended, but it can also be used to open something else—`open` is "generic." With a simple syntactic message passing protocol (a dispatch on a symbol used as the first operand), a first-class macro can support different syntactic "operations." The result is a sort of syntactic analog of object-oriented programming. All this territory remains to be explored.

There is no type-inference here. Types must be completely declared in every `lambda`-expression. It would be interesting to try to add some form of type-inference. The difficulty I foresee is that type-inference depends on knowing data flow, and data flow depends on having parsed the program, and first-class macros make parsing depend on types, closing the loop.

Some limited form of type-inference may be possible. If an argument to a procedure is never used in operator position, its macro type does not need to be precisely known at compile time. So some polymorphism is still possible. For example

```
(lambda (x) x)
```

can still be compiled and given the type $\alpha \to \alpha$, while

```
(lambda (f) (f))
```

can not be compiled without knowing exactly what the type of `f` will be.

Although it is hard to see how to make macro types culturally compatible with the ML type system, it is quite easy to see how combine them with the `C` (or other traditional) type system. In place of `<plain>`, our single non-macro type, substitute `int`, `char`, `struct point *`, etc.

There are many features of this system that are less than satisfactory. The `declare` syntax added to `lambda`-expressions is ugly. The body of a `template` really should look like a sequence of ordinary definitions. (So that macros that expand into a sequence of definitions could be used there.) Using `set!` to initialize the values in a `instantiate`-expression seems out-of-place in a mostly functional language like Scheme. But none of these problems seem to be more than ordinary issues of programming language design—none are fatal flaws.

Despite all the mechanisms and syntax introduced here, the key observation is actually very simple: With the support of a type system, macros can become first-class values, and the result is a useful and powerful new programming tool.

Acknowledgments

I had helpful discussions with Olin Shivers and Harry Mairson while developing these ideas. Glenn Burke and Olivier Danvy provided useful feedback on drafts of this paper. Thanks also to the anonymous referees for their useful comments and suggestions.

References

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proc. Symposium on Lisp and Functional Programming*, pages 86–95. ACM, July 1988.

[Cli91] William Clinger. Hygienic macros through explicit renaming. *LISP Pointers*, 4(4):25–28, December 1991.

[CR90] Pavel Curtis and James Rauen. A module system for Scheme. In *Proc. Symposium on Lisp and Functional Programming*, pages 13–19. ACM, July 1990.

[CR91] William Clinger and Jonathan Rees. Macros that work. In *Proc. Symposium on Principles of Programming Languages*, pages 155–162. ACM, January 1991.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1992.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[KFFD86] Eugene M. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proc. Symposium on Lisp and Functional Programming*, pages 151–161. ACM, August 1986.

[Mac84] David B. MacQueen. Modules for standard ML. In *Proc. Symposium on Lisp and Functional Programming*, pages 198–207. ACM, August 1984.

[Pit80] Kent M. Pitman. Special forms in LISP. In *Proc. Symposium on Lisp and Functional Programming*, pages 179–187. ACM, August 1980.

[Ree89] Jonathan Rees. Modular macros. Master's thesis, MIT, May 1989. Dept. of Electrical Engineering and Computer Science.

[Ree93] Jonathan Rees. Another module system for Scheme. unpublished manuscript, January 1993.

[Ste90] Guy L. Steele Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.

[WD99] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proc. Symposium on Principles of Programming Languages*, pages 203–213. ACM, January 1999.