Sharing Code through First-class Environments

Christian Queinnec* & David De Roure†
École Polytechnique Department of Electronics and Computer Science,
& INRIA-Rocquencourt University of Southampton

Abstract

Nowadays the Net is one of the most obvious driving forces. Yet, to consider it as one global store through which values and code may be shared is still immature. This paper suggests first-class environments as a means to achieve that goal in a multi-user Scheme framework. We propose two new special forms with a simple semantics. Our model allows precise control over environments (including extensible environments) and does not require (but does not prevent) reflective operations.

1 Motivation

Millions of users now dream of means through which they may share programs and/or data. Data are becoming more and more complex; they are no longer made of simple, atomic, flat values such as numbers or characters but now mundanely incorporate pointers, as exhibited by the increasing number of WWW pages full of references towards remote pieces of information. Other data are only acquired after long computations and are better shared rather than recomputed. Of course, data may be stored in shared files but then need to be parsed to resurrect programmatically. All these reasons favor the invention of a common shared distributed memory that offers a language independent API (Application Programming Interface) allowing numerous users to devise structures or classes, to instantiate them and to share or migrate the resulting instances so they can be directly computed upon.

On the other hand, programs are mainly shared through `tar.gz` files (provided they can be recompiled), exchanged via strings as in Tcl [Ous93] or compiled into bytecode as in Java [Sun95]. Basically when a program text is interactively submitted for evaluation, it is converted into some code (presumably accompanied by some literals) and some free variables. These free variables have to be bound to some locations before this program may be run. Reciprocally, a program is implicitly parameterized with respect to its free variables. Therefore, to be able to share code, we must specify in which environments free variables must be found; this may be achieved through the existence of first-class environments as in [BL84, RAM84, FF86, GJL87, MR91, Jag94, LF93]. A code to be shared may then be represented by a *module*, a function that expects an environment providing the (locations of the) free variables that the program needs for its evaluation.

The essence of our proposal is to offer two new special forms to handle first-class environments i.e. collections of named locations. The first one, `export`, allows us to reify a lexical environment (or part of it) into a first-class value. The second special form, `import`, evaluates some forms in an environment which is a mixture of an explicitly stated first-class environment composed with the current lexical environment. Our proposal still provides efficient compilation as well as some additional functions to manage first-class environments. First-class environment is the last component of the computation state that was not reifiable in regular Scheme. Our aim is to show that we can *(i)* overcome this limitation without degrading performance, *(ii)* ease code sharing by a precise control over the environments within which code is installed. Programs are parameterized with respect to their free variables and as such are much more reusable than closures that may only be customized as far as their arguments allow customization.

A closure gets values in a combination; an importation gets locations from a first-class environment. Making some resources available is better achieved through a first-class environment than through a list, a record or even a closure since these values impose some order, some accessing method and are too overspecified. A first-class environment can only be imported and allows reference to the captured variables with the least possible syntax: that of variable reference.

Importation allows us to parameterize [Lam88] a program with respect to an unordered set of variables. Different programs may be parameterized with respect to intersecting sets of variables, thus achieving a sort of fragmented `letrec` between them if they are imported into the same environment; different programs may thus share mutable locations. Conversely, one program may be imported into different environments by different users, or into different environments by a single user to achieve fault-tolerance by replication of

*Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France – Email: Christian.Queinnec@polytechnique.fr This work has been partially funded by GDR-PRC de Programmation du CNRS and EC project reference ERB 4050 PL 930186.

†Southampton SO17 1BJ, UK – Email: dder@soton.ac.uk This work has been partially funded by EPSRC GR/K36409 and EC project reference ERB 4050 PL 930186.

computations: importation features a sort of dynamic reentrant linking facility.

Our first-class environments *(i)* obey the *quasi-static discipline* meaning that variables are, on importation side, statically known to be either static or quasi-static as in [LF93] and, on exportation side, local variables are statically known to be either captured or non captured in a first-class environment, *(ii)* do not require reflection (but do not prevent it), *(iii)* are versatile enough to promote new useful programming techniques (modules, objects, etc.), *(iv)* may be exercised to explain the behavior of toplevel environments.

Our proposal roughly stands as a new point among the pioneering works of [BL84, RAM84, FF86, GJL87, Lam88, CR90, MR91, Jag94, Tun92, LF93, DPS94] on module, exportation and importation systems. We improve on [LF93] since we are able to manage extensible environments and implicit sets of free variables with a much simpler semantics, while we offer some reflective features of [Jag94] without impeding compilation efficiency. Detailed comparison is deferred until section 7.

Since the semantics of these new special forms is fairly straightforward, we dare start by immediately introducing their denotational semantics in section 2. We then present, in section 3, a series of examples illustrating some of their usage. Section 4 introduces auxiliary functions to enhance environment management as well as new examples of use. A naïve implementation is sketched in section 5. Section 6 comments on reflective aspects while comparison with related works appears in section 7.

2 Formal Semantics

This section presents the denotation of two new special forms named `export` and `import`. Table 1 contains the main fragment of a denotational semantics for Scheme. The notations are inspired from [CR91]. Given a memory state (a store) and a number of locations, the *allocate* function invokes its third argument on the resulting store and the freshly allocated locations. The major difference from the semantics of [CR91] is the structure of the **Env** domain. Instead of mapping an identifier directly into a location, an environment ρ rather has signature:

$$\mathbf{Id} \times \mathbf{Store} \times (\mathbf{Store} \times \mathbf{Loc} \rightarrow \mathbf{Answer}) \rightarrow \mathbf{Answer}$$

In other words, given an identifier and a store, an environment invokes its third argument with the location associated to the identifier and the resulting store. This signature is not new [FW84] but will ease our later introduction of extensible environments. This signature is illustrated by the denotations of reference and assignment in table 1.

Conventions for environment handling are inspired by [LF93]; they appear in table 2. The null environment is named $\rho_\perp$. Environments may be restricted: $\rho\backslash_\nu$ returns a new environment that is similar to ρ except that it only contains the identifier ν. This restriction is naturally extended to a set of identifiers using notation $\rho\backslash_{\{\nu^*\}}$. Environments may be chained: $\rho \oplus \rho'$ returns a new environment that behaves as ρ but for the identifiers not contained in ρ that are then looked for in ρ'. We overload the usual extension notation, $\rho[\nu \rightarrow \alpha]$, to cope with the new signature of environments. We also extend it to sets of identifiers and locations as in $\rho[\nu^* \stackrel{*}{\rightarrow} \alpha^*]$. Finally it is possible to actively check, with $\rho \ni \{\nu^*\}\sigma$, whether an environment really associates locations to a set of identifiers in a given store.

The denotations of `export` and `import` each have two cases depending on whether or not the set of given identifiers is empty. Since to export or import an empty set of variables is pointless, these special cases will mean export or import *all possible* variables where "all possible" is explained below. Table 3 contains these denotations with $\mathcal{FV}(\pi^+)$ standing for the free variables of the π^+ forms. These denotations may be paraphrased as follows:

`(export)`: yields the entire current lexical environment.

`(export` ν^+`)`: yields the current lexical environment restricted to the identifiers given. Due to their presence in the `export` form, these identifiers appear free and therefore are necessarily contained in the current lexical environment i.e. have an associated location. The initial store, σ_0 in table 1, shows that locations may be uninitialized provided they are not read.

`(import ()` π π^+`)`: evaluates π to obtain an environment, checks that the free variables of π^+ are contained in this environment then evaluates π^+ in this sole environment. The current lexical environment is therefore totally shadowed by the new environment during the evaluation of π^+. As for implicit `begin` forms, the final value of the `import` form is the last value of π^+.

`(import (`ν^+`)` π π^+`)`: evaluates π to obtain an environment, checks that the variables ν^+ are defined in this environment then evaluates π^+ in this very environment restricted to ν^+ and chained to the current lexical environment. The ν^+ identifiers of the new environment shadow their homonyms in the current lexical environment. The final value of the `import` form is the last value of π^+.

Observe that a free variable of π^+ is either explicitly mentioned by the `import` form (and must be provided by the obtained environment—this is checked only once at importation time, that is linking time) or, must be present in the lexical environment of the `import` form. Reciprocally, a local variable is either exported or not. This first case occurs when an `export` form mentions the local variable (implicitly or explicitly) from within its binding scope. This categorization of free and local variables is statically decidable: we name it the *quasi-static discipline* in honor of [LF93] who introduced the term "quasi-static".

Note that `(export` ν^+`)` and `(import (`ν^+`) ...)` restrict the first-class environment to only contain the ν^+ variables. This is also the case of `(import () ...)` which restricts the imported environment to all the possible variables i.e. the free variables of its body. On the other hand, `(export)` lets the caller grab the entire lexical environment with all the variables it contains or potentially contains if extensible; see section 4 below.

3 Examples

This section presents some examples involving importation and exportation. This first example creates then exports locations containing values. It shows that it is possible to alter the exported environment from the outside with values computed as if evaluated inside. This facility is important

$\nu \in$ **Id**
$\alpha \in$ **Loc** $=$ **Address** $+ \{\textit{non-existent-variable}\}$
$\rho \in$ **Env** $=$ **Id** $\times$ **Store** $\times$ (**Store** $\times$ **Loc** $\rightarrow$ **Answer**) $\rightarrow$ **Answer**
$\kappa \in$ **Cont** $=$ **Value** $\times$ **Store** $\rightarrow$ **Answer**
$\sigma \in$ **Store** $=$ **Loc** $\rightarrow$ **Value**
$\varepsilon \in$ **Value** $=$ **Fun** $+$ **Env** $+$ **Id** $+ \ldots$
$\varphi \in$ **Fun** $=$ **Value*** $\times$ **Cont** $\times$ **Store** $\rightarrow$ **Answer**

$$\mathcal{E}[\![\nu]\!]\ \rho\ \kappa\ \sigma = (\rho\ \nu\ \sigma\ \lambda\sigma'\alpha.(\kappa\ (\sigma'\ \alpha)\ \sigma'))$$

$$\mathcal{E}[\![\texttt{(set!}\ \ \nu\ \pi\texttt{)}]\!]\ \rho\ \kappa\ \sigma = \mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma'.(\rho\ \nu\ \sigma'\ \lambda\sigma''\alpha.(\kappa\ \varepsilon\ \sigma''[\alpha \rightarrow \varepsilon]))\ \sigma$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{(lambda}\ \texttt{(}\nu^*\texttt{)}\ \ \pi^+\texttt{)}]\!]\ \rho\ \kappa\ \sigma = \\ &(\kappa\ \mathit{in}\mathbf{Value}(\lambda\varepsilon^*\kappa'\sigma'.\ \ \mathbf{if}\ \#\varepsilon^* = \#\nu^* \\ &\qquad\qquad \mathbf{then}\ \mathit{allocate}\ \sigma'\ \#\nu^*\ \lambda\sigma''\alpha^*.\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \kappa'\ \sigma''[\alpha^* \xrightarrow{*} \varepsilon^*] \\ &\qquad\qquad \mathbf{else}\ \mathit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )\ \sigma)\end{aligned}$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\ \rho\ \kappa\ \sigma = \mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma'.\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma''.(\varphi\ |_{\mathbf{Fun}}\ \varepsilon^*\ \kappa\ \sigma'')\ \sigma'\ \sigma$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\ \rho\ \kappa\ \sigma = \mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\ \rho\ \kappa\ \sigma = \mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma'.\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma'\ \sigma$$

$$\mathcal{E}^+[\![\pi]\!]\ \rho\ \kappa\ \sigma = \mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\ \rho\ \kappa\ \sigma = \mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma'.\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma''.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma'')\ \sigma'\ \sigma$$

$$\mathcal{E}^*[\![\ ]\!]\ \rho\ \kappa\ \sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\sigma_0 = \lambda\alpha.\mathit{wrong}\ \text{“}\texttt{Uninitialized address}\text{”}$$

Table 1. Denotational semantics of a subset of Scheme

$$\rho_\perp = \lambda\nu\sigma q.(q\ \sigma\ \textit{non-existent-variable})$$

$$\rho\backslash_\nu = \lambda\nu'\sigma q.\ \mathbf{if}\ \nu = \nu'\ \mathbf{then}\ (\rho\ \nu\ \sigma\ q)\ \mathbf{else}\ (q\ \sigma\ \textit{non-existent-variable})\ \mathbf{endif}$$

$$\rho\backslash_{\{\nu\ \nu^*\}} = (\rho\backslash_\nu) \oplus (\rho\backslash_{\{\nu^*\}})$$

$$\rho\backslash_{\{\}} = \rho_\perp$$

$$(\rho) \oplus (\rho') = \lambda\nu\sigma q.(\rho\ \nu\ \sigma\ \lambda\sigma'\alpha.\ \mathbf{if}\ \alpha = \textit{non-existent-variable}\ \mathbf{then}\ (\rho'\ \nu\ \sigma'\ q)\ \mathbf{else}\ (q\ \sigma'\ \alpha)\ \mathbf{endif}\ )$$

$$\rho[\nu \rightarrow \alpha] = \lambda\nu'\sigma q.\ \mathbf{if}\ \nu' = \nu\ \mathbf{then}\ (q\ \sigma\ \alpha)\ \mathbf{else}\ (\rho\ \nu'\ \sigma\ q)\ \mathbf{endif}$$

$$\rho[\nu^* \xrightarrow{*} \alpha^*] = \ \mathbf{if}\ \#\nu^* > 0\ \mathbf{then}\ \rho[\nu^* \dagger 1 \xrightarrow{*} \alpha^* \dagger 1][\nu^* \downarrow_1 \rightarrow \alpha^* \downarrow_1]\ \mathbf{else}\ \rho\ \mathbf{endif}$$

$$(\rho) \ni \{\}\sigma q = (q\ \sigma)$$

$$(\rho) \ni \{\nu\ \nu^*\}\sigma q = (\rho\ \nu\ \sigma\ \lambda\sigma'\alpha.\ \mathbf{if}\ \alpha = \textit{non-existent-variable}\ \mathbf{then}\ \mathit{wrong}\ \text{“}\texttt{Missing variable}\text{”}\ \mathbf{else}\ (\rho) \ni \{\nu^*\}\sigma' q\ \mathbf{endif}\ )$$

Table 2. Auxiliary functions for environment

$$\mathcal{E}[\![\texttt{(export)}]\!] = \lambda\rho\kappa\sigma.(\kappa\ in\mathbf{Value}(\rho)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(export}\ \nu^{+}\texttt{)}]\!] = \lambda\rho\kappa\sigma.(\kappa\ in\mathbf{Value}(\rho\backslash_{\{\nu^{+}\}})\ \sigma)$$

$$\mathcal{E}[\![\texttt{(import ()}\ \ \pi\ \pi^{+}\texttt{)}]\!] = \lambda\rho\kappa\sigma.\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\rho'\sigma'.(\rho'\ |_{\mathbf{Env}}) \ni \{\mathcal{FV}[\![\pi^{+}]\!]\}\sigma'\lambda\sigma''.\mathcal{E}^{+}[\![\pi^{+}]\!]\ \rho'\ |_{\mathbf{Env}}\ \ \kappa\ \sigma''\ \sigma$$

$$\mathcal{E}[\![\texttt{(import (}\nu^{+}\texttt{)}\ \ \pi\ \pi^{+}\texttt{)}]\!] =$$
$$\lambda\rho\kappa\sigma.\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\rho'\sigma'.(\rho'\ |_{\mathbf{Env}}) \ni \{\nu^{+}\}\sigma'\lambda\sigma''.\mathcal{E}^{+}[\![\pi^{+}]\!]\ (\rho'\ |_{\mathbf{Env}}\ \backslash_{\{\nu^{+}\}}) \oplus (\rho)\ \kappa\ \sigma''\ \sigma$$

Table 3. Denotation of export and import

for debugging, profiling, instrumenting, or upgrading programs as well as for autoload functions. An `export` form is a mechanism that extends the scope of the exported variables. Reciprocally, `import` forms provide the regions where this scope do extend.

```
(define (create-stack . content)
  (let ((counter -1))         ;introduce local variable
    (define (push x)          ;define push
      (set! counter (+ counter 1))
      (set! content (cons x content)) )
    (define (pop)             ;define pop
      (if (pair? content)
          (let ((top (car content)))
            (set! content (cdr content))
            top ) ) )
    (push (length content))   ;perform some computation
    ;;export three local bindings and one global
    (export counter + push pop) ) )
```

```
(let ((stk-env (create-stack 11 22 33 44 55))
      (+       *) )
  ;;import only push, pop and counter
  (import (push pop counter) stk-env
     (set! pop (let ((old-pop pop))      ;modify pop
                 (lambda ()
                   (import (+) stk-env ;import +
                      (set! counter (+ counter 1))
                      (old-pop) ) ) ))
     (push (+ (pop) (pop))) )            ;three operations
  (import (counter) stk-env              ;import counter
     counter ) )                   → 3
```

The first program defines a generator of stack utilities to be gathered in a first-class environment. The second program creates such a first-class environment then imports it for some computations (redefining the `pop` location to count the number of times it is invoked) and, finally, reads the `counter` variable. Observe that these importations do not shadow the local variable `stk-env` nor the global variable `*`. Observe also that the redefinition of `pop` is possible since the `stk-env` environment captures the location of `pop` and not only its value. Moreover the new value of `pop` is evaluated after importing the same addition function that the original `push` was using, and of course the same `counter` variable.

3.1 Modules

A reverse view would be to fill externally provided locations with values. In the first case, the definer of the program builds a ready-to-use environment while in the second case the definer only provides a function, a *module*, that installs a set of values into locations provided by the client of the module. Here is an example of that second technique:

```
(define (install-stack-module env)
  (import (push pop counter) env ;acquire locations
     (let ((content '()))
       (set! push (lambda (x)     ;install push
                    (set! counter (+ counter 1))
                    (set! content (cons x content)) ))
       (set! pop (lambda ()       ;install pop
                   (if (pair? content)
                       (let ((top (car content)))
                         (set! content (cdr content))
                         top ) ) )) ) ) )

(define counter 0)
(define (get-pusher)
  (let ((push 'wait)(pop 'wait)) ;provide locations
    (install-stack-module (export counter push pop))
    push ) )
(define synchronous-push
  (let ((pushers (list (get-pusher) (get-pusher))))
    (lambda (x)
      (for-each (lambda (push) (push x))
                pushers ) ) ) )
```

In this example, the `get-pusher` function installs the `push` function twice, into two distinct but similar environments. The `synchronous-push` function just pushes its argument onto two different stacks. Note that `counter` is shared by these two installations so both `push` functions increment the same `counter`. This is an example where mapping two a priori different variables onto the same location offers a customization that was not explicitly apparent in the `install-stack-module` code and difficult to implement if only values were exported.

Our view of "module" should not be confused with the "unit of compilation" concept (as provided by compilers such as Bigloo) where the goal is efficiency rather than reusability.

3.2 Objects

The relationship between objects and first-class environments has been studied for long and basically the examples we could provide with our special forms would be in spirit similar to those of [LF93]. Nevertheless we do not think this is a good idea since one of the critical point for method lookup is the ability to test efficiently the class of an object. The `class-of` function that returns the class of an object would be something like:

```
(define (class-of o)
  (import (class) o class) )
```

The linking cost of that `import` form is much higher than the execution cost of its body because the location of `class` is not necessarily in a fixed position in the first-class environment o and must be looked for dynamically.

3.3 Enquiring closures

Some Scheme implementations provide a function, `procedure->environment`, that given a closure returns the environment it closed. This is a dangerous feature since it has to be precisely defined and may break previous optimizations. For instance in the following example, does `(procedure->environment foo)` contains `y` and does the addition disappear since `x` is zero ?

```
(define foo
  (let ((x 0)(y 'foo))
    (lambda (z) (+ x z)) ) )
(import (x y) (procedure->environment foo) ;import y !?
   (set! x y) )  ;change x to a non-number !?
```

To open up a closure should not break the compilation of the associated function, just as the exportation of a location must not break the exporting code. It is therefore necessary to know statically which variables may be exported, so the compiler can be cautious with these variables. We therefore introduce a new syntax to define a function and the environment it may export. Let's call this syntax `enquirable-lambda` and suppose the existence of a couple of functions (using *ad-hoc* primitives, hash-tables, whatever) associating/retrieving a value to/from a closure. Of course, to store this value in the closure object itself would be the best implementation.

```
(define-syntax enquirable-lambda
  (syntax-rules ()
    ((enquirable-lambda exporting formals body ...)
     (let ((proc (lambda formals body ...))
           (env  (export . exporting)) )
       (set-exported-env! proc env)
       proc ) ) ) )
(define (procedure->environment proc)
  (get-exported-env proc) )
```

A generic function, as in CLOS [BDG+88], may look for a method in a dispatch table, held in some closed variable, according to the class of a receiver. This dispatch table has also to be made available to the general outer `add-method!` function to be enriched with a new method. A good way to extract a dispatch table from a generic function is to use our previous `enquirable-lambda` facility:

```
(define-syntax define-generic
  (syntax-rules ()
    ((define-generic (selector receiver arguments ...))
     (define selector
       (let ((dispatch-table (initial-dispatch-table)))
         (enquirable-lambda (dispatch-table) ;export
                            (receiver arguments ...)
           ((lookup-method dispatch-table
                           (class-of receiver) )
            receiver arguments ... ) ) ) ) ) ) ) )
(define (add-method! generic class method)
  (import (dispatch-table)                        ;import
          (procedure->environment generic)
     (augment-dispatch-table!
      dispatch-table class method ) )
  generic )
```

3.4 Dynamic evaluation

The `import` form may be viewed as a kind of "static" `eval` facility where it is possible to evaluate some static forms π^+ in a dynamically chosen environment. The difference from a full-fledged `eval` is that here the forms to evaluate are known statically and all the bindings they need must exist beforehand: there is no possible incremental definition i.e. dynamic creation of new bindings.

The `import` form corresponds to the many cases where `eval` is invoked on a `backquote` form as suggested by the following approximate relation (paying attention to the possible lexical capture of variable `x`):

`(eval `(... ,x ...) env)` ≈ `(import (x) env (... x ...))`

Toplevel facilities that acquire dynamically the forms to evaluate (i.e. performing something like `(eval (read))`) cannot be helped by our `import` form. However, our `import` form do correspond to the cases where some code has to be evaluated in some environment and it is not possible to write this code in the scope of that environment. This impossibility may occur for many reasons; for example, the size of the code may make it unmanageable in a single file, or some deferred user code (hooks) may be allowed to customize a predefined library. Even if a direct programmation would not have used `eval` but objects instead, here follows a realistic example where a byte-code machine may be dynamically instrumented from the outside.

```
(define (build-machine)
  (define bytes #(255))
  (define pc    0)
  (define val   'wait)
  (define env   '())
  (define (wrong) "no such opcode")
  (let ((opcodes (make-vector 256 wrong)))
    (define (run)
      (let* ((byte (vector-ref bytes pc))
             (op   (vector-ref opcodes byte)) )
        (set! pc (+ pc 1))
        (op) ) )
    (define machine-env
      (export bytes pc val env opcodes
              wrong run machine-env ) )
    (vector-set! opcodes 0 run) ;nop
    machine-env ) )
(define machine0 (build-machine))

(import (opcodes bytes pc val run) machine0
  (vector-set! opcodes 1 ;load-quick
    (lambda ()
      (set! val (vector-ref bytes pc))
      (set! pc (+ pc 1))
      (run) ) )
  (vector-set! opcodes 2 ;exit
    (lambda () val) )
  (set! bytes #(0 1 23 2))
  (run) )                    → 23
```

The load-quick bytecode instruction is synthesized out of the definition of the `build-machine` abstraction but as if it were defined in. The load-quick instruction just loads the `val` register with the next bytecode.

While `import` allows to evaluate some code as if contained in some environment, `import` is free of leakage. The following example leads to an error since it is not possible to extract `build-machine` from `machine0`:

```
(import () machine0
  (export build-machine) )
```

4 Library

The `export` special form is currently the only way to create first-class environments. It is a simple step to enrich the model with regular side-effect-free functions that create or manage first-class environments. These functions give the user access to the auxiliary functions of the denotation: one may create an empty (or arid [LF93]) environment with `create-empty-environment`; see table 4.

The major innovation is the introduction of extensible environments. An extensible environment is an immutable first-class value that contains all possible variables. The trick is whenever an identifier is looked for in an extensible environment that does not contain it yet, a new location is associated to it and recorded. Therefore no identifier can be found not to belong to an extensible environment! This is this feature that imposed the signature of **Env**. Extensible environments are created by $\Omega^{\top}$; see again table 4. An extensible environment uses a location as a place holder for the actual environment it stands for and this location is initialized with $\rho_{\perp}$.

4.1 `letrec` visited

Extensible environments stand for environments in which all possible variables are bound to locations [CR91, § 5.2.1]. To say that variables are bound does not mean that they are initialized: they are not, and it is an error to ask for the value of an uninitialized variable. The `create-complete-environment` function creates a new extensible environment where all possible variables are uninitialized. It is therefore possible to define `letrec` as a regular syntax without resorting to the existence of the necessarily non existent <undefined> value as done in [CR91, § 7.3]. We neglect here the order of evaluation to simplify the expansion:

```
(define-syntax letrec
  (syntax-rules ()
   ((letrec ((var val) ...) body ...)
    (import (var ...) (create-complete-environment)
       (set! var val) ...
       body ... ) ) ) ) )
```

The local variables of the `letrec` form are looked for in the extensible environment created, and their associated locations are of course found since this is an extensible environment. The initializing assignments are evaluated in that environment followed by the body of the `letrec` form. If some variable is referenced before being assigned then an error will be raised since the variable was created uninitialized in the extensible environment.

Although extensible, a smart compiler may notice that the locally created environment is only used for a fixed number of variables that are not reexported (unless the body contains such an `export` form). Therefore the environment does not need to be actually extensible. On the other hand the compiler has to be cautious when referencing an imported variable and check whether it is initialized or not. It is possible to perform an approximate analysis [Que94, p. 389] to discover some surely initialized variables and avoid checking them.

This new definition of `letrec` is intended to be an explanation not a actual implementation. Compilers should still consider `letrec` as a primitive special form.

The technique used to illustrate a correct expansion for `letrec` i.e. introducing new uninitialized locations out of an extensible environment may be turned into a syntax, say `uninitialized-let`, but may also be used to shadow i.e. exclude some locations from an environment as in:

```
(import ( excluded variables )
        (create-complete-environment)
  (export) )
```

4.2 Linking environments

The `chain-environment` function allows us to use the $\oplus$ operation and thus to compose environments piece by piece. To chain an extensible environment with anything is meaningless but the reverse is really useful. Suppose that the initial environment contains all the predefined variables, then this environment may be reified with an `(export)` form and stored in the `scheme-env` variable. Chaining `scheme-env` with an extensible environment allows multiple users to share a common environment with their own additional bindings gathered in the second part of this environment. See section 6.3 for a different solution where users' bindings are recorded before the initial environment and may thus shadow initial bindings. A function to create such toplevel environments may be defined as:

```
  ...
(set! scheme-env (export))
  ...
(define (create-user-toplevel-environment)
  (chain-environment scheme-env
                     (create-complete-environment) ) )
```

Our proposed special forms do not provide control of the mutability of bindings, therefore the previous solution does not prevent `scheme-env` from being mutated. This point will be addressed later in section 6.4.

Chaining does not mean that the environments involved are really chained by the implementation; see section 5.

A common problem of first-class environment lies with incremental definition [RAM84, MR91, Tun92] which requires adapting the resolution of free variables to the evolution of first-class environments. More precisely, the problem arises when the current lexical environment is $(\rho) \oplus (\rho')$. If ρ is an environment not containing a variable `X` then the location associated to `X` must be provided by ρ'. Now, if ρ is mutated to contain `X` then this new definition dynamically shadows that of ρ'. Such a change is against quasi-static discipline and is not possible in our model since our environments are immutable. Even if ρ is an extensible environment, it totally shadows ρ' and it is pointless to chain it in front of any other environment. The magic also comes from `import` which is a special form and not a function as `eval`; see section 3.4.

We do not think that this position is against interactive modular programming [Tun92] as one can recreate a new environment excluding some locations, but rather suggest that it avoids subtle bugs as names clashes should not be automatically ignored and it avoids developing sophisticated implementation techniques to ensure efficiency.

To export the entire environment is dangerous: it degrades efficiency since all locations may potentially be mutated and therefore prevents optimizations. It is nevertheless mandatory if one wants to reify the entire state of computation. Given the semantical gap between `(export` ν`)` and `(export)`, the latter might be renamed `(the-environment)` to prevent hazardous typos.

5 Naïve implementation

Fundamentally a first-class environment is nothing more than an unordered set of second-class named locations and can be implemented as a table (possibly accessed with some

$$
\begin{array}{l}
[\![\texttt{create-empty-environment}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 0 \\
\qquad\qquad\qquad \mathbf{then}\ (\kappa\ in\mathbf{Value}(\rho_\perp)\ \sigma) \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{chain-environment}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 2 \\
\qquad\qquad\qquad \mathbf{then}\ (\kappa\ in\mathbf{Value}((\varepsilon^* \downarrow_1|_{\mathbf{Env}}) \oplus (\varepsilon^* \downarrow_2|_{\mathbf{Env}}))\ \sigma) \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

$$
\begin{array}{l}
(\Omega^\top \alpha) = \\
\lambda \nu \sigma q.\ \ \mathbf{let}\ \rho = (\sigma\ \alpha)\ |_{\mathbf{Env}} \\
\qquad \mathbf{in}\ (\rho\ \nu\ \sigma\ \lambda \sigma' \alpha'.\ \ \mathbf{if}\ \alpha' = non\text{-}existent\text{-}variable \\
\qquad\qquad\qquad\qquad \mathbf{then}\ allocate\ \sigma'\ 1\ \lambda \sigma'' \alpha^*.\ \ \mathbf{let}\ \alpha'' = \alpha^* \downarrow_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (q\ \sigma''[\alpha \rightarrow in\mathbf{Value}(\rho[\nu \rightarrow \alpha''])]\ \alpha'') \\
\qquad\qquad\qquad\qquad \mathbf{else}\ (q\ \sigma'\ \alpha')\ \mathbf{endif}\ \ )
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{create-complete-environment}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 0 \\
\qquad\qquad\qquad \mathbf{then}\ allocate\ \sigma\ 1\ \lambda \sigma' \alpha^*.(\kappa\ in\mathbf{Value}((\Omega^\top \alpha^* \downarrow_1))\ \sigma'[\alpha^* \downarrow_1 \rightarrow in\mathbf{Value}(\rho_\perp)]) \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

Table 4. Environment related functions

$$
\begin{array}{l}
[\![\texttt{environment-get}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 2 \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{let}\ \rho = \varepsilon^* \downarrow_1|_{\mathbf{Env}} \\
\qquad\qquad\qquad\qquad \mathbf{and}\ \nu = \varepsilon^* \downarrow_2|_{\mathbf{Id}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ (\rho\ \nu\ \sigma\ \lambda \sigma' \alpha.\ \ \mathbf{if}\ \alpha = non\text{-}existent\text{-}variable \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ wrong\ \text{“}\texttt{Non existent variable}\text{”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\kappa\ (\sigma'\ \alpha)\ \sigma')\ \mathbf{endif}\ \ ) \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{environment-present?}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 2 \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{let}\ \rho = \varepsilon^* \downarrow_1|_{\mathbf{Env}} \\
\qquad\qquad\qquad\qquad \mathbf{and}\ \nu = \varepsilon^* \downarrow_2|_{\mathbf{Id}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ (\rho\ \nu\ \sigma\ \lambda \sigma' \alpha.(\kappa\ in\mathbf{Value}(\alpha \neq non\text{-}existent\text{-}variable)\ \sigma')) \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{environment-rename}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 3 \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{let}\ \rho = \varepsilon^* \downarrow_1|_{\mathbf{Env}} \\
\qquad\qquad\qquad\qquad \mathbf{and}\ \nu = \varepsilon^* \downarrow_2|_{\mathbf{Id}} \\
\qquad\qquad\qquad\qquad \mathbf{and}\ \nu' = \varepsilon^* \downarrow_3|_{\mathbf{Id}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ (\rho\ \nu\ \sigma\ \lambda \sigma' \alpha.\ \ \mathbf{if}\ \alpha = non\text{-}existent\text{-}variable \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ wrong\ \text{“}\texttt{Non existent variable}\text{”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\kappa\ in\mathbf{Value}(\rho[\nu \rightarrow non\text{-}existent\text{-}variable][\nu' \rightarrow \alpha])\ \sigma')\ \mathbf{endif}\ \ ) \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{environment-enrich}]\!] \leadsto \\
in\mathbf{Value}(\lambda \varepsilon^* \kappa \sigma.\ \ \mathbf{if}\ \#\varepsilon^* = 2 \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathbf{let}\ \rho = \varepsilon^* \downarrow_1|_{\mathbf{Env}} \\
\qquad\qquad\qquad\qquad \mathbf{and}\ \nu = \varepsilon^* \downarrow_2|_{\mathbf{Id}} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ allocate\ \sigma\ 1\ \lambda \sigma' \alpha^*.(\kappa\ in\mathbf{Value}(\rho[\nu \rightarrow \alpha^* \downarrow_1])\ \sigma') \\
\qquad\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”}\ \mathbf{endif}\ \ )
\end{array}
$$

Table 5. Environment related reflective functions

hash coding if its size or extensibility justifies this) mapping names to their associated locations; see figure 1.

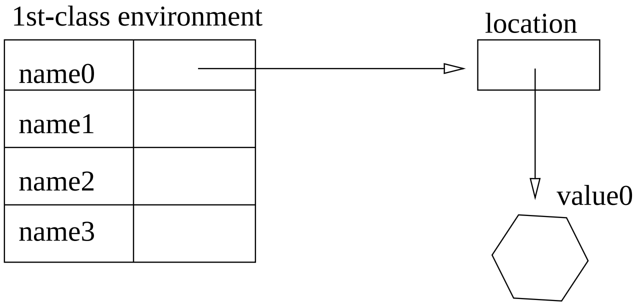

Figure 1. First-class environment

Let's suppose that the lexical environment is represented by linked lexical contours i.e. sequences of locations. Another technique to implement first-class environment [Que94, p. 269], improving multiple reification of similar environments (see figure 2), is to pair the current lexical environment with a (static) table mapping names to their associated lexical indexes (i, j) where j is an offset in the i^{th} contour. The cost of reification is small but the linking time is slowed down and too much of the current environment is captured.

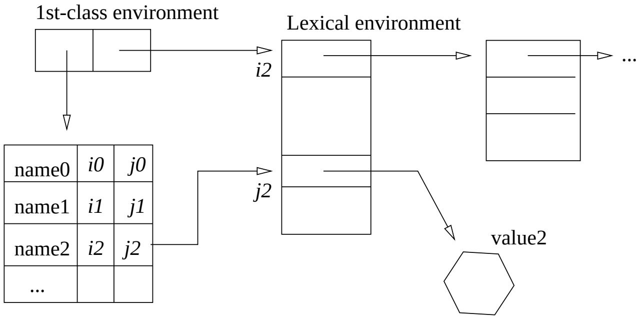

Figure 2. Variant of first-class environment

When importing a first-class environment, there is always a finite number of statically known variables to resolve in that environment, that is the given variables of the `import` form, or the free variables of the body of the `import` form. We now suppose that these variables are gathered in an import contour map.

Before evaluating the body of `import`, a fresh contour is allocated and for each variable of the contour map, its location is dynamically looked for in the first-class environment and copied, according to the contour map, in the fresh contour; see figure 3. Access to these variables passes through this new contour. For instance, a reference to the imported `name0` variable (stored at position i in the import contour map) will be compiled by fetching the i^{th} content of the fresh import contour, dereferencing it into the location holding the value. Note that the body of the `import` form does not need any longer the names of the imported variables, their rank in the import contour map is sufficient. However the names are required at link-time to fill the fresh contour.

The additional cost of accessing such an imported variable is just one indirection and a check that it is not uninitialized. This mechanism is reminiscent of the closure creation of the FAM [Car84]. Any missing variable is immediately reported, so no other test is needed when evaluating the body of the `import` form. The real cost of an `import` form is concentrated in the linking cost, that is the research of the required named locations when building the contour for the `import` form.

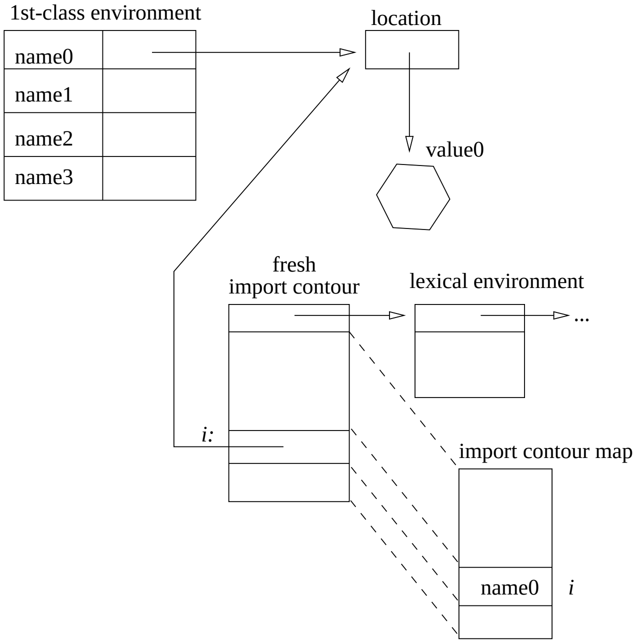

Figure 3. Lexical environment at `import` time

As mentioned earlier, some improvements are possible when an extensible environment is explicitly imported: this is quite similar to the optimization where a combination has an explicit `lambda` form as operator.

6 Reflective aspects

It is clear from the information gathered in a first-class environment that, given the name of a variable, its associated value may be found dynamically and/or mutated (if allowed; see section 6.4 below) in this environment. It is also easy to determine whether a name is present or not in a first-class environment. The `environment-present?` predicate (see table 5) corresponds to a stable property: its answer cannot vary. To check whether a binding is initialized is a different predicate whose answer may evolve from false to true.

If the language provides an object system, a simple way to offer reflection over first-class environments is to make them regular objects. Otherwise, specific functions may be offered. Observe that, conversely to other proposals for first-class environment [RAM84, MR91], the presence of these additional functions is completely independent of the importation mechanism and does not impact it.

6.1 Renaming

An interesting effect might be achieved if one is allowed to create a modified copy of a first-class environment: it is then possible to change the name under which a location is exported, thus providing a renaming facility for exported environments; see function `environment-rename` in table 5. Renaming is not universally recognized as a good tool [DPS94] but we think in a more and more persistent, multi-user world, name clashes are very likely to happen.

This renaming facility may accompany α-conversion to still produce a first-class environment defining the same set of names. The renaming work is done at run-time and does not impact the compilation of the `import` form since the compiled code only depends on the presence of some locations in some well defined place (the import contour) and is independent of the names that were used to find these very locations.

6.2 Extension

Another interesting effect is to enrich a first-class environment with a given name. The `enrich-environment` function may be given a name ν and an environment ρ and returns a new first-class environment, the equivalent of:

```
(import (ν) (create-complete-environment)
   (chain-environment (export ν) ρ) )
```

In the resulting environment, the name is associated with an uninitialized location. Note that the environment which is extended, ρ, is not mutated even if extensible.

6.3 Toplevels

All these functions allow us to precisely and reflectively enquire and control first-class environments without impeding the semantics and the efficiency of the exportation/importation mechanism. They allow us to make completely explicit the work of a toplevel loop (as in [GJL87]) that reads a program π, extracts its free variables $\mathcal{FV}(\pi)$ while analyzing (expanding, compiling) it, turns the program into a module corresponding to `(lambda (r) (import (`$\mathcal{FV}(\pi)$`) r `π`))`, selects one environment in which to evaluate it, enriches that environment with the free variables (or checks that they are already present) and, finally, evaluates the above entity. Depending on the chosen environment, one may only access a restricted set of variables, share a common environment with others or even create one's own private extensible environment as shown in the following example:

```
(define (create-user-toplevel-loop read initial-env)
  (letrec ((the-environment
            (chain-environment (export the-environment)
                               initial-env ) ))
    (let loop ()
      ((turn-into-module (read)) the-environment)
      (loop) ) ) )

(create-user-toplevel-loop
 read-from-window0 pure-scheme-env )
(create-user-toplevel-loop
 read-from-window1 (create-user-toplevel-environment) )
```

Within these two toplevel loops, one may read and write the `the-environment` variable. In the first, one has no access to the `create-complete-environment` function but may already simulate hyperstatic environment *à la* ML just by saying:

```
(letrec ((fact ...))
  (set! the-environment (export)) )
```

By storing previous values of `the-environment` and reinstalling them, one is also able to regress in time or to switch among many different environments that may be managed as any other first-class value. The second toplevel loop is much more permissive.

6.4 Mutability

Currently, if a variable is exported, all its importers may mutate it, even if the author of the module did not want it. It is therefore necessary for the module author to restrict writing access to some exported variables. One way to achieve this is to only export *observer* functions as suggested in [FF86, Tun92] but this prevents the importation of mutable locations such as the `errno` variable from the C library. Another way is to enhance the syntax of `export` to specify which variables are exported read-only. Since `export` forms are special, they cannot be ignored by the compiler which exactly knows the status of every variable.

On the `import` side, the linking phase may check, when an imported variable is assigned in its body whether the imported environment allows it or not. Finally, a regular function can also be offered to make read-only a named location of a first-class environment. Note that only the enhanced `export` form allows the compiler to better handle non-assigned variables that are exported read-only.

7 Related Works

Pebble [BL84] introduced first-class bindings somewhat similar to first-class named locations (except that variables were immutable). Bindings may be paired to form first-class environments. Bindings may be made active in a scope using LET form. Nevertheless this form does not exactly comply with quasi-static discipline since variables cannot be unambiguously resolved with their sole names. This is, for instance, the case for x, in LET x:$\sim$ 3 IN LET b IN x, whose meaning depends on the actual type of b that may or may not contain a binding for x. An expression may be isolated from the current lexical context with IMPORT corresponding to our `(import () ...)` form.

Importation and exportation were analyzed in [FF86] in terms of syntaxes, that is without introducing new primitive special forms to Scheme. They essentially export values rather than locations. This is most of the time sufficient since values may be closures capturing mutable locations and offering some protocol to mutate them, delay their initialization etc. Nevertheless it is not possible to really share locations not containing functions nor to reify the entire environment and these are precisely the aspects our special forms were made for.

The module system of [CR90] for Scheme favors static development of large applications rather than dynamic creation of first-class environments. This is mainly due to the presence of syntaxes whose expansion must be controlled statically (as in [QP91, DPS94]) and to the research of efficiency, which requires accompanying locations with extra information qualifying their content and chiefly their type. Nevertheless we believe that our proposal is more useful in a context where many programs will be reused not through their text but from their runtime appearance.

First-class environments were known in T as *locales* but are also present in MIT-Scheme; a detailed presentation appears in [MR91]. Environments are created with `make-environment` and they are all extensible since computations may be performed in any first-class environment using a binary `eval` function. Therefore it is not possible to capture a finite set of locations and the problem of incremental definition occurs negating the quasi-static discipline. Apart the runtime burden suffered by `eval` which, compared to our `import`, has to compile expressions on the fly, a very efficient implementation of dynamic resolution of variables is proposed.

Interactive modular programming [Tun92] aims to allow the concurrent development of code within distinct module environments. Definitions are qualified as private or public to specify their scope. It is interactively possible to add or retract importations of modules inside a module environment; these operations affect the resolution of variables and thus negates quasi-static discipline.

Reflective first-class environment appear in [Jag94]. Importation and exportation are accomplished with operations respectively named `reflect` and `reify`. To ease the implicit manipulation of environments, these operations may have an additional argument: a closure standing for its closed environment. Variables may be qualified as public and are thus automatically capturable by a `(reify)` form. Reification barrier may also be installed to limit these captures. The implementation suggested in section 5 is similar to the "local caching" strategy of [Jag94].

The closest work is [LF93] whose goal was similar to us i.e. to share locations; they introduced the "quasi-static" concept from which we derive the quasi-static discipline that allows variables to be unambiguously resolved once and for all, a paramount quality in our eyes. They introduce two new special forms: `qs-lambda` and `resolve1`. A quasi-static procedure is returned by `qs-lambda` and represents a parameterized piece of code which may be applied as any regular function but may also be made more static if resolving (with `resolve1`) one of its parameterizing variables to some location. We consider this double behavior as a nuisance and favor first-class environment on which there exists a unique operation: the importation. Renaming is natively offered by `qs-lambda` as well as an original lexical inheritance mechanism relating parameterized variables of nested `qs-lambda` forms. We instead provide renaming and extension of first-class environments by functions that do not impact our `import` form. We also offer the possibility to manage implicit sets of free variables and extensible environments therefore providing a substrate on which it is possible to build or describe interactive toplevel loops.

8 Conclusion

The Scheme Repository is an example of an Internet resource containing interesting programs. Its usability is compromised by, at least, the presence of heterogeneous macros and implicit assumptions on how to process that code (compile, load, eval and their variation). After macroexpansion, such programs may be made more sharable if packaged as pure Scheme textual modules expecting an environment providing the required locations.

If the Net soon appears as a big repository of values and programs then to be able to capture these locations or values for later use will be of primary importance. Parts of applications may be released as modules and thus may be automatically upgraded whenever their imported locations are upgraded. To avoid these dependencies, other parts may be delivered as ready-to-use values or first-class environments. In both cases, first-class environments are "modern hooks" that represent a simple and controlled way to customize packages by operating on the true (locations of) variables.

Were we to summarize the significance of our paper, we would say that *(i)* it binds quasi-static variables with first-class environments, *(ii)* it introduces extensible environments avoiding the incremental definition problem, *(iii)* it reconciles efficiency of first-class environment and some reflective operations, *(iv)* it makes a clear distinction between the orthogonal roles of mapping names to locations and locations to values.

Acknowledgments

Many thanks to Luc Moreau and the program committee for their enlightning comments.

Bibliography

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *SIGPLAN Notices*, 23, September 1988. special issue.

[BL84] R Burstall and B Lampson. A kernel language for modules and abstract data types. Technical Report 1, DEC – SRC, September 1984.

[Car84] Luca Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217, Austin, Texas, August 1984. ACM Press.

[CR90] Pavel Curtis and James Rauen. A module system for scheme. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[CR91] William Clinger and Jonathan A Rees. The revised[4] report on the algorithmic language scheme. *Lisp Pointer*, 4(3), 1991.

[DPS94] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming*, pages 113–120, Orlando (Florida USA), June 1994. ACM Press.

[FF86] Matthias Felleisen and Daniel P. Friedman. A closer look at export and import statements. *Journal of Computer Languages*, 11(1):29–37, 1986.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, Austin, TX., August 1984.

[GJL87] David Gelernter, Suresh Jagannathan, and Thomas London. Environments as first-class objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 98–110. ACM Press, January 1987.

[Jag94] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transaction on Programming Languages and Systems*, 16(3):456–492, May 1994.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *LFP '88 – ACM Symposium on Lisp and Functional Programming*, pages 316–326, Snowbird, Utah, July 1988. ACM Press.

[LF93] Shinn-Der Lee and Daniel P Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *POPL '93 – Twentieth Annual ACM symposium on Principles of Programming Languages*, pages 479–492, Charleston (South Carolina, USA), January 1993. ACM Press.

[MR91] James S Miller and Guillermo J Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation: An International Journal*, 4(2):107–141, 1991.

[Ous93] John K Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1993.

[QP91] Christian Queinnec and Julian Padget. Modules, Macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[Que94] Christian Queinnec. *Les langages Lisp*. InterÉditions, Paris (France), 1994. ISBN 2 7296 0549 5, 61 2448 1, English version soon available from Cambridge University Press.

[RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition*. Yale University Computer Science Department, January 1984.

[Sun95] Sun Microsystems. *Java Language Specification*, 1995.

[Tun92] Sho-Huan Simon Tung. Interactive modular programming in scheme. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 86–95, San Francisco, USA, June 1992.