Indiana University
Computer Science Department

Technical Report 619
http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR619

Scheme 2005

Proceedings of the Sixth Workshop on Scheme and Functional Programming

September 24, 2005
Tallinn, Estonia

J. Michael Ashley and Michael Sperber, editors

Sponsored by the Association for Computing Machinery's Special Interest Group on Programming Languages (ACM/SIGPLAN)

Preface

This report contains the papers presented at the Sixth Workshop on Scheme and Functional Programming, on September 24, 2005, in Tallinn, Estonia.

The purpose of the workshop is to discuss experience with, and future developments of, the Scheme programming language, as well as general aspects of Computer Science loosely centered on the general theme of Scheme. The intention of the steering committee is that the workshop provide an annual focal point where the Scheme community can gather and share ideas: researchers, educators, implementors, programmers, hobbyists, and enthusiasts of all stripes—all welcome.

Eleven papers were submitted in response to the workshop's call for papers. Paper submission and review was conducted via electronic mail. Each paper was read by at least three reviewers including at least two members of the program committee. We are grateful to Matthias Neubauer for his service as outside reviewer.

Several others helped with the planning for the workshop. Olin Shivers provided tools that helped produce this technical report. Olivier Dancy, ICFP general chairman, and Patricia Johann, ICFP workshop chairman, were consistently helpful throughout the process. Tarmo Uustalu helped with the local arrangements. We are thankful for all of this support and assistance.

Some of the software described in the papers is available from or listed on the permanent home page of the workshop:

`http://www.deinprogramm.de/scheme-2005/`

J. Michael Ashley and Michael Sperber,
For the program committee

Program committee

Martin Gasbichler (University of Tübingen)
Jonathan Rees (Millennium Pharmaceuticals)
Dorai Sitaram (Verizon)
Jonathan Sobel (SAS Institute)

Contents

Type Classes Without Types *

Ronald Garcia Andrew Lumsdaine

Open Systems Lab
Indiana University
Bloomington, IN 47405
{garcia,lums}@cs.indiana.edu

Abstract

Data-directed programs consist of collections of generic functions, functions whose underlying implementation differs depending on properties of their arguments. Scheme's flexibility lends itself to developing generic functions, but the language has some shortcomings in this regard. In particular, it lacks both facilities for conveniently extending generic functions while preserving the flexibility of ad-hoc overloading techniques and constructs for grouping related generic functions into coherent interfaces. This paper describes and discusses a mechanism, inspired by Haskell type classes, for implementing generic functions in Scheme that directly addresses the aforementioned concerns. Certain properties of Scheme, namely dynamic typing and an emphasis on block structure, have guided the design toward an end that balances structure and flexibility. We describe the system, demonstrate its function, and argue that it implements an interesting approach to polymorphism and, more specifically, overloading.

1. Introduction

Data-directed programs consist of collections of *generic functions*, functions whose underlying implementation differs depending on properties of their arguments. In other words, a generic function is *overloaded* for different argument types. Data-directed style appears often in Scheme programs, even in the Scheme standard library. The standard generic arithmetic operators include functions such as `+` and `*`, which exhibit different behavior depending on what kind of arguments they are applied to. For example, applying `+` to two integers yields an integer value; adding two complex values, on the other hand, yields a complex value. A binary version of `+` could be implemented with the following general form:

```
(define +
  (lambda (a b)
    (cond
    [(and (integer? a) (integer? b))
     (integer-+ a b)]
     [(and (complex? a) (complex? b))
      (complex-+ a b)]
     ...
     [else (error "invalid arguments")])))
```

The body of `+` is simply a `cond` expression that tests its operands for various properties and dispatches to the implementation upon finding a match. Assuming specific implementations of addition for integers and complex numbers, the function dispatches to integer addition when the operands are integers, and complex numbers when the operands are complex. [1]

For all their benefits, generic functions implemented using `cond` as above have their shortcomings. Such functions are not meant to be extended to support new types of arguments. Nonetheless, such a function may be extended at the top-level using ad-hoc means as in the following:

```
(define +
  (let ([old-+ +])
    (lambda (a b)
      (cond
      [(and (my-number? a) (my-number? b))
       (my-+ a b)]
       [else (old-+ a b)]))))
```

A function may also be extended in a manner that limits the extension to the current lexical scope, as in the following:

```
(let ([+
       (let ([old-+ +])
         (lambda (a b)
           (cond
           [(and (my-number? a) (my-number? b))
            (my-+ a b)]
            [else (old-+ a b)])))])
  (+ my-number-1 my-number-2))
```

The above examples assume a user-defined number, of which `my-number-1` and `my-number-2` are instances, and a `my-number?` predicate that tests for such numbers. Both versions of `+` can handle these new numbers. Although the second example only introduces the new `+` in the scope of the `let` expression, the function could be returned as a value from the expression and subsequently used in other contexts.

These methods of extending `+` are ad-hoc. They don't directly capture the intent of the programmer, and much of the content is boiler-plate code. Another issue with this style of extending data-directed functions is that it does not respect the grouping of related functions. For example, the `+` operator is just one of a group of arithmetic operators that includes `*`, `-`, and `/` as well, and in general they should be introduced and extended together. Using the above method of introducing overloads, one must manually duplicate the

* This material is based on work supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

[1] This model disregards the possible coercion of arguments to match each other because such a mechanism is outside the scope of this work.

idiom for each operator, resulting in duplicate boilerplate code and no intentional structuring of the set of operators.

The Haskell [Pey03] language community has previously investigated overloading in the context of a statically typed language and as their answer to the problem produced the *type class* facility [WB89], which we describe later. Type classes are an elegant, effective approach to overloading and have spawned significant research that has advanced their capabilities [NT02, Jon00, CHO92].

This paper describes a language extension for Scheme that supports the implementation of groups of generic functions and their overloads. This system is heavily inspired by Haskell's type classes, but is designed to function in a latently typed language, where types appear as predicates on values. For that reason, we consider ours to be a *predicate class* system.

In order to fit with Scheme, this system differs from Haskell's type classes in some significant ways. Haskell is solely interested in dispatch based on static type information. In contrast, the ad-hoc method of constructing and extending generic functions can dispatch on arbitrary predicates, including standard predicates such as `number?` and `char?`, as well as user-defined predicates such as `my-number?` from the earlier examples. The described system also supports overloading based on arbitrary predicates. Also, whereas Haskell emphasizes compile-time type checking, error-checking is subservient to flexibility in this model. The overloading mechanism described here eschews the conservative practice of signaling errors before they are encountered at run time.

The combination of block structure, lexical scoping, and referential transparency plays a significant role in Scheme programs. Some of the previously discussed ad-hoc methods show how overloading can be performed in Scheme and how those methods fall short in lexical contexts. The predicate class system we present directly supports overloading functions under such circumstances.

Our overloading mechanism was implemented for Chez Scheme using the syntax-case macro system [DHB92, Dyb92], an advanced macro expansion system provided by some popular Scheme implementations that combines hygienic macro expansion [KFFD86] with controlled identifier capture. Because our system is implemented as macros, its semantics can be described in terms of how the introduced language forms are expanded (See Section 6).

2. Contributions

The contributions of this paper are as follows:

- A language mechanism for overloading is described, inspired by the type class model but modified to better match the capabilities and the philosophy of Scheme. Scoped classes and instances that allow both classes and instances to be shadowed lexically is an interesting point in the design space. Furthermore, expressing this facility in the context of a dynamically typed language allows some interesting design options and tradeoffs that are not available to statically typed languages.
- The described dynamic dispatch model combines the flexibility of ad-hoc techniques available in Scheme with a more structured mechanism for overloading functions. Previous mechanisms for overloading in Lisp and Scheme have pointed toward a relationship to objects and object-oriented programming. Our system supports dispatch based on arbitrary runtime properties. Furthermore, the predicate class model groups related generic functions into extensible interfaces.
- A point of comparison is provided between the overloading mechanisms expressed in statically typed Haskell and dynamically typed Common Lisp traditions.

3. A Brief overview of Haskell Type Classes

Haskell [Pey03] is a statically typed functional programming language, featuring Hindley/Milner-style type inference [Mil78, DM82] and its associated flavor of parametric polymorphism. Haskell also, however, supports a form of *ad-hoc* polymorphism, or overloading, in the form of type classes [WB89], which contribute substantial expressive power to the language. In order to introduce the concepts involved, and to provide a point of comparison, we briefly describe the type class system.

```
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
   x /= y         =  not (x == y)
   x == y         =  not (x /= y)

instance Eq Integer where
  x == y          =  x `integerEq` y

instance Eq Float where
  x == y          =  x `floatEq` y

elem :: (Eq a) => a -> [a] -> Bool
x `elem`  []      = False
x `elem` (y:ys)   = x==y || (x `elem` ys)
```

Figure 1. Haskell type classes in action

Consider the problem of specifying and using operators for numeric types, specifically the equality operator. Figure 1 illustrates how the equality operator is specified for Haskell in its Standard Prelude. First a *type class* is introduced. A type class is an interface that specifies a set of *class operators*, generic functions associated with a particular type class. The above type class definition essentially says "for all types `a` that belong to the class `Eq`, the operators `==` and `/=` are overloaded with values of the specified type signatures." The `Eq` class defines default implementations for `==` and `/=`, however in order to use them, a type must be explicitly declared to overload the type class functions. This role is played by *instance declarations*. An instance declaration declares membership in the type class and implements *instance methods*, specialized overloads of the class operators. For example, the first instance declaration for `Integer`s declares that `Integer` is a member of the `Eq` class, and provides an explicit overload for `==`. The `==` operator for `Integer` values is implemented in terms of a hypothetical `integerEq` operator defined solely for integers. [2] An analogous instance for floats is also presented. Both instance declarations inherit the default `/=` method, which will call the specific `==` overload associated with the type. In fact one may legally omit the `==` implementation as well, but then a call to either operator yields an infinite recursion. Finally, the `elem` function, analogous to Scheme's `member`, is presented. This generic function is not part of the `Eq` type class, yet still relies upon it. Its type, `(Eq a) => a -> [a] -> Bool` is *qualified* with `Eq` and essentially says "`elem` is overloaded for all types `a` that belong to `Eq`, in which case its type is `a -> [a] -> Bool`."

4. Language Description

The predicate class mechanism introduced in this paper forms an embedded language for overloading in Scheme and thus purely extends the existing language. This section introduces and describes the forms with which we extend Scheme to provide type class-like functionality. The extended language syntax is summarized in Figure 2.

[2] In Haskell, a binary function can be called in infix position by enclosing it in single back quotes

```
⟨definition⟩ +→
      (define-class (⟨identifier⟩ ⟨variable⟩+)
        ⟨op-spec⟩+)
    | (define-instance (⟨identifier⟩ ⟨expression⟩+)
        (⟨method-name⟩ ⟨expression⟩)+)
    | (define-qualified ⟨identifier⟩ (⟨identifier⟩+)
        ⟨expression⟩)
    | (define-open-qualified ⟨identifier⟩ (⟨identifier⟩+)
        ⟨expression⟩)

⟨expression⟩ +→
      (let-class ([(⟨identifier⟩ ⟨variable⟩+)
                   ⟨op-spec⟩+])
        ⟨body⟩)

    | (let-instance ([(⟨identifier⟩ ⟨expression⟩+)
                      (⟨method-name⟩ ⟨expression⟩)+])
        ⟨body⟩)

⟨op-spec⟩ → (⟨operator-name⟩ ⟨variable⟩*)
    | [(⟨operator-name⟩ ⟨variable⟩*) ⟨expression⟩]
```

Figure 2. Syntax extensions for type classes in Scheme

4.1 Predicate Classes

A predicate class is a form that establishes an interface for overloading. Predicate classes are introduced using either the `define-class` form, for top-level definitions, or the `let-class` expression, for lexically scoped definitions. The syntax that we use for these constructs is as follows: [3]

```
(define-class (class-name pv ...)
   op-spec
   ...)

(let-class ([(class-name pv ...)
             op-spec ...])
  expr ...)
```

The `define-class` form introduces a new predicate class at the top-level with the name *class-name*. The `let-class` form correspondingly introduces a type class that is visible within the scope of its enclosed body (*expr* ...). The name of the type class is followed by a list of *predicate variables* (*pv* ...). A class's predicate variables determine the number of predicate functions that will be used to establish an instance of a predicate class. The order of the predicate variables matters, and corresponds directly to the order of the predicates that are used to define an instance (as shown in the next section). Whereas Haskell type class instances are determined by the type used in an instance definition, predicate classes are determined by a list of Scheme predicate functions. This corresponds directly to the Haskell extension that supports multiple parameter type classes [PJM97]. Following the name of the class and its list of predicate variables is a list of *class operation specifications*, signified above by *op-spec*. Each operation specification takes one of the following two forms:

```
(op-name sym ...)

[(op-name sym ...) expr]
```

their purpose is to establish the names of the operators belonging to the class, as well as to specify which arguments will be used to determine dispatch based on which predicates. The second syntax for operation specifications illustrates how to supply a *default instance method* for a class operation. Each symbol *sym* marks an argument position for the operation. Any position marked with a predicate variable will be used to determine dispatch to the proper instance method. If a predicate variable is placed in an argument position, then a call to that class operation will use that argument position to test for instance membership: The instance predicate associated with the given predicate variable will be applied to the passed argument. Instances of the class are tested until an instance is found whose predicates return `#t` for each argument position marked with a predicate variable. The dispatch algorithm implies that the order in which instances are declared can affect the instance that a class operator dispatches to. In this regard, the mechanics of dispatch are analogous to the `cond` form of dispatch described earlier.

For example, consider the following rendition of the `Eq` type class in Scheme:

```
(define-class (Eq a)
  [(== a a) (lambda (l r) (not (/= l r)))]
  [(/= a a) (lambda (l r) (not (== l r)))])
```

This definition looks similar to the Haskell equivalent in Figure 1, but there are a few differences. A Haskell type class specification is used for type checking as well as dispatch. The class's type variable would be instantiated and used to ensure that code that calls the class operators is type safe. In the case of the above Scheme code, however, the predicate variable `a` simply specifies how to dispatch to the proper instance of a method. As written, calls to the `==` method determine dispatch by applying the predicate to both arguments. In some cases, however, the underlying implementations all require both arguments to have the same type. Under that assumption, one can optimize dispatch by checking only the first argument: the dispatched-to function is then expected to report an error if the two values do not agree. The following example shows how to implement such a single-argument dispatch:

```
(define-class (Eq a)
  [(== a _) (lambda (l r) (not (/= l r)))]
  [(/= a _) (lambda (l r) (not (== l r)))])
```

In the above code, the second reference to `a` in each of these operations is replaced with the underscore symbol (`_`). Since the underscore is not one of the specified predicate variables, it is ignored. Symbols that do not represent predicates are most useful, however, when dispatch is dependent on argument positions other than the first. For example in the form:

```
(define-class (Eq a)
  [(== _ a) (lambda (l r) (not (/= l r)))]
  [(/= _ a) (lambda (l r) (not (== l r)))])
```

dispatch is determined by the second argument to the operations.

Under some conditions, it is useful to develop a class that dispatches on multiple predicates, rather than two. For example, consider a type class that specifies overloaded operators that operate on vector spaces. A vector space must take into consideration both the sort of vector and scalar types used, and this can be done as follows:

```
(define-class (Vector-Space v s)
  [vector-add v v]
  [scalar-mult s v])
```

Notice that in particular, scalar multiplication takes a scalar as its first argument and a vector as its second. Classes that represent multi-sorted algebras are bound to have one predicate for each sort.

4.2 Class Instances

A class instance is an implementation of overloads for a specified predicate class that is associated with a particular list of Scheme

[3] Throughout the text, code uses square brackets (`[]`) and parentheses (`()`) interchangeably for readability. Several Scheme implementations, including Chez Scheme, support this syntax.

predicates. they are introduced using the `define-instance` form or the `let-instance` expression. The syntax for these constructs is as follows:

```
(define-instance (class-name pred ...)
  (method-name expr) ...)

(let-instance ([(class-name pred ...)
                (method-name expr) ...])
  expr ...)
```

The `define-instance` form introduces a new top-level instance of a previously declared class. The `let-instance` form correspondingly introduces a new instance of a class for the scope of its body (*expr* ...). An instance definition names the referent class followed by a list of Scheme predicates–functions of one parameter that verify properties of objects. Built-in examples include `integer?` and `boolean?`, but any function of one argument is acceptable (though not necessarily sensible). These predicates are used during dispatch to find the proper overload.

Following the class name and list of predicates is a list of method bindings for the class operations. The first component, *method-name* specifies the name of an operation from the class definition. The method binding, *expr*, should evaluate to a function that is compatible with the operation specification from the class definition. The expressions that define instance methods become suspended: the entire expression will be evaluated for each call to the method, therefore any side-effects of the expression will be repeated at each point of instantiation. Because this behavior differs from that for traditional scheme definitions, the expression that defines an instance method should simply be a `lambda` form or a variable. An instance declaration must have a method definition for each class operation that has no default.

The following code shows an instance of the above `Eq` class for integers:

```
(define-instance (Eq integer?)
  (== =))
```

Following the above definition, applying `==` to integers will dispatch to the standard `=` function. However, the class could be redefined in a controlled context using `let-instance` as follows:

```
(let-instance ([(Eq integer?)
                (== eq?)])
  ...)
```

Applications of `==` to integers in the `let-instance` form body will dispatch to the standard `eq?` function.

Class operations are not always open to additional overloads in this system. As shown later, they are implemented as identifier macros (also called symbolic macros), and when referenced expand to an *instantiation*. When a class operation is instantiated, the result is a function that may dispatch only to overloads of the operation that are defined visible at the point of instantiation. In particular, if a function definition calls a class operation, those calls will recognize no new lexical instance declarations introduced before the function itself is called. Continuing the `Eq` class example, consider the following program:

```
(define-instance (Eq char?) (== char=?))

(define elem
  (lambda (m ls)
    (cond
      [(null? ls) #f]
      [(== m (car ls)) #t]
      [else (elem m (cdr ls))])))

(let-instance ([(Eq char?) (== char-ci=?)])
  (elem #\x (list #\X #\Y #\Z)))
```

First an instance of `Eq` is defined for character types, using `char=?`. Next, the `elem` function is implemented. This function is analogous to the Haskell function from Figure 1. The `elem` function implements the same functionality as its Haskell counterpart, but due to the instantiation model of instance methods, calls to the function will dispatch based on the instances visible at the point that `elem` is defined. Thus, even though the next expression shadows the instance declaration for characters, using `char-ci=?` to implement `==`, the call to `elem` still dispatches to the first instance declaration, which uses the case-sensitive comparator, and the expression yields the result `#f`. Had the new instance been defined using `define-instance`, then `elem` would have used the case-insensitive comparator, and the above expression would have yielded `#t`.

4.3 Qualified Functions

The previous example illustrates how class operators and `let-instance` expressions preserve lexical scoping. Unfortunately, this introduces a difference between generic functions implemented as class operators and generic functions that are implemented as Scheme functions that apply class operators. It is beneficial to also have generic Scheme functions implemented in terms of class operators, that exhibit the same overloading behavior as class operators.

Haskell functions are overloaded by expressing their implementations in terms of class operators. When overloaded, a function type is then qualified with the type classes that define the operations used in the function body. Recall the `elem` function defined in Figure 1. It has qualified type `(Eq a) => a -> [a] -> Bool`, which expresses its use of type class operators.

Scheme functions require no such qualification to call class operators, but we borrow the notion to express our more dynamic generic functions, which we call `qualified functions`. Qualified functions take one of the following forms:

```
(define-qualified fn-name (class-name ...)
  expr)

(define-open-qualified fn-name
  (class-name ...)
  expr)
```

The function *expr* defined by this form is qualified by the list of classes (*class-name*...). Qualified functions have the same overload model as class operators. When referenced inside a `let-instance` form that overloads one of the qualifying classes, a qualified function's body can use the lexically introduced overloads. Qualified functions are also subject to instantiation. Inside a qualified function, the operations from the list of qualifying classes dispatch to the overloads visible at the point in the program where the function is *referenced*, rather than the point where the function was *defined*. As such, the behavior of the function can be overloaded at or around call sites. Furthermore, the expression that defines a qualified function is suspended in the same manner as for instance methods. It is thus expected that qualified functions will be implemented with `lambda` forms. However, qualified functions suffer this strange evaluation property in exchange for the ability to dynamically overload their behavior.

Revisiting the `elem` example from the previous section, the function is now defined using `define-qualified`:

```
(define-qualified elem (Eq)
  (lambda (m ls)
    (cond
      [(null? ls) #f]
      [(== m (car ls)) #t]
      [else (elem m (cdr ls))])))
```

The call to `==` within the function body will now dispatch based on the instances visible at the point that `elem` is called, rather than where it was defined. Using this definition of `elem`, the expression:

```
(let-instance ([(Eq char?) (== char-ci=?)])
  (elem #\x (list #\X #\Y #\Z)))
```

yields the value `#t`.

The following program illustrates a qualified function called in two different instance contexts:

```
(define-qualified ==-3 (Eq)
 (lambda (x y z) (and (== x y) (== y z))))

 (cons (let-instance ([(Eq char?)
                       (== char-ci=?)])
         (==-3 #\x #\X #\z))
       (let-instance ([(Eq char?)
                       (== (lambda (a b)
                             #t))])
         (==-3 #\x #\X #\z)))
```

The `==-3` qualified function performs a 3-way equality comparison. Both applications of `==-3` take the same arguments, but each application occurs within the scope of a different instance declaration. This results in dispatch to two different implementations of the `==` method inside the body of the qualified function: the first performing a case-insensitive comparison and the second always yielding `#t`. Evaluation of this expression yields the pair `'(#f . #t)`.

A self-reference inside the body of a function defined with `define-qualified` refers to the current instantiation of the function. However, if a function is defined with `define-open-qualified`, then a self-reference results in a new instantiation of the qualified function. Thus it is possible for such a qualified function to call itself with new instances in scope, as in the following (admittedly bizarre) example:

```
(let-class ([(B p) (o p)])
  (let-instance ([(B boolean?)
                  (o (lambda (x)
                       'boolean))])
    (define-open-qualified f (B)
      (lambda (x)
        (cons
         (o x)
         (if x
             (let-instance ([(B boolean?)
                             (o (lambda (x)
                                  x))])
               (f #f))
             '()))))
    (f #t)))
```

The above expression defines a class, `B`, that specifies one operation of one argument. It then establishes an instance of the class for booleans and defines a function `f` that is qualified over instances of the class. Calling `f` with the value `#t` results in a call to the instance method in the scope of only the outer `let-instance` definition. The result of this call, the symbol `'boolean`, is paired with the result of recurring on `f`, this time in the scope of an instance that implements the instance method `o` as the identity. The final result of these gymnastics is the list `'(boolean #f)`. Whether this functionality serves a useful purpose is a subject of future investigation.

5. Examples

Under some circumstances, a set of instance methods will be implemented such that each applies its own associated class operator. This makes sense especially when defining class instances for data structures that contain values for which instances of the same class exist. For instance, consider the following implementation of `Eq` for Scheme lists:

```
(define-instance (Eq list?)
  [== (lambda (a b)
        (cond
          [(and (null? a) (null? b)) #t]
          [(or (null? a) (null? b)) #f]
          [else (and (== (car a) (car b))
                     (== (cdr a) (cdr b)))]))])
```

This instance of `Eq` requires that `==` be overloaded for every element of the list. The nested calls to `==` in the Scheme implementation are resolved at runtime and will fail if the arguments are not members of the `Eq` class.

Scheme lists result simply from disciplined use of pairs and the null object (`'()`). As such, a more fitting implementation of equality would handle pairs and the null object separately, as in the following:

```
(define-instance (Eq null?)
  [== (lambda (a b) (eq? a b))])
(define-instance (Eq pair?)
  [== (lambda (a b)
        (and (== (car a) (car b))
             (== (cdr a) (cdr b))))])
```

Scheme programs often use lists as their primary data structure, and operate upon them with higher order functions, especially the standard `map` function. Nonetheless, lists are only one data structure among others, trees for instance, and it may be desirable to map a function over other such data structures. The Haskell standard library specifies an overloaded implementation of map, called `fmap`, which varies its implementation depending on the data structure over which it maps. Haskell supports overloading on *type constructors* [Jon93], and this functionality is used to implement generalized mapping.

In Haskell, the `fmap` function is the sole operator of the `Functor` constructor class, which is defined as follows:

```
class  Functor f  where
  fmap :: (a -> b) -> f a -> f b
```

The proper implementation of `fmap` for lists is the standard `map` function, and the instance for lists is simple:

```
instance Functor [] where
  fmap = map
```

where `[]` is the type constructor for lists.

What follows is a Scheme implementation of `fmap` in the same style as the Haskell version:

```
(define-class (Functor p)
  (fmap fn p))

(define-instance (Functor list?)
  (fmap map))
```

In order to match standard Scheme `map`, `fmap` is not curried. The analogous instance declaration for Scheme lists is shown above. Scheme has no notion of type constructor analogous to that in Haskell. This is especially clear in that Scheme lists are heterogeneous: any given list can contain any Scheme value, regardless of its type. Though Haskell considers type constructor to be distinct from types, Scheme has no such distinction, and a simple predicate, such as `list?` for lists, suffices.

Given the above definition of `Functor`, one might define a tree data type and an overload of `fmap` for it as follows:

```
(define-record tree-branch (left right))
(define-record tree-leaf (item))

(define-instance (Functor tree-branch?)
  (fmap
    (lambda (fn branch)
      (make-tree-branch
```

```
       (fmap fn (tree-branch-left branch))
       (fmap fn
             (tree-branch-right branch))))))


(define-instance (Functor tree-leaf?)
  (fmap
    (lambda (fn leaf)
      (make-tree-leaf
       (fn (tree-leaf-item leaf))))))

(fmap add1 (list 1 2 3))

(fmap (lambda (x) (fmap add1 x))
 (make-tree-branch
  (make-tree-leaf (list 1 2 3))
  (make-tree-leaf (list 4 5 6))))
```

This example uses Chez Scheme's record facility for defining data types. The syntax:

```
(define-record rname (slotn ...))
```

defines a new data type and along with it a constructor *make-rname*, a type predicate *rname?* that returns `#f` for any other scheme type, and accessors of the form *rname-slotn* for each element. Most Scheme implementations supply similar facilities.

First, two data types with which trees can be described, `tree-branch` and `tree-leaf`, are defined. Then for each of these data types an instance of `Functor` is defined. Each instance's implementation of `fmap` constructs a new record from the result of recursively applying `fmap` to its components. Finally, two examples of calls to `fmap` are shown. They yield the expected results: a data structure of the same shape with each number incremented by one.

The Common Lisp Object System (CLOS) [GWB91] is another example of a LISP system that provides support for generic functions and overloading. CLOS is an object-oriented system whose dispatch is primarily based on class identity, but it also supports the overloading of generic functions on based on specific values. For example, the CLOS method:

```
(defmethod == ((x number) (y (eql 7)))
  'never)
```

defines an overload of the `==` generic function that is called whenever the first argument is a number and the second argument is exactly equal to the number 7.

Since the system described here supports arbitrary predicates, it too can implement such overloads. The following Scheme code mimics the above:

```
(define-class (Eq a b)
  (== a b))
(define is-seven? (lambda (x) (eq? x 7)))
(define-instance (Eq number? is-seven?)
  [== (lambda (x y) 'never)])
```

A new version of the `Eq` class uses two predicate variables in order to establish the two separate overloads. Then an instance of `Eq` is declared using the `number?` predicate and a hand-crafted predicate that checks for equality to `7`.

6. Translating Predicate Classes to Standard Scheme

Since the predicate class facility that we describe here is implemented using Scheme macros, programs that use them correspond directly to traditional Scheme code, the output of macro expansion. In this section, we illustrate how programs written using this system can be understood in terms of the resulting Scheme code.

The system implementation relies on the syntax-case macro expander's controlled variable capture, as well as `define-syntax` macro definitions scoped within function bodies. However, forms like `let-class` and `let-instance` could be similarly implemented in terms of `letrec-syntax`.

A class definition form, `define-class` or `let-class`, introduces two artifacts to the final program. First, an empty class table is created. In this system, a class table is a list of entries, one for each instance of a class. Each entry in the table is a pair of vectors: a vector of predicates, and a vector of instance methods.

The class definition form also introduces a predicate dispatch function for each operation specified. Based on the operation specification, a function is created that searches the class table, trying to find a set of predicates that match the arguments passed to the function.

For example, consider again the `Eq` class:

```
(define-class (Eq a)
  [(== a _) (lambda (l r) (not (/= l r)))]
  [(/= _ a) (lambda (l r) (not (== l r)))])
```

For illustration purposes, the `==` operation dispatches on its first argument but the `/=` operation dispatches based on its second. The code resulting from this form is similar to what is shown in Figure 3.

The class definition introduces a class table, named `Eq-table`, which starts out empty. Next, the default instance methods are defined. Each default becomes the body of a lambda expression that takes a class table in order to implement recursion among the instance methods. Then for each class operation, `==` and `/=`, a dispatch function is introduced. This function is curried, first accepting a class table and then an arbitrary list of arguments. The bodies of the dispatch functions traverse the instance entries in the class table, searching for a match between the predicate `a` and the dispatch argument. Both dispatch functions access the predicate `a` as the first element of the vector of predicates. Since `==` dispatches based on its first argument, `==-dispatch` runs the predicate on `(car args)`, the first argument to the function, but `/=-dispatch` runs the same predicate on `(cadr args)`, its second argument. If the class had more than one predicate, each predicate would be tried on its corresponding argument in an attempt to detect a matching instance. Finally, `==-dispatch` applies to its arguments the first method in the method vector, `op-vec`, whereas `/=` applies the second method. Each instance is passed the current class table in order to properly support recursion among instance methods.

The instance definition forms, `define-instance` and `let-instance`, introduce new methods to the class instance table and ensure that those instances are visible. To do so, an instance definition produces code that updates the class instance table and defines identifier macros for each class operation. These macros, which are not shown for they are implementation details, cause class operations to recognize the new instance. For example, consider the following expression:

```
(let-instance ([(Eq integer?)
                (== =)])
 (cons
   (list (== 5 6) (/= 5 6))
   (list == /=)))
```

This program introduces an instance of the `Eq` class based on the standard `integer?` predicate, assuming the previously described definition of the class. The `=` function is named as the implementation of the `==` operator, and the `/=` is left undefined, thereby relying upon the default method. Within the scope of this instance definition, both `==` and `/=` are called with integer arguments, and the results are collected alongside their instantiations.

```
(define Eq-table '())

(define ==-default
  (lambda (Eq-table)
    (lambda (l r) (not ((/=-dispatch Eq-table) l r)))))

(define /=-default
  (lambda (Eq-table)
    (lambda (l r) (not ((==-dispatch Eq-table) l r)))))

(define ==-dispatch
  (lambda (Eq-table)
    (lambda args
      (letrec ([loop
                (lambda (table)
                  (let ([pred-vec (caar table)]
                        [op-vec (cdar table)])
                    (cond
                     [(null? table) (error "No matching instance...")]
                     [((vector-ref pred-vec 0) (car args))
                      (apply ((vector-ref op-vec 0) Eq-table) args)]
                     [else (loop (cdr table))])))])
        (loop  Eq-table)))))

(define /=-dispatch
  (lambda (Eq-table)
    (lambda args
      (letrec ([loop
                (lambda (table)
                  (let ([pred-vec (caar table)]
                        [op-vec (cdar table)])
                    (cond
                     [(null? table) (error "No matching instance...")]
                     [((vector-ref pred-vec 0) (cadr args))
                      (apply ((vector-ref op-vec 1) Eq-table) args)]
                     [else (loop (cdr table))])))])
        (loop  Eq-table)))))
```

Figure 3. Expansion of the `Eq` class

The following roughly illustrates the expansion of the above expression:

```
(let ([Eq-table
       (cons
        (cons (vector integer?)
              (vector (lambda (Eq-table) =)
                      /=-default))
        Eq-table)])
  (cons
    (list ((==-dispatch Eq-table) 5 6)
          ((/=-dispatch Eq-table) 5 6))
    (list (==-dispatch Eq-table)
          (/=-dispatch Eq-table))))
```

First the above code adds a new entry to the instance table reflecting the structure of the supplied instance. The entry is a `cons` of two vectors. The first contains the `integer?` predicate, or more generally all the predicates needed to describe the instance. The second vector holds the operators, in the same order as specified in `define-class` (instance operators may be specified in any order and they will be appropriately reordered). This entry is then added to the front of the table and bound to a new lexical variable `Eq-table`. As with the default method implementations, the user-supplied implementation of the `==` method becomes the body of a lambda. Since no implementation is provided for `/=`, the default implementation is substituted.

In this new lexical scope, identifier macros for the operators `==` and `/=` are introduced. These macros handle instantiation of class operators when they are referenced. Thus, following all macro expansion, the calls to the operators in the original code are transformed to calls to the dispatch functions, passing along the proper class table, and then applying the result to the intended arguments. As previously mentioned, class operations are not first class entities. Class operations are implemented using identifier macros, so each class operation expands to replace any reference to it with an expression that applies its associated dispatch function to the class table.

The `define-instance` form differs from its lexical counterpart in that it updates the class table in place. For example, the instance illustrated above could also be written as follows:

```
(define-instance (Eq integer?)
                 (== =))
```

And its expansion is as follows:

```
(set! Eq-table
      (cons
       (cons (vector integer?)
             (vector (lambda (Eq-table) =)
                     /=-default))
            Eq-table))
```

Rather than lexically binding a new table to extend the old one, it applies side effects to the existing table to add the new instance entry.

The `define-qualified` form introduces functions that look up class operation overloads visible at the point where the function is referenced, rather than where the function is defined. To implement

such functionality, this form introduces an implementation function that takes one class table argument for each class that qualifies it. Consider, for example, the following qualified function:

```
(define-qualified assert-equal (Eq)
  (lambda (a b)
    (if (/= a b)
      (error "Not equal!"))))
```

This function uses whatever instance of `Eq` matches its arguments at its instantiation point to test them for inequality. This program expands to the following:

```
(define assert-equal-impl
  (lambda (Eq-table)
    (letrec
      ([assert-equal
        (lambda (a b)
          (if ((/=-dispatch Eq-table) a b)
              (error "Not equal!")))])
      assert-equal)))
```

The `define-qualified` form generates the above function, which takes a class instance table and uses it to dispatch to the proper implementation of the `/=` method, as reflected by the call to `/=-dispatch`. The body of `assert-equal` is wrapped within a `letrec` form and bound to the name `assert-equal` so that self-references refer to the current instantiation. At the top-level, the name `assert-equal` is bound to a macro whose expansion applies the implementation function to the class table. For example, consider the following expression:

```
(cons (assert-equal 5 5)
      assert-equal)
```

Its expansion takes the following form:

```
(cons ((assert-equal-impl Eq-table) 5 5)
      (assert-equal-impl Eq-table))
```

The references to `assert-equal` expand to apply the implementation function, `assert-equal-impl`, to the newly extended class table.

The `assert-equal` qualified function can be implemented using the `define-open-qualified` form as follows:

```
(define-open-qualified assert-equal (Eq)
  (lambda (a b)
    (if (/= a b)
        (error "Not equal!"))))
```

Then only the expansion of the implementation function differs, as shown in the following:

```
(define assert-equal-impl
  (lambda (Eq-table)
    (lambda (a b)
      (if ((/=-dispatch Eq-table) a b)
          (error "Not equal!")))))
```

In this case, the body of the function is no longer wrapped within a `letrec` form. Thus, calls to `asset-equal` within the body of the function refer to the aforementioned macro and are expanded as described above.

7. Related Work

Although type classes in particular have been studied in the statically typed functional programming languages, overloading in general has also been added to dynamically typed programming languages.As mentioned earlier, for example, the Common Lisp Object System (CLOS) provides many of the benefits of an object-oriented programming language. Its design differs from other object-oriented languages in that operations are implemented using *generic functions* in the form of overloaded methods. These methods differ from the methods of most object-oriented languages in that they are not represented as messages passed to an object. Rather they are applied like Lisp functions, but each generic function name can refer to multiple method definitions, each supplied with a different set of *parameter specializers*. This mechanism applies to more than user-defined Lisp classes. Lisp methods can also be overloaded based on native Lisp types as well as equality requirements. Furthermore, specialization can be determined based on arbitrary argument positions in a method. As such, some consider the CLOS generic functions to be a generalization of the typical object-oriented style.

The following code illustrates the implementation of generic methods in Common Lisp:

```
(defmethod == ((x number) (y number))
  (= x y))
(defmethod == ((x string) (y string))
  (string-equal x y))
(defmethod != (x y)
  (not (== x y)))
(defmethod == ((x number) (y (eql 7)))
  'never)
```

The `defmethod` special form is the means by which Common Lisp code expresses generic functions. Each call to `defmethod` introduces an overload. The first two lines establish overloads for the `==` function, one for numbers and one for strings. Each indicates its overload by listing the types of its arguments, and uses the appropriate concrete function to implement the overload. Next, a `!=` generic function is implemented with one overload that places no constraints on its arguments. Its body is expressed in terms of the previously defined `==` function. Finally, a curious overload of the `==` function specifies different behavior if its second argument is the number `7`. Given this definition, the expression `(== 7 7)` yields the symbol `'never`.

Although standard Scheme does not specify a mechanism for implementing overloaded functions, rewrites of the CLOS mechanism are available for certain Scheme implementations [Xer, Bar]).

Overloading functions in Scheme has been the subject of previous research. In [Cox97], a language extension for Scheme is described that adds a mechanism for overloading function definitions. The forms `lambda++` and `define++` extend the definition of an existing function, using either user-supplied predicates or an inferred predicate to determine the proper implementation. In this regard it is similar to the Common Lisp Object System. However, it differs from CLOS in that the implementation combines all overloads at compile time and generates a single function with all dispatch functionality inline. Our design is fully implemented within a macro system, whereas this extension requires modifications to the underlying Scheme implementation.

Other programming languages have also investigated models of overloading. Cecil [Cha93] is a prototype based (or classless) object-oriented programming language that features support for multi-methods [Cha92]. It differs from systems like CLOS in that each method overload is considered to be a member of all the objects that determine its dispatch. These methods thus have privileged access to the private fields of those objects. Cecil has a very flexible notion of objects, and since objects, not types, determine dispatch for Cecil multi-methods, it can capture the full capability of CLOS generic functions, including value equality-based parameter specializers. Furthermore, Cecil resolves multi-method calls using a symmetric dispatch algorithm; CLOS uses a linear model, considering the arguments to a call based on their order in the argument list.

MultiJava [CLCM00] is an extension to the Java [GJSB00] programming language that adds support for symmetric multiple dispatch, as used in Cecil. This work emphasizes backward-compatibility with Java, including support for Java's static method overloading mechanism alongside dynamic multi-method dispatch.

Recently, the language F^G [SL05], an extension of the polymorphic typed lambda calculi of Girard and Reynolds [Gir72, Rey74], introduced mechanisms similar to the design described here. It introduces `concept` and `model` expressions, which are analogous to `let-class` and `let-instance`. It also adds a notion of *generic functions*, which are analogous to our qualified functions, as well as closely related to Haskell overloaded functions. Generic functions can have qualified type parameters much like Haskell, but the dispatch to its equivalent of instance operators is also based on the instances visible at the point of a function call. Generic functions do not have a notion of instantiation however: they have first class status and can be called elsewhere yet still exhibit their dynamic properties. The language F^G is statically typed but its type system does not perform type inference. In this language, instances of a class that have overlapping types cannot exist in the same lexical scope. Our system allows them, but recognizes that they may lead to undesirable results. Furthermore, F^G does not have top-level equivalents to `define-class` and `define-instance`.

In [OWW95], an alternative facility for overloading in the context of Hindley/Milner type inference is described. The language, named System O, differs from Haskell type classes in that overloading is based on individual identifiers rather than type classes. A function may then be qualified with a set of identifiers and signatures instead of a set of type class constraints. Compared to Haskell type classes, System O restricts overloading to only occur based on the arguments to a function. Haskell, in contrast, supports overloading on the return type of a function. As a result of System O's restrictions, it has a dynamic semantics that can be used to reason about System O programs apart from type checking. Haskell type class semantics, on the other hand, are intimately tied to the type inference process. Because of this, it is also possible to prove a soundness result with respect to the type of System O programs. Furthermore, every typeable term in a System O program has a principal type that can be recovered by type inferencing (types must be explicitly used to establish overloads however).

System O's dynamic semantics are very similar to those of the system we describe. Overloaded functions are introduced using the form:

```
inst o : s = e in p
```

where `o` is an overloaded identifier, `s` is a polymorphic type, `e` is the body of the overload, and `p` is a System O expression in which the overload is visible. This form is analogous to our `let-instance` form. However, `inst` introduces an overload only on identifier `o`, whereas `let-instance` defines a set of overloaded operators as described by the specified class.

Overload resolution in System O searches the instances lexically for a fitting overload, much like our system does. As such, System O's dynamic semantics allow shadowing of overloads, as our system does, but the type system forbids this: overloads must be unique. System O's overloaded operators are always dispatched based on the type of the first argument to the function. Our system, however, can dispatch based on any argument position, and uses arbitrary predication to select the proper overload. Also, our system can use multiple arguments to determine dispatch. Finally, though System O's dynamic semantics closely match those of our system, it can be still be implemented as a transformation to the more efficient dictionary-passing style that is often used to describe Haskell type classes.

Some Scheme implementations provide the `fluid-let` form, which supports controlled side-effects over some dynamic extent. To understand how `fluid-let` behaves, consider the following program and its result:

```
(let ([x 5])
  (let ([get-x (lambda () x)])
    (cons (fluid-let ([x 4]) (get-x))
          (get-x))))

=> (4 . 5)
```

The code above lexically binds `x` to the value `5`, and binds `get-x` to a function that yields `x`. Then, two calls to `get-x` are combined to form a pair. The first is enclosed within a `fluid-let` form. The `fluid-let` form side-effects `x`, setting its value to `4` for the dynamic extent of its body. The result of the `fluid-let` form is the result of its body, but before yielding its value, `fluid-let` side-effects `x` again, restoring its original value. Thus, the code:

```
(fluid-let ([x 4]) (get-x))
```

is equivalent to the following:

```
(let ([old-x x] [t #f])
  (set! x 4)
  (set! t (get-x))
  (set! x old-x)
  t)
```

The value of x is stored before assigning `4` to it. Then `get-x` is called and its value stored before restoring `x` to its old value. Finally the expression yields the result of `(get-x)`.

This mechanism is in some respects comparable to our predicate class mechanism. For example, consider the following program:

```
(let ([== #f])
  (define is-equal?
    (lambda (a b) (== a b)))
    (fluid-let ([==
                 (lambda (a b)
                   (if (number? a)
                       (= a b)))])
      (is-equal? 5 5))))
```

It binds the lexical variable `==` to a dummy value, `#f`. Then a function `is-equal?` is implemented in terms of `==`. Finally `==` is effected via `fluid-let`, and within its extent, `is-equal?` is called. This entire expression evaluates to `#t`. Compare the above program to the following, which is implemented using predicate classes:

```
(let-class ([(Eq a) (== a _)])
  (define-qualified is-equal? (Eq)
    (lambda (a b) (== a b)))
  (let-instance ([(Eq number?) (== =)])
    (is-equal? 5 5)))
```

It yields the same result as the `fluid-let` example. Here, `==` is introduced using the `let-class` form. Also, `is-equal?` is now implemented as a qualified function. Then `let-instance` replaces the `fluid-let` form. Due to this example's simplicity, the extra machinery of predicate classes exhibits some syntactic overhead, but programs involving more structure and content may be better formulated using type classes than using `fluid-let` to manually implement the same functionality.

8. Conclusion

Predicate classes loosely determine what properties may guide function dispatch. Traditional object-orientation determines dispatch based on one entity involved in a method call: the class to which the method belongs. Some operations, however, require dispatch based on more than the type of one entity. Idioms such as

the Visitor pattern [GHJV95] have been invented to support dispatch based on multiple types in object-oriented languages. Haskell type classes support dispatch based on all the arguments to a function. However, they specifically rely upon the types of function arguments to guide dispatch. Types can encode some sophisticated properties of objects, including relationships between them, but they cannot capture all runtime properties of programs. Common Lisp generic functions also dispatch on the types of arguments, but as shown earlier, they also support predication based on the particular value of an argument. In this regard, some runtime properties of values are available for dispatch. In our system, any Scheme predicate, meaning any function of one argument that might yield `#f`, can be used to define an instance. Thus, any predicate that is writable in the Scheme language can be used to guide dispatch. Predicates may mimic types, as in the standard Scheme predicates like `integer?`, and one may also compare an argument to some constant Scheme value, just as in Common Lisp. As such the mechanism described here can subsume much of the generic function mechanism in Common Lisp. The Common Lisp Object System, however, orders function specializations based on the inheritance hierarchy of any objects passed as arguments as well as their ordering in the function call. This differs from the Scheme system, which matches symmetrically across all predicates but also relies upon the ordering of and lexical distance to instance definitions. Thus, one may mimic this behavior, but such simulation depends upon the ordering of instance definitions.

The structure imposed by predicate classes provides a means to capture relationships between operations. Related functionality can be described as a unit using a class and subsequently implemented as instances of that class. Applications can thus use predicate classes to organize problem domain abstractions systematically and render them in program text. Such is the organizational power commonly associated with object-orientation; however, CLOS implements an object-oriented system that places less emphasis on the discipline of grouping functionality, preferring to focus on the expressiveness of generic function dispatch. The predicate class mechanism expresses the organization of objects but retains the emphasis on functions, rather than objects, generally associated with functional programming.

The flexibility of dynamic typing must, however, be weighed against the limitations imposed by a lack of static information during compilation. A static type system imposes some limitations on how programs can be written, but this rigidity in turn yields the ability for the language implementation to infer more properties from programs and use this extra information to increase expressiveness. For example, consider the following sketch of the standard Haskell `Num` type class:

```
class Num a where
  ...
  fromInteger :: Integer -> a
  ...
```

The `Num` type class has operations whose arguments do not contain enough information to determine how dispatch will proceed. Specifically, the `fromInteger` method, when applied to an `Integer` value, yields a value of type `a`, where `a` is the overload type. Since this method always takes only an integer, it relies on the return type to distinguish overloads, a feature that our system does not support. In order to implement something like the above in Scheme, the operations must take an additional argument that determines dispatch:

```
(define-class (Num a)
   ...
  (fromInteger a i)
  ...)
```

Here, the `fromInteger` method has an extra parameter, which must be the type to which the supplied integer is converted. Such a contortion is significantly less expressive than the Haskell analogue: a value of the proper type must be available in order to convert another `Integer` to it. This value's sole purpose is to guide dispatch. The change gives the `Num` class an object-oriented feel and muddies the abstraction with implementation details.

In the case of multiple parameter classes, operations need not be dependent upon all the class predicates. Despite interest in and known uses for multiple parameter type classes for Haskell, as well as support for them in several implementations, type checking of programs that make use of them is undecidable in the general case. Nonetheless they are considered useful, and various means have been proposed to make them tractable [Jon00, PJM97, DO02, CKPM05]. In the Scheme system, lack of dispatch information can also be problematic, especially if multiple instances of the class have overlapping predicates. A call to an operation with this sort of ambiguity results in the most recent instance's operation being called. An implementation of this system could recognize such ambiguities and report them as warnings at compile-time and as errors at runtime, but the system we describe here does not.

In Haskell, a class instance method can be overloaded for some other class. In this manner, even a particular method can utilize ad-hoc polymorphism in its implementation. Since methods in the Scheme system are implemented using macros, it is not possible to implement an instance method as a qualified function. One may use such a function as a class method, but it will be bound to a particular class table at the point of its definition so it will not be dynamic over its class qualifications.

As with Haskell, classes that qualify a function must not have overlapping operation names. However, multiple classes whose operation names overlap can be defined, but the behavior for this situation is rather idiosyncratic. Suppose two predicate classes share an operation name. Then at any point in the program, the method name corresponds to the class with the instance that is most recently defined (at the top level using `define-instance`) or most closely defined (using `let-instance`). Thus, instance definitions introduce, or re-introduce, their method names and in doing so shadow the value most recently associated with those names. One may still use commonly-named methods from multiple classes, but this requires the lexical capture of one class's instance method prior to defining an instance of the other class.

Haskell type classes model more of the functionality typical of object-oriented mechanisms than the described Scheme system. For example, type classes can derive from other type classes, much as an object-oriented class can be derived from another using inheritance. The predicate class mechanism does not support the derivation of one type class from another, but this functionality could be added to the system.

Combining the top-level class and instance definitions of Haskell with lexically scoped class and instance definitions increases expressive power. The ability to override an instance declaration as needed lends flexibility to how applications are designed. For example, an application may establish some problem-specific abstractions using classes and provide some default instances for them to handle the common cases. Nonetheless, any portion of the application that uses instance methods or qualified functions may now override the default instances in a controlled fashion as needed. Haskell could also benefit from this capability, though we are unaware of any investigation of such functionality for Haskell.

9. Acknowledgments

This work benefited from discussions with R. Kent Dybvig, Jeremy Siek, and Abdulaziz Ghuloum as well as collaborations with the latter two.

References

[Bar] Eli Barzilay. *Swindle*. http://www.barzilay.org/Swindle/.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, 1992.

[Cha93] C. Chambers. The Cecil language: Specification and rationale. Technical Report TR-93-03-05, 1993.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes (extended abstract). In *1992 ACM Conference on Lisp and Functional Programming*, pages 170–181. ACM, ACM, August 1992.

[CKPM05] Manuel M. T. Chakravarty, Gabrielle Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[Cox97] Anthony Cox. Simulated overloading using generic functions in Scheme. Master's thesis, University of Waterloo, 1997.

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

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[DO02] Dominic Duggan and John Ophel. Type-checking multi-parameter type classes. *J. Funct. Program.*, 12(2):133–158, 2002.

[Dyb92] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Computer Science Department Technical Report #356, Indiana University, Bloomington, Indiana, June 1992.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[Gir72] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thse de doctorat d'état, Université Paris VII, Paris, France, 1972.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[GWB91] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Commun. ACM*, 34(9):29–38, 1991.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61, New York, NY, USA, 1993. ACM Press.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. 9th European Symp. on Prog. (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244, New York, NY, March 2000. Springer-Verlag.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[NT02] Matthias Neubauer and Peter Thiemann. Type classes with more higher-order polymorphism. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 179–190, New York, NY, USA, 2002. ACM Press.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM Press.

[Pey03] Simon Peyton Jones. The Haskell 98 language. *Journal of Functional Programming*, 13:103–124, January 2003.

[PJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Proceedings of the 1997 Haskell Workshop*, June 1997.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, NY, 1974. Springer-Verlag.

[SL05] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, 2005. ACM Press.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM Press.

[Xer] Xerox PARC. *Tiny-CLOS*. ftp://ftp.parc.xerox.com/pub/mops/tiny/.

Eager Comprehensions in Scheme

The design of SRFI 42

Sebastian Egner

Philips Research Laboratories, The Netherlands

sebastian.egner@philips.com

Abstract

This article is about a certain style of programming iterative programs. It is based on a concept we have named "eager comprehension," which is a convenient and efficient alternative to tail recursion, `do`-loops, and lazy list comprehensions (aka "ZF expressions"). Eager comprehensions are syntactic forms that encapsulate the details of an accumulation process (counting elements, creating a list, etc.). Within these forms, expressions called generators hide the details of enumerating basic sequences (running through a list, through a range of integers, etc.). By combining these elements in a clearly structured and well-defined way, a concise and powerful notation for writing loops emerges.

Of course, this style of programming is not new—it is implicitly present in any form of `loop`-macro already—and so we discuss several concrete designs that aim for the same goal. Surprisingly, however, none of these designs has had much impact on Scheme, despite the fact that their common floor plan has been around for decades. A particularly clean new design, SRFI 42, on the other hand has already made some friends in the first few years of its existence. Explaining the design and implementation of SRFI 42 constitutes the main part of this article.

1. Introduction

The original motivation for working on a library for comprehensions in Scheme was my dissatisfaction with the available mechanisms for writing trivial loops. In addition, I wanted to create an efficient mechanism for converting data structures without a quadratically increasing number of conversion operations named *chalk->cheese*.

The most basic example for a trivial loop is the construction of a list of the first n non-negative integers, using the constructs available in the Revised^5 Report on the Algorithmic Language Scheme ($\mathrm{R^5RS}$) [1] only. Maybe the shortest[1] and clearest (!?) expression for this is

```
(do ((k (- n 1) (- k 1))
     (x '() (cons k x)) )
  ((< k 0) x) )
```

[1] Please let me know if you can do shorter than this in $\mathrm{R^5RS}$.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

This is terrible, not so much for the number of key strokes but as an example where details obscure intention.

In SRFI 42 that would be `(list-ec (: k n) k)`, for interactive use, or `(list-ec (:range k n) k)` if speed is worth another five key strokes. Since this article is not about the specification of SRFI 42, but about the design principles, a basic familiarity with the following document will be assumed from now on:

`http://srfi.schemers.org/srfi-42/srfi-42.html`

(Alternatively, there is a brief introduction in the appendix.) While initially the goal was adapting the comprehensions found in Haskell to Scheme, a number of insights turned this enterprise into a whole new direction and eventually led to the concept later coined "eager comprehensions." These ideas can be summarized as follows:

1. Truly lazy comprehensions are not an attractive option in Scheme because the overhead for non-strict data structures and explicit handling of continuations is high. Moreover, lazy comprehensions can be confusing in the presence of side-effects.
2. While list comprehensions and list generators are sufficient for comprehensions in lazy languages, in eager languages it is essential to be able to add application-specific comprehensions and generators easily—and without modifying the existing ones.
3. While simple comprehensions resemble mathematical set comprehensions, more complex expressions increasingly look like nested and parallel loops with accumulation of the results. In fact, that is what they are.
4. The fundamental eager comprehension has nothing to do with lists, but executes a command repeatedly according to its generators. The fundamental eager generator repeatedly modifies a state explicitly.

After these insights it was obvious that "bringing Haskell's comprehensions to Scheme" is the wrong goal to pursue. The more interesting question is "What would be a useful corresponding concept in an eager programming language?" The answer is quite surprising:

> *Eager comprehension:* A convenient style of programming nested and parallel loops with accumulation of results. Ideally, scope and syntax are easy to remember and the irrelevant details of the iteration are hidden from the user.

The concept can also be interpreted as an (essentially syntactic) abstraction mechanism from details of iteration: if you have a new data structure that has given rise to some natural iteration, then it might pay to encapsulate the details of this iteration process in a generator. Similarly, if there is a natural way of constructing a data structure from a sequence of states—a comprehension might be useful to applications.

Structure of this article. The remainder of this article is organized as follows. In Section 2 different notions of "comprehension" are introduced. These notions are related but must not be confused. In particular the term "eager comprehension" is being used as a reminder that this concept has in fact very little to do with lazy list comprehension.

Section 3 continues with stating the major design issues for (eager or lazy) comprehensions as a general and practically useful language construct. Section 4 discusses a number of concrete designs of loop facilities and comprehensions for the Lisp family of languages, and related work. The implementation of SRFI 42 is the subject of Section 5. It explains the overall structure and implementation strategy used in the reference implementation (which, unfortunately performance-wise, is the only one available till today). In Section 6 the performance of the portable reference implementation is compared with other libraries. Finally, for entertainment, Section 7 presents a modular way of adding *lazy* comprehensions to SRFI 42.

2. Concepts of comprehensions

In this section we briefly review different concepts of comprehension. For the sake of clarity we will always refer to them by a longer name than just "comprehension."

Set comprehension. The mathematical notation[2]

$$\text{“}\{f(x) \mid P(x), x \in S\}\text{”}$$

denotes the set of all values of the function f for arguments in the set S and satisfying the predicate P. The notation explicitly refers to a candidate element x, a predicate P, a universe S, and a mapping f. This notation is called a *set comprehension*.

The stated form is maybe the most frequently seen, but it is not the most fundamental. The most basic form of set comprehension is $\{y \mid Q(y)\}$, where $y = f(x)$, $P(x)$, and $x \in S$ have been combined into $Q(y)$. This concept, i.e. denoting a set defined by a predicate (formula), is the core of what is meant by "comprehension." While this concept has been in use for a long time already, it was not before the development of axiomatic set theory by Zermelo, Fraenkel and others in the 1920s that the idea was studied systematically. The notion of set comprehension, and its notation, is so natural that it has gradually become a mathematical standard, i.e. the reader of a mathematical article is expected to understand it without definition.

Comprehensions in programming languages. The notational convenience of set comprehension has inspired programming language constructs with similar intent: name the data structure defined by an expression for its elements. For example, in the SETL language [19, 20]

```
{n**2 : n in {0..9}}
```

denotes the set of the first ten integer squares. This construct, however, does not only specify the result but also an *algorithm* for constructing the result ("execute a loop over `n`, square the values, collected them in a data structure"). It is often convenient to ignore the algorithmic aspect, but most of the time this is not possible—after all, algorithms do take time, or may not terminate at all. For this reason, set comprehension in mathematics and in programming languages should never be confused.

Lazy comprehensions. While comprehensions were contained in some eager (aka call-by-value) programming languages for a long time, they only became popular once they were introduced for lazy lists in lazy (aka call-by-need) functional programming languages (mostly based on a typed λ-calculus with normal order reduction) in the early 1980s. In contemporary syntax (Haskell), for example,

```
[x*x | x <- [0..]]
```

denotes the (infinite) lazy list of all integer squares. Its elements will be made explicit once they are needed.

	primitive	*purpose*
set	set	denote a set by properties
lazy	lazy list	sequence processing
eager	side-effect	writing nested and parallel loops with accumulation of results

Table 1. Different concepts of "comprehension."

Such a *lazy comprehension* provides a convenient notation for processing lazy lists by means of mapping, filtering, and concatenation. The primitive lazy comprehension (written `[`*exp*`|`*qual*$^+$`]` in Haskell) constructs a lazy list, and the primitive lazy generator (`<-`, read 'drawn from') binds a variable[3] to the elements of a lazy list. In addition, several generators can be nested, elements can be filtered from the sequence, and local variables can be defined.

Lazy list comprehensions are widely accepted due to their concise notation, and good readability in most cases. Their efficiency is as good as any (lazy) alternative. Their primary shortcoming is an implicit tendency to overuse them, i.e. to write complicated nested lazy comprehensions where an appropriate abstraction had better been introduced. The decreasing readability of more complicated lazy comprehensions is probably due to the use of infix operators and the "$[expr|outer..inner]$" scoping rule, which is not simply left to right.

From the point of view of programming language design, it is most informative to recall the historical development of lazy comprehensions [18, Chapter 7]—in particular that their true nature was not fully understood for a long time: lazy comprehensions were first introduced as part of the NPL language (Burstall, 1977) [25]. In NPL, however, comprehensions construct a set of objects. While this construct is closest to the mathematical notion, *lazy sets* are not nearly as useful as *lazy lists* are. It appears that lists and graphs are more fundamental to programming than sets (unordered collections); in addition, lists (in particular lazy lists) are a universal and natural mechanism of communication between different parts of a program. Consequently, set comprehensions were not essential and when NPL evolved into the Hope language (Burstall, 1980) [26] lazy comprehensions where not included.

Lazy list comprehensions made their debut in the KRC language (Turner, 1981) [27] as "ZF expressions." Later they were included in several other functional programming languages like Miranda [28, 29] (Turner, 1985). But still mathematical beauty has distracted the mind from proper programming pragmatics for some time: generators in lazy list comprehensions can denote infinite iterations. Hence, from a mathematical point of view the most natural way of advancing nested generators is by (Cantorian) diagonalization, also known as "dove-tailing." This is the only way of reaching every pair eventually in the case of an infinite inner generator. While diagonalization looks like a good idea at first, it is not. Mathematical "eventually" can be a long time, and in practice diagonalization is not worth a lot. Thus lazy list comprehensions evolved to run the generators in the straight-forward way, i.e. exhausting the inner loop before advancing the outer loop, while the diagonalizing variants slowly went out of fashion (not without constantly being reinvented).

[2] Instead of "|" also ":" and "," are in use.

[3] A pattern possibly containing variables to be precise.

Specialized eager comprehensions. Encouraged by the success of lazy comprehensions, designers of eager programming languages recently started to include comprehensions again. E.g. Python [11] contains list comprehensions. While these *eager comprehensions* can be quite useful, in particular for interactive use and scripting, they are much less universal in nature than their lazy counterparts. This is explained in greater detail in Section 3.3.

Eager comprehension as abstraction of iteration. Surprisingly, this perceived limitation is again due to a lack of understanding for the true nature of comprehensions, eager comprehensions this time. As explained above, lazy comprehensions for lists are fundamental. For eager comprehensions, however, *side-effect* (state) is the most basic concept[4].

As indirect evidence of this fact consider that any eager comprehension can be implemented in terms of

(do-ec *qualifier** *command*),

which executes *command* for each state in the sequence defined by the generators and tests *qualifier**. Similarly, each eager generator can be implemented in some form of state-transforming iterator, in the sense of `do`. Amazingly, this insight—which is made explicit here—is already implicit in the design of nearly any loop facility for the Lisp family of languages, but it has not been acknowledged as such.

While the "can be implemented by"-relation usually does not lead to the most fundamental concept, it does so in this case. Consider the alternative of implementing the eager comprehensions in terms of (eager) lists: the resulting implementation will be horrible! An accumulation process (e.g. counting) cannot start until the last element of the enumerated sequence has been produced and stored in a data structure. The resulting loss in performance, as a function of sequence length, is in fact unbounded.

Being built on this insight, SRFI 42 eventually reduces any comprehension to `do-ec`, and any generator to `:do`—which is some flexible but fixed loop structure (Section 5.2) based on explicit state transformation. In combination with a number of rules simplifying the syntax and introducing a clean scoping rule, this results in a facility for iteration that is both efficient and convenient.

3. Design considerations

In this section we discuss the main issues that affect the usefulness of a programming language construct for eager comprehensions, or for writing nested and parallel loops with accumulation of the result. We approach these issues by exploring design alternatives: which design decisions exist and what are their implications? Our primary goal is not a coherent and complete theory, but rather an informal discussion of the relative benefits of various designs in terms of convenience and effectiveness of the language construct for writing programs.

3.1 Mental complexity

Maybe the most important consideration is what could be called "mental complexity." As an anecdotal quantitative measure of mental complexity we propose to count the number of times the reference manual of a loop construct was consulted when reading other people's loops, multiplied by the years of experience of the reader with that particular construct.

More seriously, we would like to point out that any concept for eager comprehensions, or loops, represents a trade-off between simplicity and flexibility. This follows from the fact that loops cover a large scale of complexity in programs, from simple repetition to complicated nested and parallel actions with several conditions in between and numerous invariants. In effect, designing "the" loop construct might not be the right goal to aim for, and it might also be necessary to predefine frequent idioms of loops. An import tool for flexibility is orthogonality—for example in SRFI 42 every generator can be modified by adding another termination condition.

While the orthogonality idea is strong in Scheme, the iterative part of it has been somewhat neglected. (More on that in Section 4.1.) Nevertheless, the looping constructs that *are* available in R^{5}RS are not too complicated to remember, i.e. mental complexity is relatively low. At the other extreme end, Common Lisp's `loop` might be found—highly flexible but also highly complicated. (Refer to Section 4.5.)

3.2 Interactive use vs. batch mode

Scheme can be used as an interactive system or for writing batch programs. Although these modes are just two extremes of an entire spectrum of human-computer interaction they are useful abstractions for evaluating designs. The two modes impose conflicting requirements: concise notation and flexibility is most important to interactivity, while robustness, efficiency, and readability are primary concerns for batch mode.

In the case of eager comprehensions, the key to efficiency is the use of typed state-based generators, i.e. programs that enumerate a sequence by modifying a local state (values of variables), the state being of statically known type (e.g. an integer counter). Note that this does not necessarily mean the state is updated by using `set!`, it could also mean that the state is updated by rebinding (as with tail-recursive procedures). If the state is represented in boxed data structures, or if each loop iteration requires dispatching, performance usually suffers. For this reason, most loop constructs for Scheme (or Lisp in general) concentrate on the batch mode, only. In SRFI 42, on the other hand, the requirements of interactive and batch mode are addressed by two different mechanisms (typed and dispatched generators) which can be mixed freely.

3.3 Modularity

Modularity for comprehensions means that new types of generators and new types of comprehensions can be added without modifying the already existing generators or comprehensions. For the sake of illustration, let us assume the new type "Fooziset" is yearning for comprehension.

In lazy comprehensions modularity is for free: adding a generator means writing a function returning a lazy list to be used on the right-hand side of the single binding and enumeration construct ("`<-`" in Haskell). Adding a comprehension means writing a function processing a lazy list, possibly constructed by a comprehension. In effect, the comprehension

```
foozi_of_list [x | x <- list_of_foozi s]
```

produces an element-wise copy of a Fooziset, whatever that actually means.

For eager comprehensions, on the other hand, modularity is a challenge. And what is more important, modularity is the key for creating an abstraction that goes beyond a mere idiom for frequent programs! Unfortunately, the importance of modularity for eager comprehensions has long been underestimated. Most designs make it either outright impossible to add new generators and comprehensions, or this is inconvenient and cumbersome. In effect, the users of the mechanism do not take the trouble of adding the comprehensions and generators they really require in the application—wasting a great opportunity for useful abstraction.

[4] We use the term 'state' here in an informal way, refering to the status of all bits that could possibly alter the future of an iteration. Later, in Section 3.5, we will clarify that a sequence of states may actually mean a sequence of binding environments.

scoping convention	*examples*
$[expr\|inner..outer]$	Magma
$[expr\|outer..inner]$	Haskell, Python, Erlang, (Mathematica), Swindle, ...
$[inner..outer\|expr]$	—
$[outer..inner\|expr]$	SRFI 42
$[[expr\|inner]\|..outer]$	Mathematica
$[outer\|..[inner\|expr]]$	Ruby, Perl, GAP

Table 2. Possible scoping conventions

For example, a library for number theory would include a generator enumerating the prime divisors of an integer, together with its multiplicity, because that is what is needed in many places. A library for graphs, on the other hand, would provide generators for enumerating the vertices of a graph, or the edges leaving a particular vertex. All this is only possible through modularity.

For the design of SRFI 42 modularity has always been one of the top priorities (right after efficiency), and the biggest challenge. The breakthrough came when I learned about the technique of using hygienic macros in continuation-passing style (CPS) [8]. This mechanism allows fully modular definition of eager generators, and it has prompted me to start the design again from scratch. The result will be explained in greater detail in Section 5.

3.4 Scope

Eager comprehensions are programming language constructs for writing loops. As such they include syntactic binding forms for the loop variables. Where there is binding, there is scope. This means a loop variable is visible to some parts of the program but not to others—irrespective of whether this scope is specified or not, or whether there are simple rules to remember it. We emphasize this trivial fact because a conscious design of the scope is another critical factor for useful eager comprehensions.

In order to be able to talk about scoping, a language is needed to represent different approaches. For this consider the following simplified view of a comprehension: a comprehension consists of an expression *expr* and zero or more nested qualifiers *inner*, ..., *outer*. If the qualifiers are generators, *inner* denotes the one spinning fastest and *outer* the one spinning slowest. Clearly, this terminology only makes sense if *inner* is in the scope of all bindings introduced by *outer*, and if *expr* is in the scope of both *inner* and *outer*. In other words, *expr*, *inner*, *outer* are pieces of code with a certain scoping relation (and control flow) with respect to one another. These pieces can then be composed into a comprehension syntax using '[', ']', and '|'. All possibilities, together with examples, are listed in Table 2. Some arguments are:

1. It is an advantage to have eager comprehensions mimic the notation of set comprehensions because it is widely known. Set comprehensions use the $[expr|qualifier^*]$ convention, where the nesting of the qualifiers is not fixed and must be deduced from the context. For simple comprehensions, this is no problem and the mathematical notation looks extremely familiar.
2. The most simple conventions nest scope in one direction, i.e. $[expr|inner..outer]$ or $[outer..inner|expr]$. In a syntactically impoverished language like Scheme this is particularly attractive.
3. More complicated comprehensions will increasingly look like explicitly nested loops (`do`, named-`let`), and possibly be mixed with them. In Scheme, bindings are always introduced *before* the body, so it is an advantage to have outer bindings appear first.

These contradicting preferences naturally lead to the most popular choice $[expr|outer..inner]$ because it looks like a set comprehension (1.) while introducing bindings left-to-right (3.); refer to Table 2.

For SRFI 42 linearity in scope was considered most important (2.), which together with Scheme's preference for left-to-right binding (3.) leads to $[outer..inner|expr]$. In effect, SRFI 42 sports an extremely simple scoping rule:

> *The bindings introduced by a generator are visible to all subsequent expressions (qualifier or other) of the same comprehension, and only to these*[5].

While in principle it would also be possible to have a compiler derive the nesting of the qualifiers from the dependency graph, this is a fundamentally bad idea. It would allow reordering the control flow by renaming variables, hashing readability in the process.

3.5 The meaning of state

As the Scheme language supports genuine state and destructive modification of data structures, it is important to clarify what is actually meant by 'iteration state.' More precisely, the designer of eager comprehensions needs to take position with respect to the following questions:

1. What is it supposed to mean if the payload of a generator retains (a reference to) an iteration variable, and uses it in later iterations or even outside the loop?
2. If the payload modifies an iteration variable?
3. If the payload modifies the loop-defining arguments or defining data structures while a loop is in progress?

Before considering possible approaches to these questions, recall that Scheme uses the following model of 'variable' [1, Section 3.1]:

> An identifier that names a location is called a variable and is said to be bound to that location. The set of all visible bindings in effect at some point in a program is known as the environment in effect at that point. The value stored in the location to which a variable is bound is called the variable's value.

Concerning the first semantic question, consider the following program (in SRFI 42 syntax):

```
((cadr (list-ec (:range n 3) (lambda () n))))
```

The result of this expression depends on how `:range` updates its loop variable: by rebinding or by state modification?

In the state modification model, the variable `n` is bound to a single location, and `set!` is used during the iteration to store the integer for that iteration. In effect, the three procedures in the list constructed by `list-ec` contain a reference to the *same* location—and the result of calling any of these procedures will be the state after the entire loop. So the result will be either `2` or `3`, depending on the way the loop modifies `n`. In this model, iteration enumerates a sequence of states stored in a given set of locations.

In the rebinding model, `n` is bound to a new location for every iteration. In this case, the three procedures each retain a different location, and the result is `1`. The rebinding model has been adopted for the iteration constructs of Scheme [1, Section 4.2.4], probably due to a desire for conceptual simplicity. Consequently, it is also the choice for SRFI 42. It should be mentioned that the overhead of rebinding is the same as for any other tail-recursive procedure, and these are supposed to be efficient in Scheme.

[5] As with everything in Scheme there is no way to enforce this, but SRFI 42 is built on this rule; users may have reason to deviate from this but it is not encouraged.

Concerning the second semantic question, consider the following program (again in SRFI 42 syntax):

```
(list-ec (:range n 3) (begin (set! n 2) n))
```

The result of this expression depends on whether `:range` uses the variable `n` itself to hold the state of the iteration (in which case the result is `'(2)`), or if `n` is just a copy of the (hidden) state of the iteration (in which case the result is `'(2 2 2)`).

In Scheme [1, Section 4.2.4], named-`let` and `do` provide access to the state of the iteration itself. This allows arbitrary modification of the state, which can sometimes simplify termination conditions. For eager comprehensions, however, the variables visible to the payload might not hold the state at all (e.g. `:list` hides the rest list still to be enumerated). Hence, for eager comprehensions only two approaches make sense: Either define that the variables visible are always a copy, or define the effect of assigning to a loop variable as unspecified. The latter approach was chosen for SRFI 42 in the name of efficiency.

Concerning the third semantic question, consider:

```
(let ((n 3))
  (list-ec (:range k n) (begin (set! n 2) k)) )
```

Here, the question is whether `:range` does access the variable `n` for every termination test, or just reads `n` once to set up the loop. Again, different solutions are possible, but the choice becomes easier once it is understood that `n` could be replaced by an arbitrary expression. If `:range` would evaluate its argument expressions repeatedly, this could unintentionally come at a hight price. For this reason, SRFI 42 specifies that the argument expressions of generators are evaluated exactly once: before the loop is set up.

Related to the question what happens if the loop-defining argument is modified is the question what happens if the loop-defining data structure is modified. As there is no way of enforcing anything in Scheme, and copying entire data structure (even if desired) could become costly, the result of modifying a data structure while it is being traversed is better defined unspecified.

3.6 Parallel loops

Often several loops must be executed in lockstep, e.g. counting the lines while reading a file. We will call this "parallel loops," but this does not mean that the processing steps are executed concurrently. Several mechanisms for comprehensions do support such a combination, for example Glasgow Haskell's extension of Haskell98's lazy comprehensions [17], Swindle [7], and SRFI 42 [2].

In the case of lazy comprehensions, parallel generators are relatively straight-forward. Since lazy comprehensions require exactly one type of generator (running through a lazy list), it is sufficient to provide "zipping" two or more lazy lists before enumerating them. In effect, the usefulness of parallel lazy generators is primarily determined by their notation.

Parallel eager generators, on the other hand, are a greater challenge. While the concept of eager comprehensions often allows the user to ignore the details of a loop (i.e. setup, iteration, and termination of the generator), parallel generators can only be constructed by interleaving the different parts of the component generators. Clearly, for this interleaving to be modular it is necessary that every generators is represented by some fixed pattern giving access to the code for setup, iteration, and termination.

In Scheme, the natural solution for this is representing a generator by a procedure computing the next element, and eventually indicating termination. The setup part of a generator constructs the procedure. This approach is used for example in Swindle and for the dispatching generator `(:)` of SRFI 42.

A different approach is to reduce each generator to some fixed "standard loop structure," which provides access to the individual parts of the generator. Then the parts can be combined syntactically for merging two or more component generators into a single parallel generator. This is exactly what the `:parallel` generator of SRFI 42 does, i.e. merging "fully decorated `:do`-loops" (Section 5.2).

3.7 Index variables

A frequent special case of a parallel loop is with an additional index variable, i.e. a variable running through $0, 1, \ldots$ while the elements of another sequence are enumerated. There are two ways of supporting this: by using `:parallel` for combining an unbounded integer counter (with generator `:integer`) with any other generator, and by adding an index variable to the other generator itself.

The first method is universally applicable to any generator, and as such fully modular. The second method provides a more concise notation (important for interactive use), and it can be a little more efficient in case the other generator uses an index anyhow (e.g. `:vector`). SRFI 42 supports both methods.

3.8 Early stopping

An important factor determining the flexibility of a looping construct is a facility for terminating generators or comprehensions early. This is a different mechanism than testing qualifiers (aka guards or filters). The difference is best illustrated by an example.

Consider a predicate for testing if a positive integer is the sum of its proper divisors:

```
(define (perfect? n)
  (= (sum-ec (:range d 1 n)
             (if (= (modulo n d) 0))
             d )
     n ))
```

The `if`-qualifier prevents the inclusion of non-divisors into the sum but it does not stop the `:range`-generator. Now we start investigating perfect numbers:

```
(first-ec #f (: n 1 100) (if (perfect? n)) n)
  ⇒ 6
```

This time the entire comprehension was finished after computing the first perfect number. But assume we need the numbers up to and including the first perfect number:

```
(list-ec (:until (: n 1 100) (perfect? n)) n)
  ⇒ '(1 2 3 4 5 6)
```

In this case the generator `(: n 1 100)` is modified to terminate *after* producing the element for which the additional condition `(perfect? n)` became true. (Note also that the scoping rule of SRFI 42 stated in Section 3.4 dictates that the condition comes *after* the generator in the `:until` expression.) Alternatively, the generator is to terminate *before* producing the element violating an additional condition.

Both forms of early-stopping generators are needed frequently. For example, consider reading a line of text by reading individual characters from a port. Since the last line may or may not have a trailing newline, it is important to append each character read to the string, including newline. This requires the use of `:until`:

```
(define (read-line port)
  (string-ec (:until (:port c port read-char)
                     (char=? c #\newline) )
             c ))
```

(In fact, this was the motivating example for including both `:while` and `:until` in SRFI 42.) The `:while` form of early termination is even more frequent since it derives directly from a precondition of the payload of a comprehension.

Coming back to `first-ec`, the two most useful and frequent early-stopping comprehensions test a predicate on a sequence of

values, stopping as soon as a violation is found. These comprehensions, named `any?-ec` and `every?-ec` in SRFI 42, can in fact be derived from `first-ec`.

3.9 Prefix vs. infix syntax

A trivial but highly visible matter is to what extent the syntax makes use of syntactic keywords in infix position (i.e. in a position not being the first after the opening parenthesis). Ultimately, this comes down to personal preference in the form of a compromise between simplicity and similarity with a natural language (which tends to be English). Most designs of comprehensions use an infix operator for the generators ('`<-`' is most popular) and possibly more infix operators for other qualifiers and options. This approach has the definitive advantage of reducing the number of parentheses.

In SRFI 42, on the contrary, no infix operators are used at all for the sake of (reducing) mental complexity. A comprehension defining *something* is probably named *something*`-ec`, and a generator defined by an object of type *type* is probably named `:`*type*. All generators are used in the syntax (`:`*type var arg**`)`, where *var* is a variable, optionally followed by an index variable *i* specified as `(index` *i*`)`.

For illustration, Table 3 shows expressions for the same nested loop in different programming languages supporting some form of comprehension. Keep in mind, though, that this is an extremely simple example where the meaning can be guessed at once. For more complicated expressions, infix notation, potentially even with precedences, adds to mental complexity.

4. Concrete designs

In this section we consider existing concrete designs for programming language constructs that enable or simplify (or obfuscate) loops in the Lisp-family of languages. Related constructs for other programming languages are beyond the scope of this article—but with the exception of lazy comprehensions, and loops with genuine parallel semantics as present in Erlang and Occam, they are also not very interesting.

The list does cover some loop-macros from other Lisp dialects, most notably Common Lisp, because these constructs represent serious efforts to provide what is called eager comprehensions in this article. It should be noted, however, that none of the Lisp looping constructs ever came to popularity in the Scheme community, unlike SRFI 42 which surprisingly has gathered quite some friends already in the first few years of its existence. (My earliest sketches date from late 2000; the SRFI got published in the beginning of 2003.)

4.1 `Lambda`, named-`let`, and `do` ($\mathrm{R^5RS}$)

In Scheme the most important construct for writing loops are recursive procedures, often in a tail recursive form. As $\mathrm{R^5RS}$ requires implementations to provide proper tail recursion [1, Section 3.5], recursion also serves as an idiom for iteration. A particularly convenient notation for defining and immediately executing recursive procedures is named-`let` [1, Section 4.2.4]. In addition, Scheme contains the `do`-syntax for defining a single loop, based on explicit state [1, Section 4.2.4].

This design represents a careful choice for including only a few clean and powerful constructs into the language, conforming to the overall minimalistic design philosophy of Scheme. Regrettably, there are two major shortcomings in practice. Firstly, it is already complicated to write the ubiquitous simple loops (refer to the example in the beginning). And secondly, the components of a loop (startup, iteration, termination) are often scattered over large amounts of source code—even if this would be unnecessary. Yet, maybe surprising, no other mechanism for writing loops has achieved considerable acceptance in the Scheme community, leaving the programmer to her own devices.

4.2 "Macros for writing loops" (Kelsey)

The "Macros for writing loops" library [4] is distributed with the Scheme 48 system [3] as the `reduce` package.

It provides the syntactic forms `iterate` and `reduce` implementing the fundamental state based eager comprehension. There are predefined generators running through lists, vectors, strings, integer ranges, reading from a port, and executing a generator procedure (called stream). Other generators can be added fully modularly by defining a hygienic macro in continuation-passing style (CPS) [4, Paragraph "Defining sequence types"]. The comprehensions (`iterate`, `reduce`) define a single, possibly parallel, loop based on explicit state modification.

"Macros for writing loops" is the probably first new loop construct to be proposed for a long time. Moreover, the implementation technique of CPS macros is the key to modularity of comprehensions. In effect, "Macros for writing loops" was most influential to the design of SRFI 42, even though the resulting mechanisms and notations bear little resemblance.

4.3 Swindle (Barzilay)

The Swindle library [7] is a collection of modules extending the PLT Scheme system [5]. It is written for and in PLT. The module "misc.ss" of Swindle contains macros for defining eager comprehensions in the sense of this article.

More precisely, there are predefined comprehensions for side-effect, making a list, numeric summation, numeric products, counting, and general reduction (`collect-of`). Generators are predefined for (integer) ranges, lists, vectors, strings, integers, executing generator procedures, and hash-tables. Swindle allows parallel execution of generators, early termination of comprehensions, has local bindings and side-effects. Generators can be added fully modularly using the generator procedure interface. Swindle makes extensive use of infix notation for expressing generators (e.g. `(n <- 0 .. 10)`, qualifiers, options, and other constructs (infix `and` for parallel execution).

The mechanisms specified in Swindle and for SRFI 42 are very closely related in their principles, but differ considerably in the details. Both acknowledge the need for modularity and well defined scope.

4.4 SRFI 40 "A library of streams" (Berwig)

Although the final form of SRFI 40 [9] does not contain comprehensions anymore, its draft versions did. These comprehensions were of course lazy. During the discussion of SRFI 40, it was decided to split the standard into a lower level part (which became the final SRFI 40) and a higher level part, including lazy comprehensions, which was to become SRFI 41.

The lazy comprehensions of SRFI 40 provided the same benefits as other lazy comprehensions, that is modularity and simplicity. The downside of lazy comprehensions in Scheme is a substantial loss in performance due to the overhead of constructing lazy streams correctly and reliably.

Recall that a lazy stream is something much more sophisticated than a generator procedure (accessing a state hidden in its closure). This implies that lazy comprehensions really require efficient non-strict evaluation, or strictness analysis. While these methods are being used in lazy languages, they are usually not available in Scheme because most programs do not require it.

4.5 Common Lisp

The Common Lisp language [10] contains several constructs for writing loops, and nested eager comprehensions in the sense of this

language	*example*
Haskell	`[k*k | n <- [0..9], k <- [0..n-1]]`
Python	`[k*k for n in range(10) for k in range(n)]`
Ruby	`(0..9).collect {|n| (0..n-1).collect {|k| k*k}}.flatten!`
Erlang	`[K*K || N <- lists:seq(0,9), N >= 1, K <- lists:seq(0,N-1)]`
Mathematica	`Join @@ Table[Table[k*k, {k, 0, n-1}], {n, 0, 9}]`
Magma	`[k*k : k in [0..n-1], n in [0..9]]`
GAP	`Concatenation(List([0..9], n -> List([0..n-1], k -> k*k)))`
PLT, Swindle	`(list-of (* k k) (n <- 0 ..< 10) (k <- 0 ..< n))`
R^5RS, SRFI 42	`(list-ec (: n 10) (: k n) (* k k))`, or with typed generators: `(list-ec (:range n 10) (:range k n) (* k k))`
Scheme48, reduce	`(reduce ((count* n 0 10)) ((r '())) (reduce ((count* k 0 n)) ((r r)) (cons (* k k) r) ) (reverse r) )`

Table 3. Examples of a simple nested loop.

article. These constructs include `do`/`do*`, `dotimes`, `dolist`, and `loop`.

`Do` is essentially the same as in Scheme, apart from the fact that Common Lisp also allows dynamic binding of variables (using `special`). `Do*` is a sequential-binding variant of `do`. `Dotimes` iterates over integer ranges, and `Dolist` over lists; these are rather specialized control structures.

The `loop` facility, on the other hand, could be interpreted as a general programming language in its own right (34 EBNF definitions, [10, Section 6.2 "LOOP"]). It is an extremely flexible mechanism for writing nested and parallel loops, possibly with early stopping, saving intermediate results, goto and labels, and several other features. Since it also supports various forms of accumulation of results, it should be seen as a syntactic form for eager comprehensions. These include comprehensions for making lists, appending, counting, max, min, summation, and general reduction. The syntax is mostly based on infix notation with syntactic keywords for clauses, options, and qualifiers.

The `loop`-syntax is one of the work horses of Common Lisp. It has evolved over a very long time towards higher and higher flexibility, often through the use of infix syntactic keywords. The mental complexity this has produced, however, is a big disadvantage in practice. In effect, the construct does not enjoy large popularity in the Scheme community.

4.6 Other iteration packages for Common Lisp

The "MIT LOOP" [35] is the predecessor of the Common Lisp `loop` facility. The "SLOOP package" (Schelter) [33] is an iteration facility generalizing MIT Lisp's `loop`. The "Yale LOOP Macro" (Ritter and Panagos) [34] is an implementation of the Yale `yloop` macro as described in [37]. All these loop facilities have in common that only the fundamental (side-effect) comprehension is implemented. The syntax is based on syntactic keywords in infix notation and the expressive power varies. Often new types of generators can be added, using the underlying macro facility (procedures as first class citizens did not exist in the language).

The "Series Macro Package" (Waters) [30, 31] implements a concept closely related to lazy comprehensions in the sense of this article. A "series" is essentially a data structure for a lazy list. The package contains operations for producing, processing, and consuming these data structures, or acting on their elements. The implementation is often able to transform the lazy operations into eager evaluation, producing efficient code for frequent loop structures.

The "Lisp comprehensions" (Lapalme) [32] is an adaptation of lazy comprehensions from Miranda into Common Lisp. In this work, Wadler's transformation of lazy list comprehensions [18, Chapter 7] is translated one to one into Lisp in order to mimic the (infix) notation of lazy list comprehensions in Miranda. As the essential conceptual difference between lazy and eager comprehensions is ignored, the resulting mechanism is only of limited usefulness in practice.

4.7 "The anatomy of a loop" (Shivers)

Recently, Shivers defined a new loop mechanism [22, 23, 24] for Scheme (in fact more generally), underpinned by a theory based on the notion of "control dominance." In a nutshell, control dominance is the static property that every access to a variable occurs within an explicit binding construct for that variable. This can be enforced by a type system restricting the control flow graph of the program.

In practice, this concept comes down to the following: all loops are reduced to a primitive loop template consisting of 8 parts, with the control flow graph being made explicit. On top of this resides a programming language very much in the style of a `loop`-macro with predefined generators, guards, and accumulators for the most common data structures. The single outer macro (named `loop`) can be seen as the fundamental eager comprehension, the 8-part loop as the fundamental eager generator (corresponding to `do-ec` and `:do` in SRFI 42).

Since the control flow is made explicit in Shiver's proposal, the looping construct is extremely flexible. However, at present it is not known whether it is also inherently more powerful than the mechanism defined in SRFI 42, or essentially equivalent. This question comes down to whether the fundamental generators (8-part loop vs. `:do`) can be expressed in terms of each other. In addition, it is too early to judge if the additional flexibility is worth the associated mental complexity (8-part loop defined by an explicit control flow graph), and what the impact of the minor design decisions (e.g. infix notation) is on usability. Either way, Shivers' work has potential for further clarifying the true nature of iteration in functional programs.

4.8 SRFI 42 "Eager comprehensions"

The term "eager comprehension" was coined for SRFI 42 [2] in order to make sure the mechanism is never confused with the well-known lazy comprehensions. The reference implementation associated with SRFI 42 is portable under R^5RS with hygienic macros. As the SRFI found some acceptance in the community, implementations are included into several Scheme systems, including PLT [5] and Scheme 48 [3].

The SRFI specifies an extensive set of predefined comprehensions based on what makes sense in R^5RS. Some infrequent com-

prehensions are left out (e.g. `gcd-ec`), while others have been added for convenience (e.g. `any?-ec`). The predefined typed generators enumerate the standard data structures $\mathrm{R}^5\mathrm{RS}$. In addition, a dispatching generator ("`:`", read "run through") selects a generator based on the type of arguments given, e.g. the range $\{0, \ldots, n-1\}$ when given an exact integer n. Generators can be run in parallel and terminated early. Other qualifiers include tests (guards), local bindings, and side-effects.

The syntax is based on a simple naming convention and prefix notation without exception. The uniform and simple scoping rule "scope extends to the right until the enclosing comprehension ends" is used (Section 3.4). Generators can be added fully modularly by defining a (hygienic) macro using continuation-passing style (CPS), or by providing a suitable generator procedure. Comprehensions can be added as (hygienic) macros. An introduction to SRFI 42 from the perspective of a user, together with some examples, is provided in the appendix.

5. The implementation of SRFI 42

In this section the overall structure of the reference implementation for eager comprehensions in Scheme is explained. The reader is assumed familiar with the specification as laid down in SRFI 42 [2]. Moreover, it is assumed that the reader is familiar with Scheme's hygienic macro facility [1, Sections 4.3, 5.3, 7.1.5], because it is the primary tool for the reference implementation of SRFI 42.

5.1 A skeleton of eager comprehensions

The following is a simplified but self-contained ($\mathrm{R}^5\mathrm{RS}$) working skeleton of eager comprehensions:

```
(define-syntax do-ec
  (syntax-rules (if :do)
    ((do-ec q1 q2 r1 r ...)
     (do-ec q1 (do-ec q2 r1 r ...)) )
    ((do-ec (if test) cmd)
     (if test cmd) )
    ((do-ec (:do lbs ne? lss) cmd)
     (do-ec:do cmd (:do lbs ne? lss)) )
    ; call g in CPS, reentry at (*)
    ((do-ec (g arg1 arg ...) cmd)
     (g (do-ec:do cmd) arg1 arg ...) )))

(define-syntax do-ec:do
  (syntax-rules (:do) ; reentry point (*)
    ((do-ec:do cmd (:do (lb ...) ne? (ls ...)))
       (let loop (lb ...)
          (if ne?
              (begin cmd
                     (loop ls ...) ))))))

(define-syntax :do
  (syntax-rules ()
    ((:do (cc ...) lbs ne? lss)
     (cc ... (:do lbs ne? lss)) )))
```

This code defines the primitive eager comprehension `do-ec` and the primitive eager generator `:do`, utilizing a helper macro `do-ec:do` for generating code for `:do`.

Other generators can now be added without modifying the existing macros. E.g. after defining

```
(define-syntax :range
  (syntax-rules ()
    ((:range cc var n)
     (:do cc ((var 0)) (< var n) ((+ var 1))) )))
```

the following comprehension is operational:

```
(do-ec (:range n 5) (:range k n) (display k))
  ⇒ prints: 0010120123
```

The critical issue is the flexibility of the generator `:do` to which all other generators are being reduced. In the skeleton above (refer to `do-ec:do`), the generator `:do` can produce a single named `let` with an arbitrary number of variables (`lb ...`) and a single `if` guarding payload (`cmd`) and next iteration.

5.2 Fully decorated `:do`

In practice, the simple loop structure of the previous section is too restricted. In particular it is not possible to derive the variables visible to the payload from other state variables, to pre-process the arguments, or to terminate after executing the payload. On the other hand, complexity must be kept down.

The particular trade-off chosen for SRFI 42 is based on a fair amount of experimentation. It turned out that the following structure ("fully decorated `:do`") covers most relevant generators:

```
(let (outer-binding ...)
  outer-command ...
  (let loop (loop-binding ...)
    (if not-end-1?
        (let (inner-binding ...)
          inner-command ...
          ≪payload≫
          (if not-end-2?
              (loop loop-step ...) )))))
```

The `:do` generator specifies all variable parts, except for ≪*payload*≫ of course. It allows termination of the loop before or after the payload has been executed. Since many generators do not require "full decoration," a simple transforming optimizer simplifies boolean conditions, eliminates redundant `if` and `let`, and turns `let` without bindings into `begin`.

Note that the use of named-`let` allows iteration by rebinding (Section 3.5), using *loop-binding* and *inner-binding*. Updating by state modification is also possible by storing the iteration state in *outer-binding*, and modifying it using `set!` within `loop`. In fact, `:do` is the only generator in SRFI 42 that allows updating by state modification because no other generator passes the names of this variables in *outer-binding* to its ≪*payload*≫.

The chosen structure for `:do` is powerful enough, and yet still restricted enough, to support the following important constructions on generators:

- Any generator can be modified to terminate early, based on some additional condition, either before (`:while`[6]) or after (`:until`) the payload is executed.
- Two or more `:do`-generators can be merged into a single generator (`:parallel`) enumerating all sequences simultaneously.

For the sake of illustration, here is the complete implementation of the generator `:list` in SRFI 42 running a variable `var` through the concatenation of one or more lists, possibly with an additional index variable `i`.

```
(define-syntax :list
  (syntax-rules (index)
    ((:list cc var (index i) arg ...)
     (:parallel cc (:list var arg ...)
                   (:integers i)) )
    ((:list cc var arg1 arg2 arg ...)
     (:list cc var (append arg1 arg2 arg ...)) )
    ((:list cc var arg)
     (:do cc (let ())
             ((t arg))
             (not (null? t))
```

[6] The implementation is complicated by the fact that the scopes of the variables bound must be preserved while adding the termination condition. This means it is *not* sufficient to add a condition to *not-end-1?*.

```
(let ((var (car t))))
#t
((cdr t)) ))))
```

The generator `:integers` runs through the infinite sequence of non-negative integers. The expressions supplied to `:do` correspond to the "fully decorated" structure given above, i.e. `(t arg)` is the *loop-binding* and `(var (car t))` is the *inner-binding*.

Note that the multiple-argument case cannot easily be converted into a nested loop because `:do` can only produce a *single* loop; nested loops would prevent generator-merging.

5.3 The dispatching generator

As an alternative to typed generators (`:range`, `:list` etc.) the dispatching generator `:` (read 'run through') of SRFI 42 first evaluates its argument expressions and then dispatches on the type of the values. In other words, `:` is a polymorphic generator. For example, `(list-ec (: x 3) x)` produces `'(0 1 2)` and `(list-ec (: x "abc") x)` produces `'(#\a #\b #\c)`. The purpose of the dispatching generator is making interactive use of comprehensions more convenient.

The implementation of `:` evaluates the arguments and calls a global dispatching procedure. The dispatcher is to construct a generator procedure which is then run to enumerate the sequence. A generator procedure g has a single argument. When called, g either returns the next value of the sequence, or, when the sequence ran out, it returns its argument. In the implementation, the argument given to a generator procedure is `(list #f)`, i.e. an object only `eq?` to itself.

For the sake of modularity, the dispatcher procedure can be retrieved and changed. Moreover, there is a macro producing a generator procedure from a typed generator; this greatly simplifies the definition of dispatching generators.

5.4 Grouping qualifiers with `nested`

In addition to defining new generators in a modular way it is also important to define new comprehensions. While in principle there is no problem (after all every eager comprehension can be reduced to `do-ec`), the fact that there can be an arbitrary number of qualifiers complicates the definition of new comprehensions. In the worst case, a variation of `do-ec` must be provided every time.

A simple trick being used in SRFI 42 keeps the amount of code for a new comprehension low. The syntactic keyword `nested` can be used for grouping an arbitrary number of qualifiers into a single equivalent qualifier understood by `do-ec`. This is illustrated by the definition of a folding comprehension:

```
(define-syntax fold-ec
  (syntax-rules (nested)
    ((fold-ec x0 (nested q1 ...) q r1 r2 r ...)
     (fold-ec x0 (nested q1 ... q) r1 r2 r ...) )
    ((fold-ec x0 q1 q2 r1 r2 r ...)
     (fold-ec x0 (nested q1 q2) r1 r2 r ...) )
    ((fold-ec x0 expr f)
     (fold-ec x0 (nested) expr f) )

    ((fold-ec x0 qualifier expr f)
     (let ((result x0))
       (do-ec qualifier
              (set! result (f expr result)))
       result ))))
```

The last case of the macro implements the functionality for the case that there is exactly one qualifier; the other cases of the macro collect all qualifiers into a single one. Now the list comprehension can be defined as

```
(define-syntax list-ec
  (syntax-rules ()
    ((list-ec r1 r ...)
     (reverse (fold-ec '() r1 r ... cons)) )))
```

Alternatively, the list could be `set-cdr!`'ed together, which may be faster (or not).

5.5 Early-stopping comprehensions

The early-stopping comprehensions of SRFI 42, that is `any?-ec` and `every?-ec`, are reduced to the fundamental early-stopping comprehension `first-ec` with the syntax

`(first-ec` *default qualifier** *expr*`)`.

This comprehension evaluates the sequence of values specified by the qualifiers, stopping after the first value of *expr*. If the sequence is found empty, the result is *default*.

`Call-with-current-continuation` could be used for a non-local exit, but the reference implementation does not. With an eye on performance it is implemented by introducing an additional stopping variable and modifying each generator to stop once this variable is found true (which is made happen when control reaches *expr*).

6. Performance

The top priority for eager comprehensions is combining convenience and performance. In this section, the performance aspect is investigated more quantitatively.

The Sieve of Eratosthenes As an example we consider computing the primes in $\{2, \ldots, n-1\}$, $n \geq 0$, by the algorithm known as the "Sieve of Eratosthenes." The algorithm (200 BC) ticks off all true multiples of the next not yet ticked off number—and the primes are left over. The following program represents the ticks in a string[7], and uses SRFI 42 for the loops.

```
(define (primes n)
  (let ((p (make-string n #\1)))
    (do-ec (:range k 2 n)
           (if (char=? (string-ref p k) #\1))
           (:range i (* 2 k) n k)
           (string-set! p i #\0) )
    (list-ec (:range k 2 n)
             (if (char=? (string-ref p k) #\1))
             k )))
```

This program is compared with three alternatives:

- The typed generators `:range` are replaced by the dispatching generator `:` of SRFI 42.
- The comprehensions are implemented in Swindle.
- The `do-ec` is replaced by two nested `do`-loops, and the `list-ec` is replaced by a tail-recursive named-`let` constructing the result list.

Figure 1 shows the execution time, divided by n. A number of things can be observed.

Firstly, all four alternatives have reasonable performance and are able to compute the primes below 10^6 in less than $10\,s$. Secondly, only the "DO loop" variant shows the slow increase expected for this $\Theta(n \ln \ln n)$-algorithm. The other curves exhibit lower order terms, probably due to the overhead of setting up a loop—which is most pronounced for the procedure-based variants ("SRFI 42 (:)" and "Swindle").

Linear model of execution time The preceeding example is based on a meaningful algorithm, which is important for a realistic impression. Now we turn to synthetic algorithms with the goal of

[7] A wasteful but practical alternative to arrays of bits, which are absent in Scheme itself and its portable libraries.

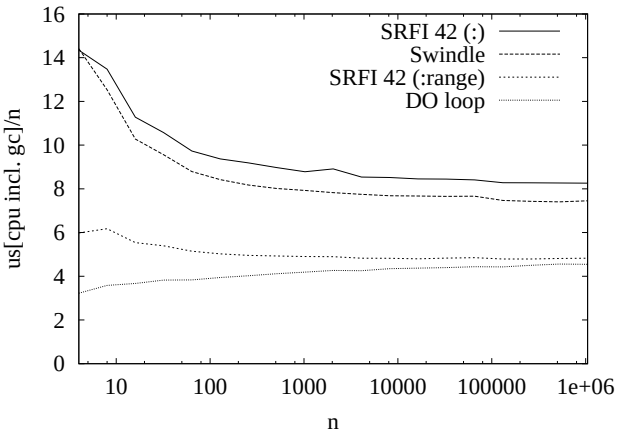


Figure 1. The "Sieve of Eratosthenes." MzScheme 208, Intel Pentium III Mobile, 1 GHz, Win2k.

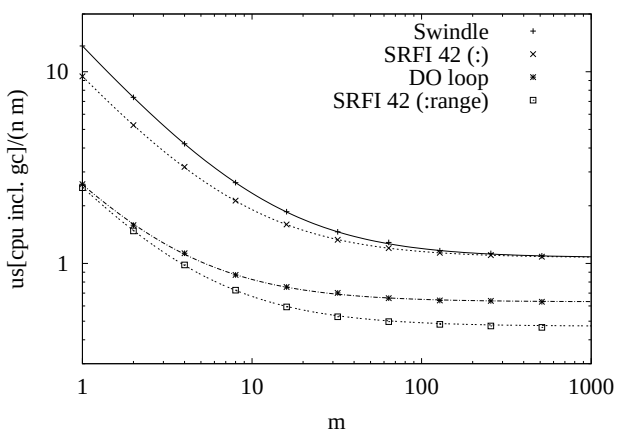


Figure 2. Two nested loops (n times outer, m times inner, $nm = 2^{24}$). MzScheme 208, Intel Pentium III Mobile, 1 GHz, Win2k.

obtaining quantitative information using an abstract model of the execution time.

It is reasonable to assume that the overhead of a loop grows according to a linear model consisting of a fixed startup overhead t_0 and some constant overhead Δt per iteration. The objective is to determine t_0 and Δt from measured execution times. For this we execute different implementations of the following nested loop:

`for` $k = 1..n$ `do for` $i = 1..m$ `do` *payload* `od od`,

where n and m integer parameters. In order to observe both startup- and iteration-overhead, the number m of inner iterations is varied, while fixing nm for obtaining sufficient total time. The data points in Figure 2 show the result.

Ignoring the time spent on the inner payload, startup- and iteration-overhead can readily be read off the curves as their start and end value. By fitting

$$t(n, m) = (1 + n)t_0 + (n + nm)\Delta t$$

to the data points in Figure 2, slightly more accurate estimates are obtained:

	$t_0/\mu s$	$\Delta t/\mu s$
Swindle	9.99	1.24
SRFI 42 (`:`)	6.59	1.21
DO loop	1.36	1.15
SRFI 42 (`:range`)	1.38	0.60

The curves associated with these parameters are shown in Figure 2.

The particular values obtained here should be taken as an indication, only. They are heavily dependent on the execution model of the underlying Scheme system (interpreted, byte-code, or native). Nevertheless, there is a remarkable gap between the eager comprehensions based on procedures and on direct state modification ("SRFI 42 (`:range`)"). As a rule of thumb, procedures cost a factor of two per iteration and five to ten in startup. We expect this gap to widen for Scheme systems with more sophisticated compilation but did not investigate this quantitatively.

7. Eager comprehensions "lazified"

For what it is worth, eager comprehensions can be turned lazy in a fully modular way. More precisely, it is possible to define the fundamental lazy list comprehension (`stream-ec` that is) in a such way that any *eager* generator can be used with it—without modifying the macros for the generators. Conversely, the eager generator `:stream` enumerates a lazy stream, i.e. runs a variable through the elements. For the streams we use SRFI 40 [9], which provides (even) lazy lists called "streams" as new data structures. With modification, it would also be possible to use simpler odd streams, for example those presented in [36].

The comprehension expression

`(stream-ec` *qualifier** *expr*`)`

constructs a stream for the sequence that a corresponding `list-ec` would create. The use of `stream-ec` is best explained by example:

```
(define s
  (stream-ec (: x 10) (begin (display x) x)) )

(stream-null? s)
   ⇒ [prints: 0] #f

(stream-null? (stream-cdr s))
   ⇒ [prints: 1] #f

(list-ec (:stream x s) x)
   ⇒ [prints: 23456789] '(0 1 .. 9)
```

In other words, the payload expression `(begin (display x) x)` is to be evaluated on demand, resulting in the digits being printed as shown.

It is an impressive illustration of the powerful mechanisms available in Scheme that `stream-ec` can in fact be defined in a modular way. A possible implementation:

```
(define-syntax stream-ec
  (syntax-rules (nested)
    ((stream-ec (nested q1 ...) q etc1 etc ...)
     (stream-ec (nested q1 ... q) etc1 etc ...) )
    ((stream-ec q1 q2             etc1 etc ...)
     (stream-ec (nested q1 q2)    etc1 etc ...) )
    ((stream-ec expression)
     (stream-ec (nested) expression) )
    ((stream-ec qualifier expression)
     (let ((value         #f)
           (produce-value #f)
           (next-value    #f))
       (define (tail)
         (stream-delay
          (if (call-with-current-continuation
               (lambda (cc)
                 (set! produce-value cc)
                 (next-value #f)
```

```
          #f ))
        (stream-cons value (tail))
        stream-null )))
(define (make-stream)
  (stream-delay
   (if (call-with-current-continuation
        (lambda (cc)
          (set! produce-value cc)
          (do-ec
            qualifier
            (call-with-current-continuation
             (lambda (cc)
               (set! next-value cc)
               (set! value expression)
               (produce-value #t) )))
          (produce-value #f) ))
       (stream-cons value (tail))
       stream-null )))
(make-stream) ))))
```

The macro combines all qualifiers into a single one using `nested` (Section 5.4) and uses the fundamental eager comprehension `do-ec` for enumerating the sequence defined by the qualifiers. `Call-with-current-continuation` is used to exit `do-ec` non-locally after producing a value and possibly resuming the very same loop again later.

The eager generator exhausting a stream can be defined as follows:

```
(define-syntax :stream
  (syntax-rules ()
    ((:stream cc var arg)
     (:do cc (let ())
          ((s arg))
          (not (stream-null? s))
          (let ((var (stream-car s))))
          #t
          ((stream-cdr t)) ))))
```

Since `:stream` is just another generator, it can of course be used in `stream-ec`—where it is executed lazily. And since `do-ec` understands guards and local definitions, we have implemented all there is to implement for *lazy* comprehensions in Scheme.

The bad news is `call-with-current-continuation` and the streams of SRFI 40 have a rather high price in terms of time and space consumption in most major Scheme systems. For this reason, the lazy comprehensions defined in this section should not be understood as a serious proposal for a programming language construct—but rather as of great educational and entertaining value. It should be emphasized, though, that lazy comprehensions can be very efficient, provided they are compiled properly.

8. Conclusions

Comprehensions are a particularly concise notation for writing nested and parallel loops with accumulation of results. In the past few years they have come to popularity in many programming languages, including Python and Erlang. When used wisely, comprehensions can improve readability, modularity, and possibly performance.

However, unlike the lazy list comprehensions (ZF expressions) of call-by-need functional languages (like Haskell), a corresponding concept in a call-by-value setting (like Scheme) has substantially different requirements in order to qualify for a generally useful programming construct. SRFI 42 is a specific design aiming at this goal. It is an impressive demonstration of Scheme's own flexibility that that the mechanism specified in SRFI 42 can be implemented naturally without extending the language itself.

Acknowledgements Mike Sperber has provided important input for eager comprehensions in Scheme, in particular he pointed me to the idea of "CPS macros." Without the discussions with Mike, SRFI 42 would probably not exist. Also I would like to thank Phil Berwig, the author of SRFI 40 ('A library of streams') for useful discussions on his lazy comprehensions for Scheme. Probably the biggest source of inspiration for my work presented here were Richard Kelsey's "Macros for writing loops"—even though the casual reader might not suspect this. I would like to thank Philips Research for making this work possible, and in particular my colleagues Philippe Coucaud, Zbigniew Chamski, and Kero van Gelder for their valuable remarks. Finally, I would like to thank the anonymous referees for their corrections and discussion. In particular the third referee brought up the important issue of semantics, and provided an example exposing the 'update by rebind vs. by side-effect' choice.

References

[1] R. Kelsey, W. Clinger, and J. Rees (eds.): Revised^5 Report on the Algorithmic Language Scheme. 20 February 1998. `www.schemers.org/Documents/Standards/R5RS`

[2] S. Egner: SRFI 42 "Eager Comprehensions". Finalized July 7, 2003. `srfi.schemers.org/srfi-42`

[3] R. Kelsey and J. Rees: The Scheme 48 System. `s48.org`

[4] R. Kelsey and J. Rees: "Macros for Writing Loops." The `reduce` library of Scheme 48 [3]. `s48.org/1.2/manual/s48manual_53.html`

[5] The PLT Team: PLT Scheme. `www.plt-scheme.org`

[6] PLT MzScheme. `www.plt-scheme.org/software/mzscheme`

[7] E. Barzilay: The Swindle Library for PLT Scheme [5]. The `collect` macro of the module "`misc.ss`." `www.cs.cornell.edu/eli/Swindle/misc-doc.html#collect`

[8] E. Hilsdale, D. P. Friedman: Writing Macros in Continuation-Passing Style. Scheme and Functional Programming 2000. September 2000.

[9] P. L. Bewig: SRFI 40 "A Library of Streams." Finalized August 22, 2004. `srfi.schemers.org/srfi-40`

[10] LispWorks Ltd.: The Common Lisp HyperSpec (1996–2005), Chapter 6 "Iteration." `www.lispworks.com/documentation/HyperSpec/Body/06_.htm`

[11] G. van Rossum: Python Reference Manual, Release 2.4.1, 30 March 2005. Section 5.2.4 "List Displays". `www.python.org/doc/2.4.1/ref/lists.html`

[12] Wolfram Research: Mathematica Version 5.0, Documentation of `Table`. `documents.wolfram.com/mathematica/functions/Table`

[13] W. Bosma, J. Cannon: Magma (V2.11, May 2004) Documentation of "Sets" and "Sequences". `magma.maths.usyd.edu.au/magma/htmlhelp/part2.htm`

[14] Ericsson AB: Erlang, Reference Manual (Version 5.4.3). Section 6.22 "List Comprehensions." `www.erlang.se/doc/doc-5.4.3/doc/reference_manual/expressions.html#6.22`

[15] Martin Schönert et. al.: GAP—Groups, Algorithms, and Programming, (Version 3 Release 4 Patchlevel 4) Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.

[16] S. L. Peyton Jones (ed.): Haskell 98 Language and Libraries, The Revised Report, December 2002. Section 3.11 "List Comprehensions." `www.haskell.org/onlinereport/exps.html`

[17] The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4. Section 7.3.4. "Parallel List Comprehensions." `www.haskell.org/ghc/docs/latest/html/users_guide/syntax-extns.html#parallel-list-comprehensions`

[18] S. L. Peyton Jones: The Implementation of Functional Programming Languages. In particular, Chapter 7 "List Comprehensions" (Philip Wadler). Prentice-Hall, Hemel Hempstead, 1987.

[19] R. B. K. Dewar: The SETL Programming Language. 1979.

[20] Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., and Schonberg, E.: Programming with Sets: An Introduction to SETL. Springer-Verlag,

New York, 1986.

[21] R. K. Dybvig: The Scheme Programming Language, 3rd edition. MIT Press 2003. Section 9.3. "A Set Constructor." www.scheme.com/tspl3/examples.html#./examples:h3

[22] O. Shivers: "The Anatomy of a Loop: a Story of Scope and Control." Presentation given at *Daniel P. Friedman: A Celebration* (Bloomington (IN), December 3, 2004). www.cs.indiana.edu/dfried_celebration.html

[23] O. Shivers: "The Anatomy of a Loop: a Story of Scope and Control." Presentation given at Laboratoire d'Informatique de Paris 6 (Paris, January 24, 2005). www.lip6.fr/fr/liens/organise-fiche.php?theme=5&RECORD_KEY(organise)=id&id(organise)=98

[24] O. Shivers: "The Anatomy of a Loop: a Story of Scope and Control." To be published at ICFP 2005, Tallinn, Estonia.

[25] R.M. Burstall: Design Considerations for a Functional Programming Language. Infotech State of the Art Conference: The Software Revolution, Copenhagen, October, 1977.

[26] R. M. Burstall, D. B. MacQueen, and D. T. Sannella: Hope: An Experimental Applicative Language (1980). Conference on LISP and Functional Programming archive Proceedings of the 1980 ACM Conference on LISP and Functional Programming, pp. 136–143, Stanford University, California, United States.

[27] D.A. Turner: The Semantic Elegance of Applicative Languages, in Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture 1981, Portsmouth, New Hampshire, USA.

[28] D.A. Turner: Miranda: A Non-strict Functional Language with Polymorphic Type. Proceedings of a Conference on Functional Programming Languages and Computer Architecture, pp. 1–16, Nancy, France, 1985.

[29] D.A. Turner: An Overview of Miranda. ACM SIGPLAN Notices, Volume 21, Issue 12, December 1986.

[30] R. C. Waters: The Series Macro Package. ACM SIGPLAN Lisp Pointers, Volume III, Issue 1, July 1989.

[31] R. C. Waters: The Series Macro Package for Common Lisp. series.sourceforge.net

[32] G. Lapalme: Implementation of a "Lisp Comprehension" Macro. ACM SIGPLAN Lisp Pointers, Volume IV, Issue 2, April 1991.

[33] W. Schelter: The SLOOP Iteration Facility (1985). www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/iter/loop/sloop/0.html

[34] F. Ritter, J. Panagos: YLOOP: Portable Implementation of the Yale LOOP Macro (1986). www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/iter/loop/yloop/0.html

[35] Massachusetts Institute of Technology: The MIT LOOP Macro (1980, 1986) www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/iter/loop/mit/mit_loop.cl

[36] H. Abelson, G. J.Sussman, J. Sussman: Structure and Interpretation of Computer Programs. 2nd ed., MIT Press, Cambridge (MA), 1996.

[37] E. Charniak, C. K. Riesbeck, D. McDermott, and J. R. Meehan: Artificial Intelligence Programming, 2nd ed. Lawrence Erlbaum Associates, 1987.

Appendix: Summary of SRFI 42

For illustration, this appendix contains a brief introduction to SRFI 42 from a user's perspective, together with examples. The actual specification is available at

`http://srfi.schemers.org/srfi-42/srfi-42.html`

In its most simple form, a comprehension according to SRFI 42 looks like this (its value after `=>`):

```
(list-ec (: i 5) (* i i)) => '(0 1 4 9 16).
```

Here, `i` is a local variable sequentially having the values $0, 1, \ldots, 4$, and the squares of these numbers are collected in a list, which is the result. The following example illustrates most conventions of SRFI 42 with respect to nesting and syntax:

```
(list-ec (: n 1 4) (: i n) (list n i))
  => '((1 0) (2 0) (2 1) (3 0) (3 1) (3 2)).
```

In this example, the variable `n` first has value 1 then 2 and finally 3. For each value of `n`, the variable `i` assumes the values $0, 1, \ldots,$ `n` $-$ 1 in turn. The expression `(list n i)` constructs a two-element list for each binding, and the comprehension `list-ec` collects all these results in a list.

Eager comprehensions in the sense of SRFI 42 are just hygienic macros. The basic syntactic form of a comprehension is

`(do-ec` *qualifier** *command*`)`,

i.e. zero or more *qualifier* arguments and a *command* argument. The `do-ec` comprehension enumerates the sequence of binding environments specified by the qualifiers and for each such environment evaluates *command* for side-effects. In a similar fashion, `(sum-ec` *qualifier** *expression*`)` sums the values obtained by evaluating *expression* for the sequence of binding environments specified by *qualifier**. If *qualifier** is empty (i.e. no qualifiers at all) then *expression* is evaluated once. The eager comprehension `list-ec` constructs a list of the values of its expression.

The most common qualifiers are generators. For example, `(:range i 5)` runs variable `i` through $0, 1, \ldots, 4$. The generator `(: i 5)` does the same but uses the type of its argument (i.e. `5`) to decide that it is a range of exact integers that is to be enumerated. In every iteration, `i` is bound to a new location where the integer for that iteration is stored. Other qualifiers are for filtering, e.g. `(if` *condition*`)`, or for side-effect, e.g. `(begin` *command*`)`. The full syntax of SRFI 42 is listed with comments in Table 4.

Checklist for adding comprehensions and generators

The following checklists can if the user wants to add application-specific comprehensions and generators in the style of SRFI 42.

For adding an application-specific comprehension:

1. Use the syntax (*accu*`-ec` $\ll$*outer*$\gg$ *qualifier** $\ll$*inner*$\gg$), with $\ll$*outer*$\gg$ being a fixed list of parameter expressions (e.g. for default values), $\ll$*inner*$\gg$ being a fixed list inner expressions (usually just *expression*), and *accu* refering to the accumulation process that is being executed.
2. Use the left-to-right scoping rule as much as possible.
3. Avoid syntactic keywords, in particular in infix position.
4. Evaluate parameter expressions exactly once, or at most once if their evaluation is control-flow dependent. Implement this by inserting `let`.
5. Make sure the implementation does not copy macro arguments, because that might lead to exponential growth in code size when nested.

For adding an application-specific typed generator:

expression → *comprehension* | ...	
comprehension →	
(*ordinary-ec qualifier** *expression*)	evaluate *expression* for the sequence of binding environments (or states) specified by the qualifiers
| (`vector-of-length-ec` *k qualifier** *expression*)	`vector-length` of result known to be *k*
| (`fold-ec` x_0 *qualifier** *expression* f_2)	$f_2(x_n, f_2(x_{n-1}, \cdots f_2(x_1, x_0) \cdots))$ for $x_{1..n}$ from *expression*
| (`fold3-ec` x_0 *qualifier** *expression* f_1 f_2)	$f_2(x_n, f_2(x_{n-1}, \cdots f_2(x_2, f_1(x_1)) \cdots))$, or x_0 if $n = 0$
| (`do-ec` *qualifier** *command*)	evaluate *command* for side-effect
| *application-specific-comprehension*	define using hygienic macro, use checklist
ordinary-ec →	
`list-ec` | `append-ec` | `string-ec` | `string-append-ec`	
| `vector-ec` | `sum-ec` | `product-ec` | `min-ec` | `max-ec`	
| `any?-ec` | `every?-ec` | `first-ec` | `last-ec`	early stopping (aka short evaluation)
qualifier →	
generator	
| (`if` *expression*)	insert test (aka guard or filter)
| (`not` *expression*) | (`and` *expression**) | (`or` *expression**)	abbreviate (`if (not` *expression*)) etc.
| (`begin` *command** *expression*)	insert side-effect
| (`nested` *qualifier**)	syntactic grouping of qualifiers
generator →	
(`:` *variables expression*$^+$)	dispatch on type (list,string,vector,integer,real,char,port)
| (`:list` *variables expression*$^+$)	elements of a (proper) list
| (`:string` *variables expression*$^+$)	characters of a string
| (`:vector` *variables expression*$^+$)	elements of a vector
| (`:integers` *variables*)	the infinite sequence $0, 1, \ldots$
| (`:range` *variables range-limits*)	exact integer range
| (`:real-range` *variables range-limits*)	real (either all exact, or all inexact) range
| (`:char-range` *variables min max*)	character range up to and including *max*
| (`:port` *variables expression* [*read*])	*read* defaults to `read`
| (`:dispatched` *variables dispatch expression*$^+$)	calls *dispatch* to construct generator procedure to run
| (`:let` *variables expression*)	single value sequence (for introducing intermediate variable)
| (`:parallel` *generator**)	interleaved execution, until one a generators is exhausted
| (`:while` *generator expression*)	execute *generator* while *expression* is non-#f
| (`:until` *generator expression*)	execute *generator* until (and incl.) *expression* is non-#f
| (`:do` [(`let` (*ob**) *oc**)] (*lb**) *ne1?* [(`let` (*ib**) *ic**) *ne2?*] (*ls**))	loop by named-`let`, possibly decorated
| *application-specific-typed-generator*	define as hygienic macro in CPS, use checklist
range-limits → *stop* | *start stop* | *start stop step*	from *start* (default `0`) to *stop* (excl.) by *step* (default `1`)
variables → *identifier* [(`index` *identifier*)]	index variable runs through $0, 1, \ldots$
x_0, f_1, f_2 *min, max, read, dispatch, start, stop, step* → *expression*	

Table 4. Syntax of SRFI 42.

1. Use the syntax (`:`*type var* [(`index` *i*)] ≪*args*≫), with ≪*args*≫ being the argument expression(s) defining the loop. Here *type* indicates the type of object to enumerate through.
2. Use the syntax (`:`*type* $var_1 \cdots var_n$ [(`index` *i*)] ≪*args*≫) if there are always exactly n variables to iterate through.
3. Use the syntax (`:`*type* (*var**) [(`index` *i*)] ≪*args*≫) if there is a variable number of variables to iterate through.
4. Use the left-to-right scoping rule as much as possible.
5. Avoid syntactic keywords, in particular in infix position.
6. Make sure argument expressions are evaluated exactly once.
7. Update the iteration state by rebinding, i.e. make sure all variables visible to the payload (*var*, *i*) are bound either in *lb** (loop bindings) or in *ib** (inner bindings).
8. Support multiple arguments if that makes sense, but avoid zero arguments.

Examples

The factorial of a non-negative integer:

```
(define (factorial n)
  (product-ec (:range k 2 (+ n 1)) k) )
```

The sum of the divisors of a positive integer:

```
(define (sigma n)
  (sum-ec (:range d 1 (+ n 1))
          (if (zero? (modulo n d)))
          d ))
```

Pythagorean Triples with entries not exceeding n, i.e. (a, b, c) such that $a^2 + b^2 = c^2$ and integer $1 \leq a \leq b \leq c \leq n$:

```
(define (pythagoras n)
  (list-ec (:let sqr-n (* n n))
           (:range a 1 (+ n 1))
           (:let sqr-a (* a a))
           (:range b a (+ n 1))
           (:let sqr-c (+ sqr-a (* b b)))
           (if (<= sqr-c sqr-n))
           (:range c b (+ n 1))
           (if (= (* c c) sqr-c))
           (list a b c) ))
```

Quicksort with naive choice of pivots (stable):

```
(define (qsort xs)
  (if (null? xs)
      '()
      (let ((pivot (car xs)))
        (append
         (qsort (list-ec (:list x (cdr xs))
                         (if (< x pivot))
                         x ))
         (list pivot)
         (qsort (list-ec (:list x (cdr xs))
                         (if (>= x pivot))
                         x ))))))
```

Approximation of π by Bailey-Borwein-Plouffe's hex-digit extraction formula, i.e. $|(\texttt{pi-BBP}\ \ m) - \pi| \leq 16^{-m}$ for $m \geq 1$.

```
(define (pi-BBP m)
  (sum-ec (:range n 0 (+ m 1))
          (:let n8 (* n 8))
          (* (- (/ 4 (+ n8 1))
                (+ (/ 2 (+ n8 4))
                   (/ 1 (+ n8 5))
                   (/ 1 (+ n8 6))))
             (/ 1 (expt 16 n)) )))
```

Adding two vectors of equal length (simple program):

```
(define (vector+ x y)
  (vector-ec (:parallel (:vector xi x) (:vector yi y))
             (+ xi yi) ))
```

Adding two vectors of equal length (no intermediate lists):

```
(define (vector+ x y)
  (vector-of-length-ec (vector-length x)
    (:range i (vector-length x))
    (+ (vector-ref x i) (vector-ref y i)) ))
```

Reading a line from an input port, returning all characters read (including newline if present), or returning the eof object:

```
(define (read-line port)
  (let ((line
         (string-ec
          (:until (:port c port read-char)
                  (char=? c #\newline) )
          c )))
    (if (string=? line "")
        (read-char port) ; eof-object
        line )))
```

Reading a file, returning a list of the lines:

```
(define (read-lines filename)
  (call-with-input-file
   filename
   (lambda (port)
     (list-ec (:port line port read-line) line) )))
```

Abstraction and Performance from Explicit Monadic Reflection

Jonathan Sobel
SAS Institute
jsobel@acm.org

Erik Hilsdale
Google Inc.
eh@acm.org

R. Kent Dybvig
Daniel P. Friedman *
Indiana University
{dyb,dfried}@cs.indiana.edu

Abstract

Most of the existing literature about monadic programming focuses on theory but does not address issues of software engineering. Using monadic parsing as a running example, we demonstrate monadic programs written in a typical style, recognize how they violate abstraction boundaries, and recover clean abstraction crossings through monadic reflection. Once monadic reflection is made explicit, it is possible to construct a grammar for monadic programming that is independent of domain-specific operations. This grammar, in turn, enables the redefinition of the monadic operators as macros that eliminate at expansion time the overhead imposed by functional representations. The results are very efficient monadic programs; for parsing, the output code is competitive with good hand-crafted parsers.

1. Introduction

The use of monads to model effect-laden computation has become commonplace. This work aims to show that a fuller appreciation of the theory of monads can improve the correctness and efficiency of such implementations. We explore this through a single application domain: parsing. First, we approach parsing from the functional perspective. Next, we observe some of the shortcomings of overly simplistic monadic programming and observe what happens when we change our language to fit the theory more closely. We then explore the efficiency improvements such a foundation allows us. Finally, we point toward how the parsing example we use may be generalized.

Most of the presentation in the following section is not new. Using monads for parsing has been discussed in detail by Wadler [18], Hutton [7] and Meijer [8, 9], and Bird [1]. In a change from these presentations, however, the programs in this paper are written in the strict language Scheme [10] and include uses of Scheme's syntactic-extension mechanism (macros). We paraphrase the material from these other texts in order to familiarize the reader with our terminology and notation.

One might reasonably ask why, when exploring a topic that involves very typeful monads and their associated operators, would the presentation use the dynamically-typed language Scheme? The answer is two-fold. First, the goals of this work are more in the realm of software engineering than theory. The monads and types are useful vehicles for understanding the programs, but the true target is easy-to-write, easy-to-maintain, efficient software. Choosing Scheme should not *prevent* the use of monads for structuring programs. Second, this presentation relies heavily on syntactic abstraction as a means of turning programming patterns into language extensions, which can then be reimplemented as more efficient patterns. Such an approach is sadly impossible in any common statically-typed language.

In Section 3 we draw an analogy between monads and abstract data types. Such an analogy is not new; the example of the simple state monad with "get" and "set" operations is often presented as an abstract data type. The problem is that in larger, more realistic examples—such as functional parsing—the number of operations that requires access to the monad's underlying representation is much larger. When seen in this light, it becomes clear that a significant portion of the typical monadic-style program is treated as if it falls *inside* the abstraction boundary of the abstract data type. To complicate matters, it is very difficult for the provider of the monad data type to guess every operation that real client code might need. A review of the definition of monads leads us to monadic reflection, which provides the right tools to draw a new boundary between the very few core monad operations and the many operations that need to be partially aware of the monad's underlying representation. We rewrite portions of the code from Section 2 in a cleaner style using monadic reflection. The reflection operators, together with the standard monadic programming operators, provide enough expressiveness for us

* This work was supported in part by the National Science Foundation under grant CCR-9633109.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

to construct a grammar for the sublanguage of monadic programs. This grammar supports three-layer monadic programming: the monad definition itself, representation-aware operators, and representation-independent client code. The three-layer model stands in contrast to the typical two-layer model where everything other than the client code is treated as part of the core monad definition.

Once we have a specification of monadic programs, we are in a good position to optimize them. This we do by changing the definitions of the monadic operators in Section 4 while leaving their interfaces intact. All unnecessary closure creation is eliminated, and the work of threading store/token-stream values through the computation is handled entirely at expansion time in the new definitions. Programs that conform to our monadic-programming grammar need not be rewritten at all to benefit from the optimizations. Furthermore, all the optimizations are handled at the source level by user-defined macros, not by a new compiler pass. The approach described here is relevant for any composition of store-like monads, possibly composed with a lifting or error monad.

2. Parsing

Parsers are often described as functions from token streams to abstract syntax trees:

$$Parser = Tokens \rightarrow Tree$$

This characterization does not account for parsers modifying the token stream. That is, by the time the parser produces a tree, the token stream no longer has its original contents. Thus, the type needs to be revised:

$$Parser = Tokens \rightarrow Tree \times Tokens$$

It could be the case, though, that the parser fails to construct a tree (for example, if the input is malformed). To handle this possibility, we lift the $Tree$ type to $Tree + ErrMsg$:

$$Parser = Tokens \rightarrow (Tree + ErrMsg) \times Tokens$$

(This compact type will continue to appear in the remainder of this article, but for efficiency the programs actually use

$$Parser = Tokens \rightarrow (Tree \times Tokens) + (ErrMsg \times Tokens)$$

which is isomorphic to the prior type by the distributive property.)

The preceding paragraph follows the standard sequence of types and justifications to arrive at a desirable type for parsers,[1] but we find that the effect is to direct one's attention the wrong way. We want primarily to think about the parser's results. Parsers, however they operate, produce trees. Yet most of the type we specified for parsers is not about trees; it's about the wiring that gives us the trees. Instead, let's just say that *parsing* (not *parsers*) is one way to describe tree-producing computation. Henceforth, we shall refer to tree-producing computations (or just *tree producers*) instead of parsers.

Trying to talk about computations presents us with a problem: how do we manipulate computations in programs? We need something to act as a "representation of a tree producer." Exactly how we represent these computations depends on what aspects we want to model. Above, in the context of traditional parsing technology, we arrived at functions of a certain shape as our representations. Specifically, our representation modeled the threading of a token stream through the computation [16], as well as the possibility of failure. We call this a *threaded functional representation* of a tree producer. Let's express this abstraction in the type constructor $Producer$:

$$Producer(\alpha) = Tokens \rightarrow (\alpha + ErrMsg) \times Tokens$$

Thus, $Producer(Tree)$ is our representation for computations that produce trees.

The sum type can be represented in many ways in Scheme. For injecting values into the left and right sides of the sum, we use the operators `inl` and `inr`, respectively. These operators are polymorphic over the number of injected values, so `(inl x y z)` is acceptable usage. For dispatching on the two sum cases, we use the `sum-case` form.

⟨*example of* `sum-case`⟩≡
```
(sum-case (inl 5 2)
  ((x y) (+ x y))
  ((a b) (- a b)))
```

The value of this expression is 7. A portable implementation of `inl`, `inr`, and `sum-case` appears in the appendix. Additional options for representing sums and a discussion of their performance implications appears in Section 4.3.

It would be inconvenient to write parsers if we had to explicitly manage values of the $Producer$ types. Monads provide just the right additional structure for manipulating these values, so that programs have a consistent style, and so that the details of the $Producer$ types are abstracted away [13, 14, 19].

To make this claim more concrete, let us construct a little program in Scheme for parsing natural numbers (non-negative integers). We begin with a version written *without* the benefit of monadic operators. Even those readers who are already quite familiar with monads may find it interesting to follow the derivation of monadic structure as a kind of "pattern-mining" via syntactic abstraction.

[1] Allowing a failed parse to return a new token stream is not really standard in the literature. Why do we allow it here? Because implementations based on real imperative input streams often modify the stream even on a failed parse. In fact, such behavior is often desirable in a robust parser, to eliminate nonsense tokens from the input and continue to make progress.

2.1 Parsing Natural Numbers

A program that reads the digits in its input and parses numbers would be more typically described as scanning, not parsing, but if we take individual characters as our tokens, the distinction becomes largely moot. Here is a grammar for natural numbers:

⟨natural⟩ → ⟨digit⟩ ⟨more digits⟩
⟨more digits⟩ → ⟨digit⟩ ⟨more digits⟩
| ⟨empty⟩

The entry point for our program is the procedure `natural`,[2] which is intended to instantiate an integer-producing computation:

⟨*long version of* `natural`⟩≡
```
(define natural
  (lambda ()
    ⟨integer producer for natural⟩))
```

Using our representation scheme for computations, this means that `natural` should return a value of type

$$Producer(Int) = Tokens \rightarrow (Int + ErrMsg) \times Tokens$$

Now let's assume the existence of a nullary procedure `digit`, which returns a character producer that gets a numeric character from the token stream. It fails (i.e., returns an error message) if the next available character is not a digit or if no characters are available. Since a natural number begins with at least one digit, we get:

⟨*integer producer for* `natural`⟩≡
```
(lambda (ts1)
  (sum-case ((digit) ts1)
    ((d ts2) (⟨integer producer, given first digit⟩ ts2))
    ((msg ts2) (inr msg ts2))))
```

The values returned by `digit` are of the sum type, so we must use `sum-case` to determine whether `digit` failed or not. If so, then `natural` itself must also fail, returning the bottom value and the new tokens `ts2`. (Failures get to eat tokens, too.) The rest of the number comes from `more-digits`—to be defined shortly—which instantiates a list-producing computation, giving us a list of all the digits (numeric characters) it can extract from the front of the token stream. The portion that reads the remaining digits, then, looks much like what we already have:

⟨*integer producer, given first digit*⟩≡
```
(lambda (ts1)
  (sum-case ((more-digits) ts1)
    ((ds ts2) (⟨integer producer, given all digits⟩ ts2))
    ((msg ts2) (inr msg ts2))))
```

Finally, we have to return the answer. For this, we need an integer producer that represents a constant value (modulo free variables), an especially simple sort of computation:

⟨*integer producer, given all digits*⟩≡
```
(lambda (ts)
  (inl (string->number
         (list->string (cons d ds)))
       ts))
```

Naturally, the token stream is guaranteed to be unchanged in a simple computation.

Having completed the definition that handles the first production in the grammar, we move on to defining a procedure that handles the ⟨more digits⟩ non-terminal. More specifically, we define `more-digits` to be a nullary procedure—like `natural`—that gives us a producer. Whereas `natural` makes an integer-producing computation, `more-digits` instantiates a computation that produces a list of characters.

The grammar for ⟨more digits⟩ specifies two alternative productions: one like ⟨natural⟩ and one empty. Assuming that we want to absorb as many contiguous digits as possible into the number, we begin by trying the first alternative. If it fails, we accept the empty production (with the original token stream). Thus, `more-digits` begins this way:

⟨*long version of* `more-digits`⟩≡
```
(define more-digits
  (lambda ()
    (lambda (ts1)
      (sum-case (⟨list producer for more-digits⟩ ts1)
        ((ds ts2) (inl ds ts2))
        ((msg ts2) (⟨empty-list producer⟩ ts1))))))
```

Let's write the producer for the empty production first. It represents a constant-valued computation, similar to the one that returns the number in `natural`:

⟨*empty-list producer*⟩≡
```
(lambda (ts)
  (inl '() ts))
```

Most of the remaining code is identical to the body of `natural`, as it should be, considering that the grammar production is identical. The difference is in the return type:

⟨*list producer for* `more-digits`⟩≡
```
(lambda (ts1)
  (sum-case ((digit) ts1)
    ((d ts2)
     ((lambda (ts1)
        (sum-case ((more-digits) ts1)
          ((ds ts2)
           (⟨list producer, given all digits⟩ ts2))
          ((msg ts2) (inr msg ts2))))
      ts2))
    ((msg ts2) (inr msg ts2))))
```

Of course, one would usually β-reduce the inner `lambda` application, but we leave it in for consistency.

The code that returns the final value is like the corresponding code in `natural`, except that it does not convert the list of characters into a number:

⟨*list producer, given all digits*⟩≡
```
(lambda (ts)
  (inl (cons d ds) ts))
```

[2] It may seem unnatural (no pun intended) to define `natural` as a nullary procedure instead of a value, but it will later take additional arguments and possibly become a macro.

This completes the code for parsing natural numbers, as written by following the types rather blindly.

2.2 Becoming More Abstract

There were two distinct patterns in the code for `natural` and `more-digits`. One represents simple computations, like returning the empty list, the list of digits, or the integer value of such a list. In each case, the code looked like this:

⟨*producer pattern for returning an answer*⟩≡
```
(lambda (ts)
  (inl ⟨answer⟩ ts))
```

The other pattern was more complicated. It consisted of

1. invoking another producer,
2. receiving its return values (the result or error message and the new token stream),
3. checking for failure, and
4. either
   (a) sending the new token stream to a second producer, or
   (b) propagating the failure, skipping the second producer.

Abstracting over such code in the preceding section, the pattern looks like this:

⟨*producer pattern for sequencing two producers*⟩≡
```
(lambda (ts1)
  (sum-case (⟨producer #1⟩ ts1)
    ((⟨var⟩ ts2) (⟨producer #2⟩ ts2))
    ((msg ts2) (inr msg ts2))))
```

These two patterns correspond to the two operations used in monadic programming: `return` (also called *unit*) and `bind` (also called *monadic let*). As promised, we make coding patterns concrete by defining them as macros. Procedural definitions would be more conventional, but these macro definitions change in Section 4 to perform code rewrites that could not be accomplished with procedural abstractions.

Now `return` implements the simple answer-returning pattern:

⟨*implementation of the* `return` *pattern*⟩≡
```
(define-syntax return
  (syntax-rules ()
    ((return ?answer)
     (lambda (ts)
       (inl ?answer ts)))))
```

and `bind` implements the producer-sequencing pattern:

⟨*implementation of the* `bind` *pattern*⟩≡
```
(define-syntax bind
  (syntax-rules ()
    ((bind (?var ?producer1)
       ?producer2)
     (lambda (ts1)
       (sum-case (?producer1 ts1)
         ((?var ts2) (?producer2 ts2))
         ((msg ts2) (inr msg ts2)))))))
```

The type constructor $Producer$, together with `return` and `bind`, form a *Kleisli triple* [11]. (Actually, the third element of the Kleisli triple is not `bind`; it is `extend`, defined in Section 3.1. We find `extend` to be more convenient for mathematical manipulation and `bind` to be more convenient for monadic programming.) A Kleisli triple is equivalent to a monad; in fact, many authors drop the distinction altogether. Also, not all definitions for $Producer$, `return`, and `bind` form a Kleisli triple. The necessary properties are spelled out in detail in Section 3.

Using the monad operations, we can rewrite `natural` to be *much* more concise and readable:

⟨*definition of* `natural`⟩≡
```
(define natural
  (lambda ()
    (bind (d (digit))
      (bind (ds (more-digits))
        (return (string->number
                  (list->string
                    (cons d ds))))))))
```

The syntactic abstraction technique we just used appears repeatedly in the following sections: find a syntactic pattern, abstract it with a macro definition, and rewrite the original code more concisely using the macro definition.

One way to think about programming with `return` and `bind` is that the $Producer$ types form a family of abstract data types, and `return` and `bind` are the public operations that construct and combine producers. When we have a simple (non-producer) value and we want to instantiate a representation of a computation that produces that value, we use `return`. When we have representations for two computations and we want to sequence them, we use `bind` to construct a representation for the computation that feeds the result of the first into the second.

2.3 Monadic Combinators

We can write `more-digits` in a monadic style, but the patterns abstracted by `return` and `bind` do not completely absorb the code in `more-digits`. The part that checks to see if the first alternative failed, and if so proceeds to the second, does not fit either pattern.

⟨*unsatisfactory definition of* `more-digits`⟩≡
```
(define more-digits
  (lambda ()
    (lambda (ts1)
      (sum-case ((bind (d (digit))
                   (bind (ds (more-digits))
                     (return (cons d ds))))
                 ts1)
        ((ds ts2) (inl ds ts2))
        ((msg ts2) ((return '()) ts1))))))
```

While the code that implements alternate productions in a grammar does not fit the pattern of one of the core monad operations, it is clearly a pattern that will appear any time we need to check for the failure of one computation and perform another instead. Abstracting over the pattern gives us `orelse`, a *monadic combinator*:

⟨*unsatisfactory definition of* `orelse`⟩≡
```
(define-syntax orelse
  (syntax-rules ()
    ((orelse ?producer1 ?producer2)
     (lambda (ts1)
       (sum-case (?producer1 ts1)
         ((ds ts2) (inl ds ts2))
         ((msg ts2) (?producer2 ts1)))))))
```

If we rewrite `more-digits` one more time, using `orelse`, we get:

⟨*definition of* `more-digits`⟩≡
```
(define more-digits
  (lambda ()
    (orelse (bind (d (digit))
              (bind (ds (more-digits))
                (return (cons d ds))))
            (return '()))))
```

The definitions of both `natural` and `more-digits` now correspond very directly to the grammar for natural numbers. Furthermore, neither procedure deals explicitly with producer types except through `return` and `bind`.

We have, until now, simply assumed the existence of `digit`. Let's write it now. A call to `digit` creates a character producer that examines the first character in the token stream. If that character is numeric, it returns the character, "removing" it from the token stream. Otherwise, the computation fails and leaves the token stream unchanged:

⟨*unsatisfactory definition of* `digit`⟩≡
```
(define digit
  (lambda ()
    (lambda (ts)
      (if (or (null? ts)
              (not (char-numeric? (car ts))))
          (inr "not a digit" ts)
          (inl (car ts) (cdr ts))))))
```

(We represent our token streams in this article as lists of characters for simplicity.) Again, neither `return` nor `bind` helps simplify or clarify this code, because `digit` must access the token stream, which is not visible in procedures like `natural` that are written only in terms of the monadic operations.

3. Monads as Abstract Data Types

When we first introduced the $Producer$ type constructor, we presented it as an abstract means of representing computations by values. When we defined the `return` and `bind` operations, we provided a uniform interface to the abstraction. Ideally, all the other definitions would inhabit a space outside this abstraction boundary, even combinators like `orelse`. In the preceding section, though, we broke the $Producer$ abstraction in two ways.

First, in `orelse`, we took the results of producer expressions (constructed with `return` and `bind`, presumably) and applied them to token streams. This violation of the abstraction boundary is similar to taking a stack (a classic ADT) and performing a vector reference on it, just because we happen to know that the stack is represented as a vector. While our current representations for computations are, in fact, procedures that expect token streams, it is wrong for arbitrary code to assume such a representation. Instead, programmers need some explicit means of reifying computations as values of $Producer$ types in order to pass their own token streams (or whatever is appropriate to the specified representation types) to them and examine the results.

Second, in both `orelse` and `digit` we cobbled together arbitrary code—which happened to be of the proper type to generate $Producer$ values—and we expected to be allowed to treat those values as valid representations of computations. This violation of the abstraction boundary is similar to constructing our own vector to represent a stack and passing it to a procedure that expects a stack. This, too, is wrong. We did it because we needed to have access to the current token stream in the computation, but instead we need some explicit means of constructing a representation of a computation and reflecting it into the system so that it is accepted as something that has access to the threaded values.

The usual way to avoid violating the monad abstraction boundary is to move the offending operations—like `orelse` and `digit`—inside the boundary and treat them as fundamental monadic operators, having nearly the same status as `return` and `bind`. The weakness of such a solution is that it is often necessary to create operators like `digit` while writing a parser, not while creating a parser monad. A better solution is to create a small abstract data type for the monad and its most basic operators and to provide an interface for users of the monad to access the underlying representation of the monad (or at least a constructed view of it) in a limited way.

Monadic reflection, as introduced by Moggi [14] (though he does not use the phrase "monadic reflection") and amplified by Filinski [5], provides a means of crossing the monadic abstraction boundary with mathematically founded operators. Neither of these authors actually extends the idea of monadic reflection into the space of exposing and hiding representations in the sense of "reflective interpreters" and the like. Such an extension is new in this work, but related to the discussions by Chen and Hudak of monadic abstract data types [2].

3.1 Foundations

A monad consists of four things [12]:

1. a type constructor, T, for lifting a type α to a type that represents computations that produce values of type α,
2. a higher-order, polymorphic function (the *mapping function* of the monad) for lifting functions so that they take and return T types,

$$(\alpha \to \beta) \xrightarrow{map} (T(\alpha) \to T(\beta))$$

3. a polymorphic function (called the *unit* of the monad) for lifting a value of type α to the corresponding value of type $T(\alpha)$,

$$\alpha \xrightarrow{unit_\alpha} T(\alpha)$$

and

4. a polymorphic function (called the *multiplication* of the monad) for "un-lifting" a doubly-lifted value of type $T(T(\alpha))$ to the corresponding value of type $T(\alpha)$.

$$T(T(\alpha)) \xrightarrow{mult_\alpha} T(\alpha)$$

(In category theory, the first two elements of the monad are combined into a functor.) The possibility of iterating the T type constructor creates a sequence of "levels." The unit of the monad shifts up a level (more nesting or wrapping), and the multiplication shifts down (less nesting or wrapping). To guarantee that all the level shifting is coherent, the mapping function, unit, and multiplication must obey three equations:

$$\begin{aligned} mult_\alpha \circ map(unit_\alpha) &= id_{T(\alpha)} \\ mult_\alpha \circ unit_{T(\alpha)} &= id_{T(\alpha)} \\ mult_\alpha \circ map(mult_\alpha) &= mult_\alpha \circ mult_{T(\alpha)} \end{aligned}$$

A Kleisli triple for the monad consists of the type constructor, the unit (that is, `return`), and an *extension* operation:

$$(\alpha \to T(\beta)) \xrightarrow{extend_{\alpha,\beta}} (T(\alpha) \to T(\beta))$$

The `bind` form is simply a convenient notation for the common usage pattern of $extend$:

`((extend (lambda (v) N)) M)` = `(bind (v M) N)`

While it is possible to define the mapping function and multiplication of each monad directly, it is also possible to define both in terms of the `return` and `bind`. Only the indirect forms of the definitions follow.

For the $Producer$ type constructor we are using in our parsing examples, the mapping function—when applied to some procedure `f`—returns a procedure that takes a producer for one type and returns a producer for another. It uses `f` to get a value of the second type.

⟨*indirect definition of* `producer-map`⟩≡

```
(define producer-map
  (lambda (f)
    (lambda (producer)
      (bind (a producer)
        (return (f a))))))
```

The multiplication of the monad takes a value that represents a producer-producing computation. In other words, when it is applied to a token stream, it either fails or returns a producer and a new token stream. We can use `bind` for a very concise definition, and write `mult` this way:

⟨*indirect definition of* `mult`⟩≡

```
(define mult
  (lambda (producer-producer)
    (bind (producer producer-producer)
      producer)))
```

The unit of the monad is actually the same thing as `return`:

⟨*indirect definition of* `unit`⟩≡

```
(define unit
  (lambda (a)
    (return a)))
```

We see, then, that a monad can be defined completely in terms of a Kleisli triple. The equivalence is bidirectional; we shall not demonstrate it here, but the Kleisli triple can be defined in terms of the monad, too.

3.2 Monadic Reflection

If Kleisli triples and monads are equivalent, why would we choose one over the other? As was evident in Section 2.2, Kleisli triples are excellent tools for monadic-style programming. That is to say, they provide an appropriate means of abstractly manipulating the values that we use to represent computations.

The unit and multiplication of a monad, on the other hand, succeed in just the place where Kleisli triples failed. They provide the appropriate means for crossing the monadic abstraction boundary via level-shifting. In other words, the `unit` and `mult` are excellent tools for *monadic reflection*.

In order to talk about "clean" reflective level crossings, it is necessary to have some notion of *opaque* and *transparent* types. A simple mathematical understanding of the definition of $Producer$

$$Producer(\alpha) = Tokens \to (\alpha + ErrMsg) \times Tokens$$

treats the two sides of the equation as synonyms. From a software engineering perspective, however, there is a significant difference between the type constructor being defined and the body of its definition. To exploit this difference, let us rewrite the types of `unit` and `mult`, treating the outermost level as opaque and the inner levels as transparent whenever there are nested applications of the type constructor. They become

$$P(\alpha) \xrightarrow{unit_{P(\alpha)}} P(T(\alpha))$$

and

$$P(T(\alpha)) \xrightarrow{mult_\alpha} P(\alpha)$$

where P represents an opaque version of T. Using these types, the outer "interface" of the type always remains opaque. The types for `return` and `extend` (and thus `bind`) refer only to the opaque version of the type constructor:

$$\alpha \xrightarrow{return_\alpha} P(\alpha)$$

and

$$(\alpha \to P(\beta)) \xrightarrow{extend_{\alpha,\beta}} (P(\alpha) \to P(\beta))$$

It might seem that these operations allow no means of "reaching through" the opaque type to do anything interesting with the transparent version, but in fact, they provide

plenty of power when the operations are used in conjunction with each other.

Let us return to our unsatisfactory definitions of `digit` and `orelse` to see how judicious use of `unit` and `mult` create clean and explicit abstraction-boundary crossings. We begin with `digit`, where we want to construct a representation for a non-standard computation (i.e., one that cannot be constructed by `return` or `bind`). Furthermore, we want our hand-constructed procedure to be accepted as a valid digit (numeric character) producer. Here is the code that we want to act as a digit producer; it is taken straight from the old definition of `digit`:

⟨*custom digit producer*⟩≡
```
(lambda (ts)
  (if (or (null? ts)
          (not (char-numeric? (car ts))))
      (inr "not a digit" ts)
      (inl (car ts) (cdr ts))))
```

Just as we do for `42` or `(car '(1 2 3))`, we use `return` to construct a computation that produces this value:

⟨*digit-producer producer*⟩≡
```
(return ⟨custom digit producer⟩)
```

Finally, we use `mult` to "shift down a level." That is, `mult` will turn the digit-producer producer into a plain digit producer, explicitly coercing our hand-constructed value into a valid instance of the abstract data type.

⟨*definition of* `digit`, *using* `mult`⟩≡
```
(define digit
  (lambda ()
    (mult ⟨digit-producer producer⟩)))
```

Although `orelse` is longer and more complicated, the same kind of techniques work for rewriting it in a more satisfactory style. This time, we use both `unit` and `mult`, because `orelse` needs to shift up (lift the representation of the underlying computation into a value the user can manipulate) as well as down. We begin by lifting both of the incoming producers:

⟨*definition of* `orelse`, *using* `unit` *and* `mult`⟩≡
```
(define-syntax orelse
  (syntax-rules ()
    ((orelse ?producer1 ?producer2)
     (bind (p1 (unit ?producer1))
       (bind (p2 (unit ?producer2))
         ⟨producer that performs alternation⟩))))))
```

As in `digit`, we need a producer that cannot be written using `return` and `bind`, so we construct one by hand and use `mult` to reflect it into the system:

⟨*producer that performs alternation*⟩≡
```
(mult (return (lambda (ts1)
                (sum-case (p1 ts1)
                  ((ds ts2) (inl ds ts2))
                  ((msg ts2) (p2 ts1))))))
```

The difference between this code and what appeared in the body of the original version of `orelse` is that we have used `p1` and `p2` in place of the producers to which `orelse` was applied. Explicitly applying `p1` and `p2` to token streams is a valid thing to do, because `unit` yields transparent values wrapped in an opaque coating, and `bind` strips away the coating.

3.3 Abstracter and Abstracter

Just as `return` and `bind` are syntactic abstractions of the patterns for simple construction and sequencing of producer values, we can formulate patterns that abstract the common usage of `unit` and `mult`. We assert that, if we were to go out and write hundreds of procedures using `unit` and `mult`, we would see the same patterns over and over: the ones used in `digit` and `orelse`. The pattern for using `unit` looks like this:

⟨*producer pattern for reifying a producer*⟩≡
```
(bind (⟨var⟩ (unit ⟨producer #1⟩))
  ⟨producer #2⟩)
```

And whenever we use `mult`, we apply `return` to a `lambda` expression:

⟨*producer pattern for reflecting a constructed producer*⟩≡
```
(mult (return (lambda (⟨var⟩)
                ⟨expression⟩))))
```

The effect of these compositions is even more evident when the constituent operations are written as arrows. Assume that ⟨*producer #1*⟩ has opaque type $P(\alpha)$ but ⟨*producer #2*⟩ treats ⟨*var*⟩ as the transparent $T(\alpha)$, returning a value of opaque type $P(\beta)$. In terms of `extend`, this means that the body is like a function

$$T(\alpha) \xrightarrow{g} P(\beta)$$

and the whole reification composition is:

$$P(\alpha) \xrightarrow{unit_{P(\alpha)}} P(T(\alpha)) \xrightarrow{extend_{T(\alpha),\beta}(g)} P(\beta)$$

The reflection composition yields a simple conversion from transparent to opaque types:

$$T(\alpha) \xrightarrow{return_{T(\alpha)}} P(T(\alpha)) \xrightarrow{mult_{\alpha}} P(\alpha)$$

As is our wont, we turn these patterns into macros. The first we call `reify`:

⟨*definition of* `reify`⟩≡
```
(define-syntax reify
  (syntax-rules ()
    ((reify (?var ?producer1)
       ?producer2)
     (bind (?var (unit ?producer1))
       ?producer2))))
```

The second we call `reflect`:

⟨*definition of* `reflect`⟩≡
```
(define-syntax reflect
  (syntax-rules ()
    ((reflect (?var) ?expression)
     (mult
       (return (lambda (?var) ?expression))))))
```

Effectively, `reflect` exposes the threaded token stream to the expression in its body.

We can now use `reflect` to simplify `digit` one more time:

⟨*definition of* `digit`⟩≡
```
(define digit
  (lambda ()
    (reflect (ts)
      (if (or (null? ts)
              (not (char-numeric? (car ts))))
          (inr "not a digit" ts)
          (inl (car ts) (cdr ts))))))
```

Using `reflect` and `reify` together, we get a new definition of `orelse`:

⟨*definition of* `orelse`⟩≡
```
(define-syntax orelse
  (syntax-rules ()
    ((orelse ?producer1 ?producer2)
     (reify (p1 ?producer1)
       (reify (p2 ?producer2)
         (reflect (ts1)
           (sum-case (p1 ts1)
             ((ds ts2) (inl ds ts2))
             ((msg ts2) (p2 ts1)))))))))
```

These are our final definitions of `digit` and `orelse`. They are now completely explicit in their crossings of abstraction boundaries. Also, the representation of computations is remarkably abstract. We need know only that producers can be applied to token streams and that they return a sum value and a new token stream. We never use `lambda` to construct producers directly.

3.4 A Grammar for Monadic Programming

When we decried the original code for `digit` and `orelse`, we were appealing to what we hoped was a shared implicit intuition, which we now make explicit. What is it that makes us uncomfortable with the following code?

⟨*bad code*⟩≡
```
(bind (x (natural))
  (lambda (ts)
    (inl (+ x 2) (cdr ts))))
```

What bothers us is that we expect the body of the `bind` expression to be another `bind` or a `return`, or maybe a `reify` or a `reflect`, but certainly not a `lambda`. In other words, programs written in a "monadic style" are really written in a particular sublanguage in which only certain forms are allowable.

We make the language of monadic programming explicit by presenting a grammar for it. This grammar requires both the right-hand side and the body of `bind` expressions to be other monadic expressions, and so on.

⟨program⟩ → $D \ldots$ (`run` M E)
D → (`define` V_M R)
R → (`lambda+` ($V \ldots$) M)
M → (`return` E)
| (`bind` (V M) M)
| (`reflect` (V) E)
| (`reify` (V M) M)
| (V_M $E \ldots$)
| derived monadic expression
E → arbitrary Scheme expression

By "derived monadic expression," we mean user-defined syntactic forms—like `orelse`—that expand into monadic expressions. By "arbitrary Scheme expression," we mean code that does *not* contain monadic subexpressions.

The relationships among `return`, `bind`, `reflect`, and `reify` might be better understood by examining typing rules for them. The rules in Figure 1, for the sake of brevity, abbreviate $Producer$ as P. No rules are given for arbitrary expressions E. Instead, these four rules are meant to augment the typing rules for standard expressions.

There are two additional forms introduced in this grammar: `run` and `lambda+`. Without `lambda+`, there would be no "roots" for the portion of the grammar that deals with monadic expressions, nowhere to get started with monadic programming. For now, we let `lambda+` be synonymous with `lambda`. To conform to this grammar, `digit`, `natural`, and `more-digits` should be modified to use `lambda+`.

The `run` form simply starts a computation by passing the initial token stream (or other store-like value) to a producer:

⟨*definition of* `run`⟩≡
```
(define-syntax run
  (syntax-rules ()
    ((run ?producer ?exp)
     (?producer ?exp))))
```

For example, this use of `run`:

```
(run (natural) (string->list "123abc"))
```

would run our natural-number parsing program and return `123` (left-injected) and the remaining characters `(#\a #\b #\c)`.

4. Optimizing Monadic Programs

With both the parsing operators like `digit` and the simple client code like `natural` written in terms of `return`, `bind`, `reflect`, and `reify`, the inner abstraction boundary around the monad is satisfyingly small. The performance, though, is inadequate for use in a real compiler or interpreter. The largest source of overhead expense is all the closure creation, which a compiler may or may not eliminate. To provide a stronger guarantee than "we hope the compiler cleans this up for us," it is possible to create new closure-free versions of the macros for the core operators.

$$(return)\quad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash (\texttt{return}\, E) : P(\tau)}$$

$$(bind)\quad \frac{\Gamma \vdash M_1 : P(\tau_1) \qquad \Gamma, v{:}\tau_1 \vdash M_2 : P(\tau_2)}{\Gamma \vdash (\texttt{bind}\, (v\, M_1)\, M_2) : P(\tau_2)}$$

$$(reify)\quad \frac{\Gamma \vdash M_1 : P(\tau_1) \qquad \Gamma, v{:}(S \to (\tau_1 + ErrMsg) \times S) \vdash M_2 : P(\tau_2)}{\Gamma \vdash (\texttt{reify}\, (v\, M_1)\, M_2) : P(\tau_2)}$$

$$(reflect)\quad \frac{\Gamma, v{:}S \vdash E : (\tau + ErrMsg) \times S}{\Gamma \vdash (\texttt{reflect}\, (v)\, E) : P(\tau)}$$

Figure 1. Typing Rules

Let's look at the expansion of a small part of our natural number parser, the first of the alternatives in `more-digits`:

⟨`more-digits` *fragment*⟩≡
```
(bind (ds (more-digits))
  (return (cons d ds)))
```

Using the most recent versions of `bind` and `return`, this code expands into:

⟨`more-digits`-*fragment expansion*⟩≡
```
(lambda (ts1)
  (sum-case ((more-digits) ts1)
    ((ds ts2) ((lambda (ts)
                 (inl (cons d ds) ts))
               ts2))
    ((msg ts2) (inr msg ts2))))
```

In the expansion, every subexpression that denotes a producer value, be it a call like `(more-digits)` or a `lambda` expression, is applied to a token stream. This property will hold in all such programs, as it is guaranteed by our grammar.

4.1 Eliminating the Closures

According to the implementation from the preceding sections, every producer expression will construct a closure, either directly (by expanding into a `lambda` expression) or indirectly (by invoking a procedure that returns a closure). These closures are then immediately applied to token streams. Of course, the direct expansion into `lambda` and immediate application (as in the preceding example) becomes `let` in nearly every Scheme implementation, but the sites where closures are returned by procedure calls are much harder for a compiler to optimize. One way to improve both the memory and space use of the code is to remove the need for the two-stage application. Since, in the expansion, the token stream is always available to finish off the application, we never need to partially apply procedures like `digit`. Instead, we can modify the definitions of our monadic-programming macros so the token stream is passed as an extra argument to the existing procedures.

The `lambda+` form, which we introduced in the preceding section, is the starting point for the extra arguments:

⟨*improved definition of* `lambda+`⟩≡
```
(define-syntax lambda+
  (syntax-rules ()
    ((lambda+ (?formal ...) ?body)
     (lambda (?formal ... ts)
       ⟨body of token-accepting function⟩)))))
```

We now need to thread the token-stream argument appropriately into the body. Since we know that this body must be a monadic expression, we need only change the implementation of those forms consistently with the new "un-curried" `lambda+` form.

The simplest case is if the body is an application of a user-defined procedure, such as a call to `digit`. In this case, we need to make sure to thread our store through as the last argument to the call. We accomplish this with the helper form `with-args`:

⟨*definition of* `with-args`⟩≡
```
(define-syntax with-args
  (syntax-rules ()
    ((with-args (?extra-arg ...)
                (?operator ?arg ...))
     (?operator ?arg ... ?extra-arg ...))))
```

It may seem that `with-args` is more general than necessary, since it can handle multiple extra arguments, but this generality offers us a great deal of leverage, as we shall see later. Using `with-args`, we can finish the definition of `lambda+` like this:

⟨*body of token-accepting function*⟩≡
```
(with-args (ts) ?body)
```

This code is well-formed only if the body is in the form of an operator and some arguments. If we look back at the grammar, we see that this is indeed the case.

The definitions of `bind` and `return` must now handle extra input in their patterns. In `bind`, these extra arguments must be threaded into the subforms:

⟨*improved definition of* `bind`⟩≡
```
(define-syntax bind
  (syntax-rules ()
    ((bind (?var ?rhs) ?body ?ts ...)
     (sum-case (with-args (?ts ...) ?rhs)
       ((?var ?ts ...)
        (with-args (?ts ...) ?body))
       ((msg ?ts ...) (inr msg ?ts ...))))))
```

The token-stream parameter(s) used in the right-hand side are the same ones (i.e., the same names as those) bound by `let-values` in the body. We need not worry about shadowing, though, since the token stream is necessarily threaded, and there can be no free references to it in the body.

In `return`, the extra arguments need to be threaded back out, along with the desired return value.

⟨*improved definition of* `return`⟩≡
```
(define-syntax return
  (syntax-rules ()
    ((return ?answer ?ts ...)
     (inl ?answer ?ts ...))))
```

Thus, `return` becomes an alias for `inl`, as it should be.

Since we no longer run a computation by first evaluating it and then passing the result a token stream, we must modify `run` to follow the new protocol:

⟨*improved definition of* `run`⟩≡
```
(define-syntax run
  (syntax-rules ()
    ((run ?producer ?exp ...)
     (with-args (?exp ...) ?producer))))
```

The new version converts the initial stream(s) into argument(s) to the producer. The grammar in the preceding section supported only a single "hidden" argument. In order for it to support the generality that is included in the new versions of these operators, it should be modified to allow additional arguments to `run`. The same sort of modification is necessary in the grammar rule for `reflect`. It should allow additional variables to be bound to the current values of the additional store-like parameters.

The `reflect` and `reify` forms require a bit more analysis before they can be optimized. We begin with `reflect`. There are two ways to proceed here. One is to recognize that while the added syntax we have imposed with `reflect` is good for software engineering, the `reflect` form is still mathematically equivalent to what we started with: a directly constructed `lambda` expression for a producer. (This mathematical equivalence, which comes from the monad equations, is a good thing. It validates our sequence of abstractions and transformations.) The other approach is simply to begin with the macro definition for `reflect` and follow all the definitions and β-reductions, eventually concluding that `reflect` is merely an alias for `lambda`. Either way, the result is the same. Applying a `reflect` form to a token stream is the same as applying the corresponding `lambda` expression. In other words, under our new protocol, `reflect` expands into a `let`.

⟨*improved definition of* `reflect`⟩≡
```
(define-syntax reflect
  (syntax-rules ()
    ((reflect (?var ...) ?expression ?ts ...)
     (let ((?var ?ts) ...)
       ?expression))))
```

We have carried the potential for threading multiple values through `reflect`, just as we did for `with-args`. This generalizes the version of `reflect` in the preceding sections. Of course, the `let` we just introduced merely renames the token-stream parameter(s).

More mechanism is required to implement `reify` well. If we continue to reify computations as values, using the threaded functional representations, we must pay for first-class procedures:

⟨*improved definition of* `reify`*, first try*⟩≡
```
(define-syntax reify
  (syntax-rules ()
    ((reify (?var ?rhs) ?body ?ts ...)
     (let ((?var (lambda (?ts ...)
                   (with-args (?ts ...) ?rhs))))
       (with-args (?ts ...) ?body)))))
```

While this works, it creates the first-class procedures we were trying to avoid. The point of `reify` is to allow the code in the body to poke at the reified producer by passing it token streams and examining the results explicitly. We can support this functionality without forming a closure by constructing the expansion-time equivalent of a locally-applicable closure: a local macro. We bind (at compile time) the variable to a syntax transformer that generates the right code:

⟨*improved definition of* `reify`⟩≡
```
(define-syntax reify
  (syntax-rules ()
    ((reify (?var ?rhs) ?body ?ts ...)
     (let-syntax
         ((?var (syntax-rules ()
                  ((?var ?ts ...)
                   (with-args (?ts ...) ?rhs)))))
       (with-args (?ts ...) ?body)))))
```

This new definition has a certain constraint that was not present in the procedural version: the bound variable must appear in the `?body` only in operator position. This is due, in part, to the inability to do macro-like replacement of plain identifiers in Scheme's standardized syntactic extension mechanisms,[3] but the restriction boosts efficiency anyway. It prevents us from leaking unwanted computational effort into the runtime.

The new definition of `reify` is backed by a mathematical equivalence, too. The original definition of `reify` was mathematically equivalent (again by the monad equations) to

[3] Some implementations, such as Chez Scheme [3], do support substitution for all identifiers in the scope of the macro binding.

substituting the right-hand side for the variable in the body. Our new definition does just this.

4.2 The Closure-Free Expansion

Using the new definitions for `return`, `bind`, etc., we get wonderfully improved expansions for monadic programs. For instance, the fragment of code at the beginning of this section, which used to contain five different closure-creation sites, now expands into the following:

⟨`more-digits`-*fragment expansion, improved*⟩≡
```
(sum-case (more-digits ts)
  ((ds ts) (inl (cons d ds) ts))
  ((msg ts) (inr msg ts)))
```

The new code creates no closures at all. The lack of rampant anonymous procedures also makes the new code much more amenable to compiler optimizations. For example, if all the code for parsing is put in a single mutually recursive block (i.e., a single `letrec`), we would expect a good compiler to turn all the calls into direct calls to known code addresses.

4.3 Alternative Sum-Type Representations

The representation we have used for sum-type values requires a dispatch at every return site (see the appendix). There are two useful alternatives to this approach.

One alternative is simply to return no value for failure, and one value for success. This is no faster in the abstract than returning a boolean value, since there remains a dispatch at every return site, but some implementations of Scheme provide especially fast ways to dispatch on argument count [4]. Thus, while this technique does not decrease the number of steps, it may decrease the absolute running time of the program.

The second alternative is the only one that really eliminates the return-site dispatch. One provable property of our monad definition is that, in the absence of reification, failures are propagated up through the entire extent of the computation. In other words, it is only in operators like `orelse` that failures may be caught and acted upon. We could capture a continuation at each such dispatch point and pass it down into the subcomputations. When we want to signal a failure (as in `digit`), we invoke the most recently captured continuation. This is close in both spirit and theory to the direct-style monadic programming of Filinski [5]. In this implementation, no checks have to be made at each normal return point, but the overhead for continuation creation may outweigh this savings. (Actually, this technique does not require full continuations; it needs only escapes, which may be implemented more cheaply than full first-class continuations.)

Naïvely implemented parsing routines, like the one we wrote for natural numbers, will make heavy use of `orelse`. Thus, depending on the expense of the second alternative, it may not be worthwhile. On the other hand, if a grammar is made very deterministic through the use of pre-calculation (of "first" and "follow" sets, for example), then failures may be truly exceptional, and the continuation-based alternative could eliminate a significant amount of overhead.

5. Conclusions

The example in this paper has been exclusively about parsing, but the results extend across a much broader scope: any composition of store-like monads, possibly composed with an error or lifting monad. The macros in the preceding section are defined in such a way that it is easy to support the threading of multiple store-like parameters through computations. In fact, the only form that must be changed to add a parameter is `lambda+`. For example, if we want to thread three stores through the computation, we rewrite `lambda+` this way:

⟨*definition of* `lambda+` *with 3 stores*⟩≡
```
(define-syntax lambda+
  (syntax-rules ()
    ((lambda+ (?formal ...) ?body)
     (lambda (?formal ... s1 s2 s3)
       (with-args (s1 s2 s3) ?body)))))
```

The use of `with-args` in all the other forms will drive them to expand in ways that propagate the store parameters correctly. With our current definitions, any user-level code that uses `reflect` must be rewritten to accept the extra store parameters, and any code that uses `reify` must apply the reified values to additional arguments. One way that this work could be extended is to implement a mechanism by which user-level code would be able to refer to only those "hidden" parameters that they need to see at any point. This is possible with more sophisticated macros.

At the end of Section 4.3 we alluded to the possibility of preprocessing the grammar and/or parser to boost its performance. Another possible direction we see for research in this area is to combine the "fast LR parsing via partial evaluation" techniques of Sperber and Thiemann [17] with our expansion-time optimizations. The primary goal of most functional parsing research is to make parsers easier for *people* to write, but the same results should simplify the work of parser generators.

Even if our goal had been to compile monadic programs directly into a lower-level language, the more rigorous style afforded by explicit monadic reflection would make the compilation process more tractable. For example, a typical parser written in Haskell or Scheme will be much easier to convert to C without arbitrary anonymous functions in the user code, which the user expects to be treated as representations of computations.

The measurable performance benefit from the optimized (store-threaded) macros varies depending on the Scheme implementation. One production-grade parser that uses the macros from this article is used to parse a kind of annotated table-definition language for databases. The parser is split into modules that do lexical analysis and phrasal analysis, with the output of the first serving as the token stream for

the second. One of the regular inputs to this parser contains about 150 tables, at a total file length of about 3000 lines. Running on Chez Scheme [3], the total time to parse the input and construct the parse tree is less than 2 tenths of a second on typical personal computer hardware. There is no measurable difference between the different versions of the macros, implying that Chez Scheme is already eliminating all the overhead that might be introduced by closure creation, even across procedure calls. Running on DrScheme [15, 6], the total parse time on the same hardware is about 1.5 seconds. There is a 10% to 12% decrease in the parse time using the improved macros from Section 4.

Thus, the benefits of following a grammar for monadic programming—even for operators that depend somewhat on the monad's representation—are two-fold: First, the programs written in a stricter monadic style are more elegant, less *ad hoc*. While it is possible to write well-typed monadic programs without using explicit reflection operators, they violate abstractions in the same ways that ill-typed (but runnable) programs do in C when they cast a file pointer to be an integer and add 18 to it, just because some programmer happens to know that the result will be meaningful. Second, the rigor that makes programs *feel* better can also make them *run* better. While a sufficiently "smart" compiler or partial evaluator might eliminate the closure overhead just as well as our rewritten operators, there is an element of certainty that comes from shifting the work even earlier than compile time. By making sure that the optimization happens at expansion time, we depend less on the the analysis phase of a compiler and more on our own mathematics.

Acknowledgments

The authors appreciate the comments and recommendations provided by anonymous referees. Kevin Millikin was involved in many discussions as the paper was initially written. Michael Sperber offered several valuable suggestions, including the recommendation that this work be shared in the context of the Scheme Workshop. A special thanks goes to Mitch Wand for extensive comments, careful readings, and pointed demands for a better treatment of the types of the basic operators.

References

[1] Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall Europe, second edition, 1998.

[2] Chih-Ping Chen and Paul Hudak. Rolling your own mutable ADT: A connection between linear types and monads. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Paris, France, January 1997. ACM Press.

[3] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[4] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, 1990.

[5] Andrzej Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, New York, January 1994. ACM Press.

[6] Matthew Flatt. PLT MzScheme: Language manual. `http://download.plt-scheme.org/doc/mzscheme/`, 2005.

[7] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[8] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[9] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

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

[11] Heinrich Kleisli. Every standard construction is induced by a pair of adjoint functors. In *Proceedings of the American Mathematical Society*, volume 16, pages 544–546, 1965.

[12] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, second edition, 1998.

[13] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1989.

[14] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[15] PLT. PLT DrScheme: Programming environment manual. `http://download.plt-scheme.org/doc/drscheme/`, 2005.

[16] David A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[17] Michael Sperber and Peter Thiemann. The essence of LR parsing. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 146–155, La Jolla, 1995. ACM Press.

[18] Philip Wadler. How to replace failure by a list of successes. In *Second International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

[19] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

Appendix

As long as sum-type values never need to be stored in data structures (and they do not, in this article), they can be represented efficiently as "tagged" multiple values. The tag is simply `#t` for left-injected values:

⟨*tag-based* `inl`⟩≡
```
(define-syntax inl
  (syntax-rules ()
    ((inl ?arg ...)
     (values #t ?arg ...))))
```

and `#f` for right-injected values:

⟨*tag-based* `inr`⟩≡
```
(define-syntax inr
  (syntax-rules ()
    ((inr ?arg ...)
     (values #f ?arg ...))))
```

For "casing" sum-type values, we use a new syntactic form `sum-case`, as demonstrated in the following example:

⟨*sum type example*⟩≡
```
(define add1-or-zero
  (lambda (thunk)
    (sum-case (thunk)
      ((n) (+ n 1))
      ((z) 0))))

(list (add1-or-zero (lambda () (inl 42)))
      (add1-or-zero (lambda () (inr 0))))
```

The last expression evaluates to the list `(43 0)`.

Defining a macro for `sum-case` is relatively straightforward in a Scheme implementation that has a direct means of generating temporary variables in macros. The portable version of the macro is made much more complicated by the need to generate a list of temporaries:

⟨*portable tag-based* `sum-case`⟩≡
```
(define-syntax sum-case
  (syntax-rules ()
    ((sum-case ?exp
       ((?left-var ...) ?left-result)
       ((?right-var ...) ?right-result))
     (gen-var-list (?left-var ...)
       (sum-case-help () ?exp
         ((?left-var ...) ?left-result)
         ((?right-var ...) ?right-result))))))

(define-syntax sum-case-help
  (syntax-rules ()
    ((sum-case-help (?temp ...) ?exp
       ((?left-var ...) ?left-result)
       ((?right-var ...) ?right-result))
     (call-with-values (lambda () ?exp)
       (lambda (tag ?temp ...)
         (if tag
             (let ((?left-var ?temp) ...)
               ?left-result)
             (let ((?right-var ?temp) ...)
               ?right-result)))))))
```

```
(define-syntax gen-var-list
  (syntax-rules ()
    ((gen-var-list ()
       (?head (?y ...) ?tail ...))
     (?head (?y ...) ?tail ...))
    ((gen-var-list (?v0 ?v ...)
       (?head (?y ...) ?tail ...))
     (gen-var-list (?v ...)
       (?head (?y ... temp) ?tail ...)))))
```

An operational semantics for R^{5}RS Scheme

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Robert Bruce Findler
University of Chicago
robby@cs.uchicago.edu

Abstract

This paper presents an operational semantics for the core of Scheme. Our specification improves over the existing R^{5}RS denotational specification in four ways. First, it is more complete, since it contains *eval*, **quote**, and *dynamic-wind*. Second, it models multiple values in a way that does not require changes to unrelated parts of the language. Third, it provides a more faithful model of Scheme's undefined order of evaluation. Finally, it is executable, because it is encoded as a program in PLT Redex, a domain-specific language for writing operational semantics. The executable specification allows others to experiment with our specification and allows us to build a specification test suite, which improves our confidence that our system is a faithful model of Scheme.

In addition to contributing a specification of Scheme, this paper presents several novel modeling techniques for Felleisen Hieb-style rewriting semantics that we discovered while developing our R^{5}RS Scheme semantics. All are applicable to a wider range of problems than the specific uses we have for them, and the fact that they combine seamlessly in our full R^{5}RS model shows that they scale to real languages.

1. Introduction

The Revised^5 Report on the Algorithmic Language Scheme [15], R^{5}RS, provides an informal, English specification of Scheme and a denotational model of a core Scheme language. The denotational specification is more precise than the informal specification, but is also incomplete with respect to it. For instance, the formal specification does not present the top-level mentioned throughout the informal specification, and is missing key procedures such as *dynamic-wind* and *eval* whose inclusion could have a significant impact on the formalism. While that is not necessarily a problem — the measure of a model is not its completeness but its ability to clearly and accurately explain its subject — Gasbichler et al's recent explanation of the difficulties involving dynamic contexts and threads [12], for instance, demonstrate that the formal model is insufficient for some important questions.

In this paper we give a new treatment of the R^{5}RS formal semantics that models more of the language described in the informal semantics section than the formal semantics section in the R^{5}RS Scheme document does. Rather than extending the denotational semantics with extra constructs, we present an alternate specification as a small-step operational semantics. We do this for two major reasons. First, to make the semantics natively executable: operational semantics are much more amenable to direct execution than denotational semantics. Second, to allow for nondeterminism and nonconfluence: small-step operational semantics are particularly well-suited for modeling programming languages with nondeterministic and nonconfluent behavior. We make important use of nondeterminism in our model, as we will explain in section 2.

As a side benefit of using a small-step operational encoding, we can use PLT Redex [17], a domain-specific language for context-sensitive term-rewriting systems, to give a directly executable operational encoding for our model. PLT Redex provides a graphical browser for exploring reduction graphs and allows us to maintain a large test suite of terms and their expected normal forms that we can run whenever we change any reduction rules. This test suite increases our confidence that our model is a faithful representation of Scheme.

While writing our model, we developed new techniques for modeling some of Scheme's features. In the rest of our paper we first introduce those techniques in isolation to explain our models for particular Scheme features, and then combine them into a single unified model. In section 2 we show how to use nondeterminism to model Scheme's unspecified application order; in section 3 we show a novel technique for modeling multiple return values; in section 4 we give a model for **quote** and *eval*; and in section 5 we give a model for *call/cc* in the presence of *dynamic-wind*. Finally in section 6 we combine all those models along with several other more straightforward features: **if**, *cons* and cons-cell mutation, variable-arity procedures, *apply*, and an object-identity-sensitive notion of *eqv?* equality.

We will assume the reader has a basic familiarity with context-sensitive reduction semantics. Readers unfamiliar with this system may wish to consult Felleisen and Flatt's monograph [5] or Wright and Felleisen [24] for a thorough introduction or our previous work with Flatt and Felleisen [17] for a somewhat lighter one. We should also emphasize before we proceed that this semantics still leaves out many important Scheme features — among them the numeric tower, the top-level environment, and macros — but that it models more features than the Report's formal semantics does and is more suitable for extension.

2. Unspecified application order

In evaluating a procedure call, the R^{5}RS document deliberately leaves unspecified the order in which arguments are evaluated, but section 4.1.3 specifies that

> *the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with* some *sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.*

In the formal semantics section, the authors explain how they model this ambiguity:

> *[w]e mimic [the order of evaluation] by applying arbitrary permutations* permute *and* unpermute *. . . to the arguments in a call before and after they are evaluated. This is not quite*

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ ((x\ v)\cdots)\ e) \\
e & ::= & (e\ e\cdots) \mid (\textbf{set!}\ x\ e) \mid (\textbf{begin}\ e\ e\cdots) \mid v \\
v & ::= & (\textbf{lambda}\ (x\cdots)\ e) \mid n
\end{array}
\qquad
\begin{array}{lcl}
C & ::= & (v\cdots C\ e\cdots) \mid (\textbf{set!}\ x\ C) \mid (\textbf{begin}\ C\ e\ e\cdots) \mid [] \\
x & ::= & \text{identifiers, store locations for mutable bindings} \\
n & ::= & \text{numbers}
\end{array}
$$

$$
\begin{array}{lcll}
(\textbf{store}\ ((x_1\ v_1)\cdots)\ C[((\textbf{lambda}\ (x_2\cdots)\ e)\ v_2\cdots)]) & \rightarrow & (\textbf{store}\ ((x_1\ v_1)\cdots(x_2'\ v_2)\cdots)\ C[e[x_2'\cdots/\ x_2\cdots]]) & (\text{MApp}) \\
 & & (\#x_2 = \#v_2, \text{each } x_2' \text{ fresh}) & \\
(\textbf{store}\ ((x_1\ v_1)\cdots)\ C[((\textbf{lambda}\ (x_2\cdots)\ e)\ v_2\cdots)]) & \rightarrow & \textbf{error:}\ \text{wrong number of arguments} & (\text{MAppErr}) \\
 & & (\#x_2 \neq \#v_2) & \\
(\textbf{store}\ ((x_1\ v_1)\cdots(x\ v)\ (x_2\ v_2)\cdots)\ C[(\textbf{set!}\ x\ v')]) & \rightarrow & (\textbf{store}\ ((x_1\ v_1)\cdots(x\ v')\ (x_2\ v_2)\cdots)\ C[0]) & (\text{MSet}) \\
(\textbf{store}\ ((x_1\ v_1)\cdots(x\ v)\ (x_2\ v_2)\cdots)\ C[x]) & \rightarrow & (\textbf{store}\ ((x_1\ v_1)\cdots(x\ v)\ (x_2\ v_2)\cdots)\ C[v]) & (\text{MLookup}) \\
(\textbf{store}\ ((x\ v)\cdots)\ C[(\textbf{begin}\ v\ e_1\ e_2\cdots)]) & \rightarrow & (\textbf{store}\ ((x\ v)\cdots)\ C[(\textbf{begin}\ e_1\ e_2\cdots)]) & (\text{MSeq}) \\
(\textbf{store}\ ((x\ v)\cdots)\ C[(\textbf{begin}\ e)]) & \rightarrow & (\textbf{store}\ ((x\ v)\cdots)\ C[e]) & (\text{MTrivSeq}) \\
(\textbf{store}\ ((x\ v)\cdots)\ C[(-\ \ulcorner n \urcorner)]) & \rightarrow & (\textbf{store}\ ((x\ v)\cdots)\ C[\ulcorner -n \urcorner]) & (\text{MNeg})
\end{array}
$$

Figure 1. Core Scheme with mutation

right since it suggests, incorrectly, that the order of evaluation is constant throughout a program . . .. [section 7.2]

In this section we present an operational technique that captures the intended semantics more faithfully. We begin by considering a core Scheme with arbitrary arity procedures, **set!**, numbers, and negation, but with a fixed left-to-right order of evaluation for applications, as shown in figure 1. It is a minor variation of Felleisen and Hieb's Λ_S [6]. A program consists of a store that associates variable names to values and an expression, where expressions are built up of numbers, arbitrary-arity lambda terms and applications, **set!**, and **begin** expressions, and a built-in negation operator. MApp gives the rule for application of a procedure to fully-evaluated arguments: make one fresh identifier x_i' for each formal parameter x_i, introduce a new binding in the store for each x_i' associating it with the corresponding argument v_i in the application, and then rewrite the application as the procedure's body with each occurrence of an x_i rewritten into the corresponding x_i' (in this figure as in all figures in this paper, we will use vertically-centered ellipses $\cdots$ to indicate any number of occurences, including zero, of the preceding element). MAppErr gives the rule for procedures applied to the wrong number of arguments: rewrite the term in its entirety to an error message, which halts the program immediately because it abandons the application's original context. MSet rewrites to the constant 0 but also replaces the value associated with the given identifier in the store with the given replacement. (We choose to have **set!** return the constant 0 in this semantics as a "quick and dirty" unique value; in the examples that follow 0 never appears in any program term except as the result of assignment.) MLookup replaces an identifier with its associated value in the store when that value becomes necessary (*i.e.*, when it appears as a redex in an evaluation context). MSeq drops the first subexpression in a **begin** when there are more expressions to evaluate, and MTrivSeq drops the **begin** when there is only one expression to evaluate. The last rule, MNeg, simply negates its argument (the notation $\ulcorner n \urcorner$ indicates the syntactic representation corresponding to the mathematical number n).

The order of evaluation is determined by the grammar for evaluation contexts (C). The first production of the grammar specifies that evaluation of a sub-expression of an application only takes place when all of the sub-expressions to its left are values (or have been reduced to values). If we replace that first production with this one:

$$C ::= (e \cdots C\ v \cdots) \mid \ldots$$

the semantics would specify a right-to-left order instead.

Either of these choices results in a system with unique decomposition. That is, each term can only be split into an evaluation context and a reducible sub-expression in one way (unless it is stuck or an answer). Accordingly, there is at most one way to reduce any expression.

To model a language with unspecified order of operations instead, we can use a reduction system with non-unique decomposition to model the choice. We might be tempted to use this definition of evaluation contexts:

$$C ::= (e \cdots C\ e \cdots) \mid \ldots$$

Since this definition allows the hole to appear in any subexpression of an application, this simple program that negates 1, negates 2, and then applies a trivial procedure to the results

$$((\textbf{lambda}\ (x\ y)\ y)\ (-\ 1)\ (-\ 2))$$

can be split into an evaluation context with either $(-\ 1)$ or $(-\ 2)$ as the reducible expression.

At first glance, this appears to be a faithful model of R^{5}RS Scheme. It is not. Consider this application of two **set!** expressions in a store binding x to *1*.

$$
\begin{array}{l}
(\textbf{store}\ ((x\ 1)) \\
\quad ((\textbf{set!}\ x\ (-\ x)) \\
\quad\ (\textbf{set!}\ x\ (-\ x))))
\end{array}
$$

In Scheme, this program should always reduce to the application of zero to zero with x set to *1* in the store (and then get stuck). According to R^{5}RS, no matter which of the application's subterms is reduced first, the result should be that x is negated twice. If we just modify evaluation contexts as above, however, we allow other interleavings. The problem is that that definition of evaluation contexts would allow a different argument of the same application to take one step of computation every step of the way, which may produce an outcome that could not be reached by any sequential ordering.

We discovered this problem while experimenting with that reduction system in PLT Redex. We encoded the erroneous reduction system in PLT Redex and automatically generated the reduction sequence for the above term, shown in figure 2. The first term is shown on the left. The top-most and the bottom-most paths correspond to the two sequential orderings and result in the proper store. In the middle section, the two assignments are interleaved, resulting in -1 being left in the store.

With that in mind, we can design a more sophisticated strategy that captures unspecified evaluation order but only allows sequential orderings. Figure 3 shows the necessary revisions to core Scheme to support R^{5}RS-style procedure applications (each replaces the appropriate rule from figure 1 — the other rules in that figure are unchanged). The basic idea is to use non-deterministic choice to pick a sub-expression to reduce only when we have not already committed to reducing some other subexpression. To achieve

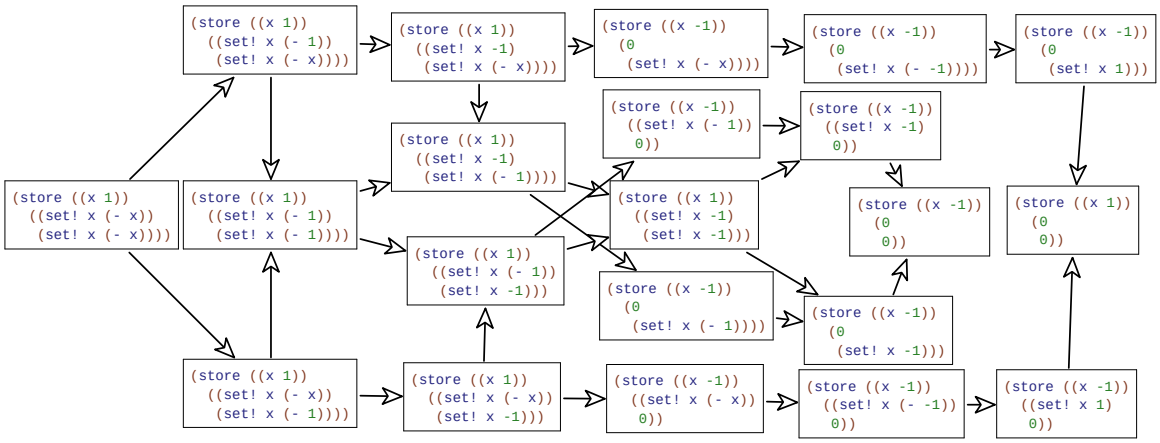


Figure 2. Interleavings possible with an erroneous unspecified-application-order model

$$\begin{array}{rcl} \mathit{inert} & ::= & v^{\circ} \mid e \\ C & ::= & (\mathit{inert} \cdots C^{\circ}\ \mathit{inert} \cdots) \mid \ldots \end{array}$$

$$\begin{array}{lcll} (\textbf{store}\ (\cdots)\ C[(\mathit{inert} \cdots e\ \mathit{inert} \cdots)]) & \rightarrow & (\textbf{store}\ (\cdots)\ C[(\mathit{inert} \cdots e^{\circ}\ \mathit{inert} \cdots)]) & (\text{UMARK}) \\ (\textbf{store}\ (\cdots)\ C[((\textbf{lambda}\ (x \cdots)\ e)^{\circ}\ v^{\circ} \cdots)]) & \rightarrow & (\textbf{store}\ (\cdots\ (x' \ v) \cdots)\ C[e[x' \cdots / \ x \cdots]]) & (\text{UAPP}) \\ & & (\#x = \#v, \text{ each } x' \text{ fresh}) & \\ (\textbf{store}\ (\cdots)\ C[(-^{\circ}\ \ulcorner n \urcorner^{\circ})]) & \rightarrow & (\textbf{store}\ (\cdots)\ C[\ulcorner -n \urcorner]) & (\text{UNEG}) \end{array}$$

Figure 3. Revisions to core Scheme to support unspecified application order

that effect, we introduce the non-terminal *inert* and the notion of a marked expression, denoted with the $\circ$ superscript. (These marks are not an extension to the general term-rewriting framework — e° and C° are just alternate typesettings of (**mark** e) and (**mark** C).) Marks identify chosen expressions: only marked expressions may be reduced, and only one reducible marked expression may appear in any application at one time. The *inert* production stands for terms in which evaluation may not occur, *i.e.*, unmarked expressions (those expressions we have not tried to evaluate yet) and marked values (those expressions we have already finished reducing). We add the UMARK reduction rule that marks an arbitrary unmarked expression in an application on the condition that every other expression is inert, and we modify the MAPP and MNEG rules rules to apply only to fully-marked applications, becoming the UAPP and UNEG rules.

Figure 4 (also generated by PLT Redex) shows how our new system evaluates the term from figure 2. The initial term appears in the center on the left. That term is an application, so the first reduction either marks the first sub-expression or the second. If the first subexpression is marked, evaluation continues down to the bottom of the figure, over to the right and back up to the middle. If the second is marked, evaluation proceeds up, over, and back to the middle. In both paths there are a few other application expressions to evaluate, leading to smaller separations. Eventually, all of the terms join back together and the final result in the store is *1*, as shown in the center on the right.

One should not take that example to mean that this language has any kind of confluence property, however. Consider this program:

```
((lambda (choice)
   ((lambda (x y) choice)
    (set! choice 1)
    (set! choice 2)))
 0)
```

It will either will return either *1* or *2*, depending on the order of evaluation. This is the way we want it; the model's nonconfluence reflects the underspecification of R^{5}RS Scheme rather than a technical bug in our model. It does, however, always make progress. We formalize this with the following theorem statement:

THEOREM 2.1. *For any closed program p in the language of figure 3, either $p \rightarrow p'$, where p' is also closed, $p \rightarrow e$ where e is some error indicator, or p is of the form* (**store** *((x v) ⋯) v).*

Proof is contained in the first author's master's thesis [16].

This technique has other uses besides giving semantics for unspecified application evaluation orders. In general, it is useful for modeling any kind of delimited nondeterminism, where evaluation may proceed arbitrarily but only at certain points in a program. This is useful for modeling unspecified behaviors and for complex nondeterministic features such as threads.

3. Multiple return values

R^{5}RS Scheme provides a facility for expressions to evaluate to multiple or no values rather than just a single value. The procedure *values* builds multiple values and *call-with-values* accepts multiple values. Unlike tuples in SML and Haskell, multiple values are not themselves values. For example, this program

```
(define (f x) (values (+ x x) (* x x)))
(define (g x y) y)
(g (f 3))
```

produces an error, since procedure application expects each of its arguments to be a single value (and the result of f is two values). Instead, the programmer must use *call-with-values* to catch multiple values. It expects a thunk as its first argument, applies the thunk, catches any number of values that thunk produces, and applies them

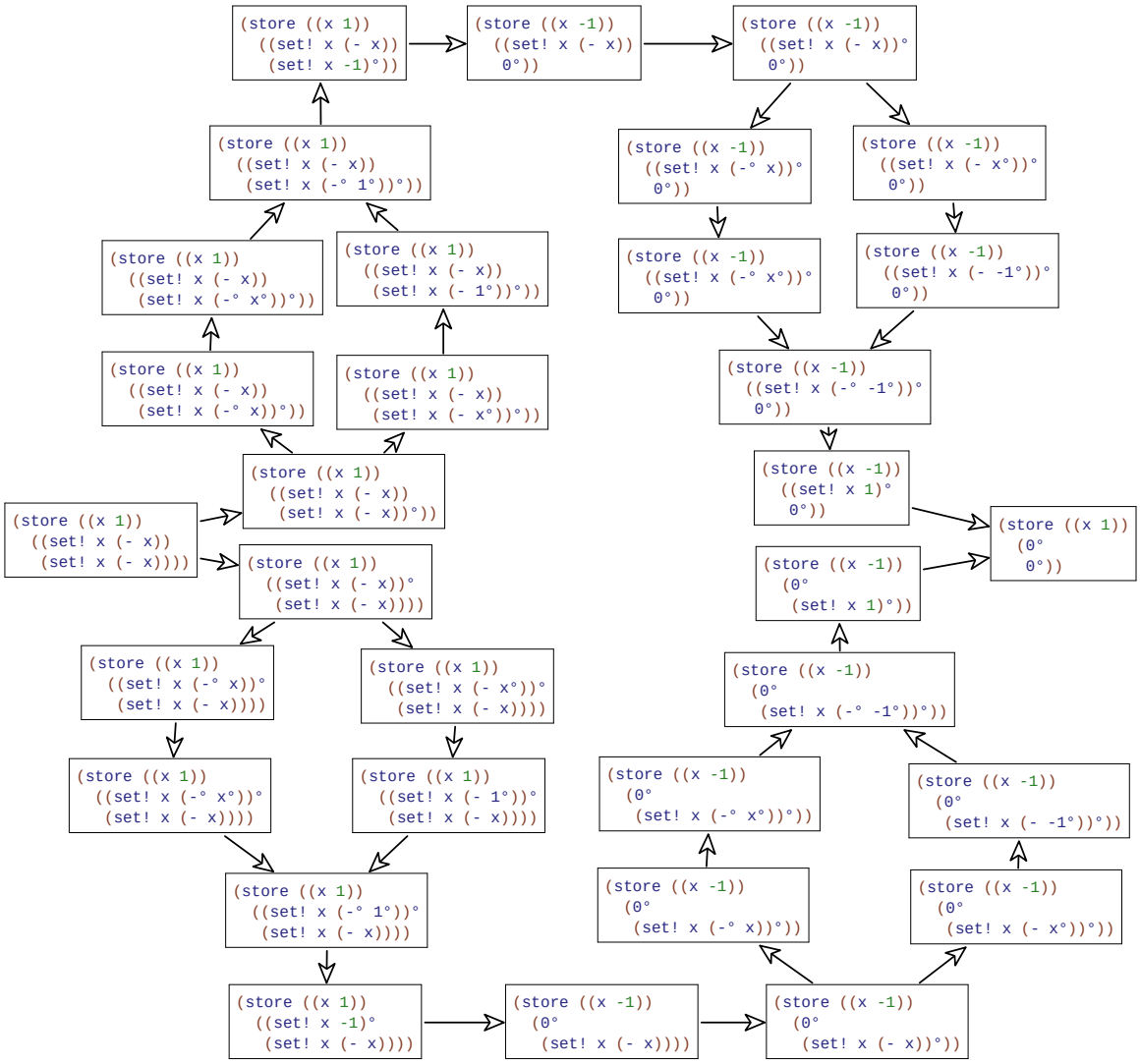


Figure 4. Evaluation in the unspecified-application-order model

to its second argument. So, a programmer could supply *f*'s results to *g* like this:

(*call-with-values* (**lambda** () (*f 3*)) *g*)

In addition, there is no difference between *values* applied to a single argument and that argument by itself, so (*g* (*values 6*) (*values 9*)) is the same as (*g 6 9*).

To model multiple values, R^{5}RS Scheme's formal semantics models continuations as functions from an arbitrary number of values to a final answer. The informal semantics says that "except for continuations created with the *call-with-values* procedure, all continuations take exactly one value" [15, section 6.4]. The formal semantics reflects this by checking the opposite property: in every context that expects a single value, it uses a helper function, *single*, to ensure that only a single value appears. This indirect checking impacts the entire semantics: it requires every continuation to accept any number of arguments initially and requires a call to *single* at every point where a continuation would be restricted.

Our semantic model captures the difference between contexts that accept multiple values and contexts that reject multiple values directly. Our strategy is distilled in figure 5. That figure contains a pure core Scheme extended with *values*, and **apply-values**, a syntactic form that has as its operands an expression that must evaluate to a procedure and another expression that may evaluate to any number of values, and calls the procedure with those values as arguments. We use **apply-values** in this section rather than *call-with-values* because the resulting model is clearer and both **apply-values** and *call-with-values* can be defined simply in terms of each other in R^{5}RS Scheme:

```
(define (call-with-values thunk f)
  (apply-values f (thunk)))

(define-syntax apply-values
  (syntax-rules ()
    [(_ f vs-expr)
     (call-with-values (lambda () vs-expr) f)]))
```

Our model uses a modest addition to the standard reduction-semantics formalism. We extend the notation so that holes have names (written as subscripts) but otherwise behave as unnamed

$$
\begin{array}{lcl}
e & ::= & (e\ e\ \cdots) \mid x \mid v \mid (\textbf{apply-values}\ e\ e) \\
v & ::= & (\textbf{lambda}\ (x\ \cdots)\ e) \mid \mathit{values} \\
C & ::= & [\,]_{\circledast} \mid (v\ \cdots\ C_{\circ}\ e\ \cdots) \mid (\textbf{apply-values}\ C_{\circ}\ e) \mid (\textbf{apply-values}\ v\ C_{*}) \\
C_{\circ} & ::= & [\,]_{\circ} \mid C \\
C_{*} & ::= & [\,]_{*} \mid C
\end{array}
$$

$$
\begin{array}{lcll}
C_{\circ}[((\textbf{lambda}\ (x\ \cdots)\ e)\ v\ \cdots)]_{\circledast} & \rightarrow & C_{\circ}[e[x\ \cdots \diagup v\ \cdots]] \quad (\#v = \#x) & (\textsc{VApp}) \\
C_{\circ}[((\textbf{lambda}\ (x\ \cdots)\ e)\ v\ \cdots)]_{\circledast} & \rightarrow & \textbf{error:}\ \text{wrong number of arguments} \quad (\#v \neq \#x) & (\textsc{VAppErr}) \\
C_{\circ}[(\textbf{apply-values}\ v_1\ (\mathit{values}\ v_2\ \cdots))]_{\circledast} & \rightarrow & C_{\circ}[(v_1\ v_2\ \cdots)] & (\textsc{VAppVals}) \\
C_{\circ}[v]_{*} & \rightarrow & C_{\circ}[(\mathit{values}\ v)] & (\textsc{VPromote}) \\
C_{\circ}[(\mathit{values}\ v)]_{\circ} & \rightarrow & C_{\circ}[v] & (\textsc{VDemote}) \\
C_{\circ}[(\mathit{values}\ v\ \cdots)]_{\circ} & \rightarrow & \textbf{error:}\ \text{expected a single value} \quad (\#v \neq 1) & (\textsc{VDemoteErr})
\end{array}
$$

Figure 5. Pure core Scheme with multiple values

holes do. The context-matching syntax is now annotated with names as well, restricting legal decompositions to those where the hole has the same name.

In figure 5 we use this feature to give three distinct names to holes, indicated with subscripts. $[\,]_{\circ}$ indicates a hole in which any expression should reduce to an element of v, $[\,]_{*}$ indicates a hole in which any expression should reduce to (*values* $v\ \cdots$), and $[\,]_{\circledast}$ indicates a hole in which either result is acceptable. There are three parallel context nonterminals. The context $C_{\circ}$ produces an element of v, C_{*} produces (*values* $v\ \cdots$), and C might produce either.

Since each subexpression of an application is expected to produce a single value, the evaluation context inside an application is $C_{\circ}$. For the same reason, the evaluation context for the first subexpression of **apply-values** is $C_{\circ}$. The evaluation context for the second subexpression, however, is C_{*} because it is expected to produce multiple values.

Since procedure applications (defined by the VApp and VAppErr reductions) and **apply-values** uses (defined by the VAppVals reduction) may produce a single value or (*values* $v\ \cdots$), they take place in $[\,]_{\circledast}$ holes. VPromote, promotes a single value v to (*values* v). Because of the subscript * on the hole, it applies only when multiple values are expected. VDemote demotes a single value inside *values* to just the value, and VDemoteErr signals an error if *values* does not return exactly one value. These two rules apply only when a *values* expression appears where a single value is expected. All reductions take place in $C_{\circ}$ to ensure that the final result of any program is a single value. If we wanted to allow any number of values as the final result of a program we could replace $C_{\circ}$ with C_{*} in all rules.

To get a sense of how evaluation proceeds, consider this reduction sequence:

```
    ((lambda (y) y)
     (apply-values (lambda (x) (values x)) 1))
→   ((lambda (y) y)
     (apply-values (lambda (x) (values x))
                   (values 1)))                    (VPromote)
→   ((lambda (y) y)
     ((lambda (x) (values x)) (values 1)))         (VAppVals)
→   ((lambda (y) y)
     ((lambda (x) (values x)) 1))                  (VDemote)
→   ((lambda (y) y) (values 1))                   (VApp)
→   ((lambda (y) y) 1)                            (VDemote)
→   1                                             (VApp)
```

First, the VPromote applies and promotes 1 into (*values 1*) because it appears as the second argument of an **apply-values** expression. Then VAppVals applies, followed by VApp. Then the term (*values 1*) is used as an argument to a procedure, so VDemote applies and converts it to the single value *1*. Finally VApp applies and the result is *1*.

The erroneous expression from the beginning of this section signals an error due to the VDemoteErr rule.

```
    (g (f 3))
→ ···
→ (g (values 3 9))
→ error: expected a single value
```

The evaluation contexts and the three promotion and demotion rules are all that we need to add multiple values to the language. Furthermore, the extension of adding names to holes does not significantly complicate proof of progress for the system, and so we can prove the following theorem reasonably straightforwardly [16]:

Theorem 3.1. *For any closed program p in the language of figure 5, either $p \rightarrow p'$, where p' is also closed, $p \rightarrow e$ where e is an error indicator, or p is of the form (***store** *((x v) ···) v).*

Proof is contained in the first author's master's thesis [16].

The strategy described in this section can be used whenever the notion of a fully-evaluated subterm is different in different parts of a program. For instance, it can be used to model embedded sublanguages such as regular-expressions, format strings, and SQL commands, which could help develop theoretical underpinnings for work like Herman and Meunier's static analysis of embedded languages [14].

4. Quote and Eval

Scheme inherits the meta-programming facilities *eval* and **quote** from Lisp [22]. The **quote** operator turns a program into data and the *eval* procedure turns that data back into a program. When quoted, a program is represented as a list of lists and symbols, where lists represent parenthesized sequences and symbols represent identifiers. For example, (**quote** (*lambda* (*x*) *x*)) is a three el-

$$
\begin{array}{llll}
e & ::= & (e\ e\ \cdots) \mid v \mid x & \\
E & ::= & [\,] \mid (v \cdots E\ e \cdots) & \\
v & ::= & (\textbf{lambda}\ (x \cdots)\ e) \mid (\textbf{quote}\ sy) & \\
 & \mid & p \mid null \mid n \mid prim \mid \#t \mid \#f & \\
prim & ::= & eval \mid cons \mid car \mid cdr \mid eqv? & \\
p & ::= & \text{pointers} & \\
x & ::= & \text{program variables} & \\
 & & \text{(members of } sy \text{ except } \textbf{lambda}, \textbf{quote}, \textbf{ccons}) &
\end{array}
\qquad
\begin{array}{llll}
s & ::= & (s \cdots) \mid n \mid sy & \\
 & \mid & (s \cdots \textbf{dot}\ sy) \mid (s \cdots \textbf{dot}\ n) & \\
S & ::= & [\,] \mid (e \cdots S\ s \cdots) & \\
 & \mid & (\textbf{lambda}\ (x \cdots)\ S) & \\
 & \mid & (\textbf{ccons}\ v\ S) \mid (\textbf{ccons}\ S\ s) & \\
n & ::= & \text{numbers} & \\
sf & ::= & (p\ (cons\ v\ v)) & \\
sy & ::= & \text{names of symbols} & \\
 & & \text{(identifiers except } \textbf{dot}) &
\end{array}
$$

$$
\begin{array}{lcll}
(\textbf{store}\ (sf_1 \cdots)\ E[(cons\ v_1\ v_2)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_1\ v_2)))\ E[p]) & \text{(ECons)} \\
 & & (p \text{ fresh}) & \\
(\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[(car\ p)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[v_a]) & \text{(ECar)} \\
(\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[(cdr\ p)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[v_d]) & \text{(ECdr)} \\
(\textbf{store}\ (sf_1 \cdots)\ E[(eqv?\ p\ p)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots)\ E[\#t]) & \text{(EEqv1)} \\
(\textbf{store}\ (sf_1 \cdots)\ E[(eqv?\ p_1\ p_2)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots)\ E[\#f]) & \text{(EEqv2)} \\
 & & (p_1 \neq p_2) & \\
(\textbf{store}\ (sf \cdots)\ E[((\textbf{lambda}\ (x \cdots)\ e)\ v \cdots)]) & \rightarrow & (\textbf{store}\ (sf \cdots)\ E[e[x \cdots / v \cdots]]) & \text{(EApp)} \\
 & & (\#x = \#v) & \\
 & & & \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{quote}\ sexp_1\ sexp_2 \cdots)] & \rightarrow & (\textbf{store}\ (sf \cdots)\ S[(\textbf{ccons}\ sexp_1\ (\textbf{quote}\ sexp_2)]) & \text{(EQuoteSeq)} \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{quote}\ ())] & \rightarrow & (\textbf{store}\ (sf \cdots)\ S[null]) & \text{(EQuoteNull)} \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{quote}\ n)] & \rightarrow & (\textbf{store}\ (sf \cdots)\ S[n]) & \text{(EQuoteNum)} \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{ccons}\ v_1\ v_2]) & \rightarrow & (\textbf{store}\ (sf \cdots (p\ (cons\ v_1\ v_2)))\ S[p]) & \text{(EQuotePair)} \\
 & & (p \text{ fresh}) & \\
 & & & \\
(\textbf{store}\ (sf \cdots)\ E[(eval\ v)]) & \rightarrow & (\textbf{store}\ (sf \cdots)\ E[\mathscr{R}\ [\![\ (sf \cdots), v\ ]\!]]) & \text{(EEval)}
\end{array}
$$

$$
\begin{array}{l}
\mathscr{R} : (p \mapsto (cons\ v\ v)) \times v \rightarrow s \\
\mathscr{R}\ [\![\ S, null\ ]\!] = () \\
\mathscr{R}\ [\![\ S, n\ ]\!] = n \\
\mathscr{R}\ [\![\ S, \#t\ ]\!] = \#t \\
\mathscr{R}\ [\![\ S, \#f\ ]\!] = \#f \\
\mathscr{R}\ [\![\ S, (\textbf{quote}\ sy)\ ]\!] = sy \\
\mathscr{R}\ [\![\ S, p\ ]\!] = \mathscr{C}\ [\![\ \mathscr{R}\ [\![\ v_a\ ]\!], \mathscr{R}\ [\![\ v_d\ ]\!]\ ]\!] \\
\qquad \text{where } S \text{ binds } p \text{ to } (cons\ v_a\ v_d)
\end{array}
\qquad
\begin{array}{l}
\mathscr{C} : s \times s \rightarrow s \\
\mathscr{C}\ [\![\ sexp_1, (sexp_2 \cdots)\ ]\!] = (sexp_1\ sexp_2 \cdots) \\
\mathscr{C}\ [\![\ sexp_1, n\ ]\!] = (sexp_1\ \textbf{dot}\ n) \\
\mathscr{C}\ [\![\ sexp_1, sy\ ]\!] = (sexp_1\ \textbf{dot}\ sy) \\
\mathscr{C}\ [\![\ sexp_1, \#t\ ]\!] = (sexp_1\ \textbf{dot}\ \#t) \\
\mathscr{C}\ [\![\ sexp_1, \#f\ ]\!] = (sexp_1\ \textbf{dot}\ \#f)
\end{array}
$$

Figure 6. Core Scheme, extended with eval and quote

ement list whose first and third elements are symbols and whose second element is a list of one element:

```
(cons (quote lambda)
      (cons (cons (quote x) null)
            (cons (quote x) null)))
```

R^{5}RS suggests (but does not require) that quoted data be allocated only once, before the program runs. In systems with that behavior (including all Scheme implementations we tested), this program returns *#t*:

```
((lambda (f) (eqv? (f) (f)))
 (lambda () (quote (x))))
```

since the thunk passed as *f* returns the same result each time it is called.

Our core Scheme calculus for modeling *eval* and **quote** is shown in figure 6. (Note that this model simplifies R^{5}RS Scheme's *eval* procedure in that it does not accept an environment argument.) To ensure that a datum behind a **quote** is inserted into the store only once, the rewriting system is structured in two tiers roughly corresponding to "compile-time" and "run-time." Initially, programs are just viewed as uncompiled s-expressions (elements of the *s* nonterminal; note that we write dotted pairs with **dot** rather than a period to avoid meta-circular confusion in our PLT Redex implementation), which in particular include programs with quoted lists. Reduction rules that apply to these uncompiled expressions do not evaluate them, but instead compile them into program expressions that do not contain quoted lists (elements of the *e* nonterminal). Evaluation reductions only apply to a program after it has been completely compiled.

Each program consists of a store and an expression. Program expressions (*e*) can be applications, values, or identifiers. Evaluation contexts (*E*) dictate that evaluation takes place in a left to right order inside application expressions. The values (*v*) are procedures, quoted symbols, pointers (to cons cells), null, numbers, primitive operations, and booleans.

The first group of evaluation rules (from ECons to EApp) correspond to the language's runtime semantics, and show how the list primitives and procedure application behave. ECons models the application of *cons* to arguments by allocating a new pair in the store; and *car* and *cdr* select the first and second values in a pair by rules ECar and ECdr. EEqv1 and EEqv2 give *eqv?*'s semantics; it compares pointers for literal syntactic equality (and, for this language, operates only on pairs). As in the previous systems we have presented, procedure application is modeled by rule EApp as substitution. Since each reduction takes place in an *evaluation* (rather than *compilation*) context, they will only apply to programs that are completely compiled.

The second group of rules (from EQuoteSeq to EQuotePair) apply at compile-time and show how to compile a program by rewriting quoted constants into locations in the store. If those rules used the *E* context and quoted s-expressions were legal expressions, **quote** would merely be a short-hand notation for building lists at run-time and the above program would return *#f*, which would not capture our intended semantics.

$$
\begin{array}{llll}
p & ::= & (\textbf{store}\ ((x\ v)\cdots)\ (\textbf{dw}\ (dws\cdots)\ e)) & \\
e & ::= & \ldots \mid (\textbf{push}\ (x\ e\ e)) \mid (\textbf{pop}) & \\
v & ::= & \ldots \mid \textit{dynamic-wind} \mid \textit{call/cc} & \\
dws & ::= & (x\ e\ e) & \\
\\
PC & ::= & (\textbf{store}\ ((x\ v)\cdots)\ DC) & \\
DC & ::= & (\textbf{dw}\ ((dws\cdots)\ C)) & \\
C & ::= & \text{(as in figure 1)} &
\end{array}
$$

$$
\begin{array}{lcll}
PC[(\textit{dynamic-wind}\ (\textbf{lambda}\ ()\ e_1) & \rightarrow & PC[(\textbf{begin}\ e_1 & \text{(DWWind)} \\
\quad (\textbf{lambda}\ ()\ e_2) & & \quad (\textbf{push}\ (x_1\ e_1\ e_3)) & \\
\quad (\textbf{lambda}\ ()\ e_3))] & & \quad ((\textbf{lambda}\ (x_2) & \\
 & & \quad\ (\textbf{begin}\ (\textbf{pop})\ e_3\ x_2)) & \\
 & & \quad\ e_2))] & \\
 & & (x_1, x_2 \text{ fresh}) & \\
PC[(\textbf{dw}\ (dws\cdots)\ C[(\textbf{push}\ x_2\ e_1\ e_2)])] & \rightarrow & PC[(\textbf{dw}\ (dws\cdots(x_2\ e_1\ e_2))\ C[\textit{0}])] & \text{(DWPush)} \\
PC[(\textbf{dw}\ (dws_1\cdots dws_n)\ C[(\textbf{pop})])] & \rightarrow & PC[(\textbf{dw}\ (dws_1\cdots)\ C[\textit{0}])] & \text{(DWPop)} \\
PC[(\textbf{dw}\ (dws_1\cdots)\ C[(\textit{call/cc}\ v_1)])] & \rightarrow & PC[(\textbf{dw}\ (dws_1\cdots) & \text{(DWCallcc)} \\
 & & \quad C[(v_1\ (\textbf{lambda}\ (x) & \\
 & & \quad\ (\textbf{throw}\ (dws_1\cdots)\ C[x])))])] & \\
 & & (x \text{ fresh}) & \\
PC[(\textbf{dw}\ (dws_1\cdots)\ C[(\textbf{throw}\ (dws_2\cdots)\ e_1)])] & \rightarrow & PC[(\textbf{dw}\ (dws_2\cdots) & \text{(DWThrow)} \\
 & & \quad C[(\textbf{begin}\ \mathscr{T}[\![\ (dws_2\cdots), (dws_1\cdots)\ ]\!] & \\
 & & \quad\ e_1)])) &
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{T}[\![\ ((x_1\ e_1\ e_2)\ dws_1\cdots), ((x_1\ e_3\ e_4)\ dws_2\cdots)\ ]\!] & = & \mathscr{T}[\![\ (dws_1\cdots), (dws_2\cdots)\ ]\!] \\
\mathscr{T}[\![\ ((x_1\ e_1\ e_2)\cdots), ((x_2\ e_3\ e_4)\cdots)\ ]\!] & = & (\textbf{begin}\ e_2\cdots_r e_3\cdots) \\
 & & (x_1 \neq x_2)
\end{array}
$$

Figure 7. Additions to figure 1 to support call/cc and dynamic-wind

Instead, the second group of rewriting rules eliminate **quote**, turning s-expressions into Scheme programs. Though we have presented them second, these rules will actually come first in reduction sequences, making reduction sequences follow a two-phase pattern where the EQUOTE rules apply in the first phase and the evaluation rules apply in the second phase. Intuitively, programs in this first phase are arbitrary s-expressions and values are Scheme programs, whereas second-phase programs are Scheme expressions and values are Scheme values. This parallelism can be seen particularly clearly in the definition of the evaluation contexts for application expressions. In *S*, a rewrite may occur once all of the s-expressions to the left have become Scheme programs. In *E*, a rewrite may occur once all of the expressions to the left have become values. So, for the program above, the only rewriting rules that apply are those that rewrite the thunk's body. Once it contains only a pointer to a store value, the outer application can proceed.

To model *eval*, we use a technique similar to Muller's *reify* [18]. The $\mathscr{R}$ metafunction accepts a value and turns it back into a program (the $\mathscr{C}$ function is used by $\mathscr{R}$; it is just the syntactic analogue of *cons*). Once $\mathscr{R}$ completes, evaluation continues as usual. Of course, reification may produce an s-expression containing **quote**. In that case, the quote rules apply and put quoted date into the store before evaluation continues.[1]

[1] Most Scheme systems share quoted data even across calls to eval. For example, our semantics produces *#f* for the following program, but most Schemes produce *#t*.

```
((lambda (f)
   (eqv? (f)
         (eval (cons 'quote (cons (f) '())))))
 (lambda () '(x)))
```

We can adapt the definition of $\mathscr{R}$ to handle this by special handling of quoted forms during reification:

$\mathscr{R}[\![\ S, p_1\ ]\!] = v$ if S maps p_1 to (*cons* (**quote** *quote*) p_2) and maps p_2 to (*cons* v '()).

which causes our semantics to produce *#t* for the above example, but this technique does not scale to a full Scheme that includes macros.

As mentioned above, the *eval* we present here and in section 6 is not as full-featured as the *eval* of the R^{5}RS informal description because it does not accept an environment argument. Modeling an *eval* that took an environment argument would be somewhat more involved but would essentially require only running *eval*ed programs in an alternate store.

The technique used in this section applies generally to languages in which computation of a term proceeds in multiple phases that must be considered together — it is not sufficient in our case to write a preprocessor that moves quoted data in a program into the store because that program could call *eval* at runtime. Scheme's macros are similar in this respect, so the technique shown here could be used as a basis for modeling them. Staged and partial evaluation could also be modeled using this technique.

5. Call/cc and dynamic-wind

Scheme's *dynamic-wind* feature for annotating the dynamic extent of a procedure call with entry and exit code that run whenever the program flows into or out of that extent, either through normal program evaluation or through the invocation of continuation objects made by *call/cc* (the latter situation being the more interesting one, of course). Unfortunately, though *dynamic-wind* has a large impact on the meaning of continuation objects *call/cc* produces, the R^{5}RS formal semantics does not include any mention of it and models *call/cc* without respect to it. Here we will show how it works in the context of the core Scheme with mutation presented in section 2. Our strategy for modeling these new features is based heavily on earlier treatments [4, 10, 12].

The language in figure 7 consists of the core Scheme with mutation as shown in figure 1 augmented with *call/cc* and *dynamic-wind*. The basic strategy we take is to maintain a stack of all *dynamic-wind* calls entered but not yet exited, which we call the dynamic-wind stack. When we capture a continuation, we record the current dynamic-wind stack. When we throw to a continuation object, we use the difference between the current dynamic-wind stack and that recorded dynamic-wind stack to determine which *pre* and *post* thunks need to be called.

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ ((ptr\ sv)\cdots)\ (\textbf{dw}\ (dws\cdots)\ e)) \\
e & ::= & (e\ e\cdots) \mid (\textbf{if}\ e\ e\ e) \mid (\textbf{if}\ e\ e) \mid (\textbf{set!}\ x\ e) \mid (\textbf{begin}\ e\ e\cdots) \\
 & \mid & (\textbf{throw}\ x\ dws\cdots \mathrm{EC}[e]) \mid (\textbf{push}\ (x\ e\ e)\ e) \mid (\textbf{pop}\ e) \\
 & \mid & lam \mid mulam \mid v \mid x \\
lam & ::= & (\textbf{lambda}\ (x\cdots)\ e\ e\cdots) \\
mulam & ::= & (\textbf{lambda}\ (x\cdots \textbf{dot}\ x)\ e\ e\cdots) \\
v & ::= & fun \mid nonfun \\
fun & ::= & cp \mid mp \mid \#\%cons \mid \#\%null? \mid \#\%pair? \\
 & \mid & \#\%car \mid \#\%cdr \mid \#\%set{-}car! \mid \#\%set{-}cdr! \mid \#\%list \\
 & \mid & \#\%+ \mid \#\%- \mid \#\%/ \mid \#\%* \mid \#\%call/cc \\
 & \mid & \#\%dynamic{-}wind \mid \#\%values \mid \#\%call{-}with{-}values \\
 & \mid & \#\%eqv? \mid \#\%apply \mid \#\%eval \\
nonfun & ::= & pp \mid number \mid \#\%null \mid \#t \mid \#f \\
 & \mid & (\textbf{quote}\ symbol) \mid unspecified \\
 & & \\
\mathrm{PC} & ::= & (\textbf{store}\ ((ptr\ sv)\cdots)\ \mathrm{DC}) \\
\mathrm{DC} & ::= & (\textbf{dw}\ (dws\cdots)\ \mathrm{EC}_\circ) \\
\mathrm{EC} & ::= & [] \mid (inert\cdots \mathrm{EC}_\circ{}^\circ\ inert\cdots) \\
 & \mid & (\textbf{if}\ \mathrm{EC}_\circ\ e\ e) \mid (\textbf{if}\ \mathrm{EC}_\circ\ e) \mid (\textbf{set!}\ x\ \mathrm{EC}_\circ) \\
 & \mid & (\textbf{begin}\ \mathrm{EC}\ e\ e\cdots) \\
 & \mid & (\#\%call{-}with{-}values^\circ\ (cwv\text{-}mark\ \mathrm{EC}_*)\ v^\circ) \\
\mathrm{EC}_\circ & ::= & [\,]_\circ \mid \mathrm{EC} \\
\mathrm{EC}_* & ::= & [\,]_* \mid \mathrm{EC} \\
inert & ::= & e \mid v^\circ
\end{array}
$$

$$
\begin{array}{lcl}
dws & ::= & (x\ cp\ cp) \\
sv & ::= & v \mid (\#\%cons\ v\ v) \mid lam \mid mulam \\
s & ::= & (s\cdots) \mid (s\cdots \textbf{dot}\ nss) \mid nss \\
nss & ::= & number \mid \#t \mid \#f \mid [\text{variable except } \textbf{dot}] \\
\mathrm{SC} & ::= & [] \mid (e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{if}\ \mathrm{SC}\ s\ s) \mid (\textbf{if}\ e\ \mathrm{SC}\ s) \mid (\textbf{if}\ e\ e\ \mathrm{SC}) \\
 & \mid & (\textbf{if}\ \mathrm{SC}\ s) \mid (\textbf{if}\ e\ \mathrm{SC}) \\
 & \mid & (\textbf{set!}\ x\ \mathrm{SC}) \\
 & \mid & (\textbf{begin}\ \mathrm{SC}\ s\cdots) \mid (\textbf{begin}\ e\ e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{throw}\ x\ dws\cdots \mathrm{SC}) \mid (\textbf{push}\ (x\ \mathrm{SC}\ s)\ s) \\
 & \mid & (\textbf{push}\ (x\ e\ \mathrm{SC})\ s) \mid (\textbf{push}\ (x\ e\ e)\ \mathrm{SC}) \mid (\textbf{pop}\ \mathrm{SC}) \\
 & \mid & (\textbf{lambda}\ (x\cdots)\ \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{lambda}\ (x\cdots)\ e\ e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{lambda}\ (x\cdots \textbf{dot}\ x)\ \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{lambda}\ (x\cdots \textbf{dot}\ x)\ e\ e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{ccons}\ \mathrm{SC}\ s) \mid (\textbf{ccons}\ v\ \mathrm{SC}) \\
 & & \\
var & ::= & [\text{variable except } \textbf{dot} \text{ and keywords}] \\
x & ::= & [\text{variable names}] \\
pp & ::= & [\text{pair pointers}] \\
cp & ::= & [\text{closure pointers}] \\
mp & ::= & [\mu \text{ closure pointers}] \\
ptr & ::= & x \mid pp \mid cp \mid mp
\end{array}
$$

Figure 8. Grammar for full Scheme semantics

That strategy is formally encoded in three parts. First, we add a dynamic-wind stack to each program context. It contains one dynamic context frame (*dws*) for each annotated dynamic extent in which the current evaluation is taking place. A dynamic context frame is a triple consisting of a unique identifier and the *pre* and *post* thunks of the corresponding *dynamic-wind* call. The unique identifier allows us to disambiguate multiple dynamic evaluations of the same syntactic appearance of a *dynamic-wind* expression. Second, we add the primitive procedure value *dynamic-wind* to the set of values, which expects each of its three arguments to evaluate to a thunk. Then using the DWWIND rule it invokes its *pre* thunk, pushes a dynamic context frame onto the stack with a fresh identifier and its own *pre* and *post* thunks, evaluates its second thunk, pops its dynamic context frame off the stack, evaluates its *post* thunk, and finally returns the value its second thunk evaluated to. To allow the program to manipulate the stack, we introduce the **push** and **pop** forms and their associated reduction rules DWPUSH and DWPOP. The former pushes a new dynamic context frame onto the end of the stack, and the latter pops the last context frame off the stack (and then evaluates to the trivial value *0*, which is never used). These two forms are intended to be used only by *dynamic-wind*, never by the programmer directly.

The third piece is *call/cc*. When *call/cc* is called, the DWCALLCC rule builds a continuation object that consists of a procedure of one argument, a fresh identifier we will call *x*. That procedure's body is a **throw** form that consists of the current dynamic stack and the expression formed by plugging *x* into the hole of the evaluation context where the application of *call/cc* itself was found. A **throw** form is itself evaluated using the DWTHROW rule by discarding the evaluation context in which it was found, replacing the dynamic stack with its own stored dynamic stack, and replacing the entire program body with a specially-constructed **begin** expression built by the $\mathscr{T}$ metafunction (where T stands for "trim," because it trims away the common context frames leaving only the suffixes whose pre- or post-thunks need to be executed). That function compares its first argument, the dynamic-wind stack of the dynamic context being exited, with its second argument, the dynamic-wind stack of the context being entered. The first rule in its definition simply discards any common prefix the two stacks may have, which correspond to dynamic extents that were never left or entered during the transitions from the time the continuation object was created and the time it was invoked. Then, once the two stacks have been trimmed to the point where they have distinct heads, the metafunction produces a **begin** expression consisting of applications of all the *post* thunks from $\mathscr{T}$'s first argument, invoked in order, followed by all the *pre* thunks from $\mathscr{T}$'s second argument, invoked in reverse order (which we indicate with the special notation $\cdots_r$, indicating a sequence being expanded out backwards).

6. Operational semantics for R^{5}RS Scheme

This section combines the techniques from sections 2 through 5 with other known techniques for modeling programming languages to build a model of R^{5}RS Scheme that includes all the features from those sections along with **if** and booleans, mathematical operations (but not the numeric tower), list constructors, selectors, mutators and predicates, μ-lambda procedures[2], *apply*, and object identity-based equivalence. Although this section appears large and complex at first, it is mostly just a simple combination of the previous four sections.

This specification is executable, and the figures presented in this section were automatically generated from the source code that implements the specification. Since an executable specification was an explicit goal of our work, we have made some modeling choices that may not be obvious at first. For example, there are many expressions whose return values are explicitly unspecified in the R^{5}RS Scheme document, such as the result of a **set!** expression. A non-executable specification might model the evaluation of those expressions using the rule schema

$$\forall\ v.\ PC[unspecified] \rightarrow PC[v]$$

[2] Procedures declared with an improper list of formal arguments described in section 4.1.4 of the Report that accept an arbitrary number of arguments beyond a certain minimum. The name dates back at least to Indiana University's Scheme 84 system where `MULAMBDA` was a keyword used to declare procedures that accepted any number of arguments and collected them in a list [11].

meaning that an unspecified term reduces to any value. Instead, we model unspecified results with a special value *unspecified* that has no associated reduction rules and will cause programs that inspect it to get stuck. We also chose to ignore out-of-memory errors. These would be easy to add at the expense of a additional clutter when visualizing traces: reductions from each allocation site to the out-of-memory error would suffice.

6.1 Grammar

The grammar for R^{5}RS Scheme programs is given in figure 8. In that figure, a program (given by the *p* nonterminal) consists of a store, a dynamic-wind stack, and an expression. The *e* nonterminal gives expressions, which in addition to standard Scheme core forms can be **throw**, **push** and **pop**, as in section 5. Values (*v*) are either procedures or non-procedure values, but notice that syntactic **lambda** terms are not values themselves. Instead, procedure values (*fun*) can be references to procedures in the store (*cp* and *mp*) or the built-in procedures, while the **lambda** form, as we will see, places new procedure values into the store when evaluated. Non-procedure values (*nonfun*) include pair pointers, numbers, *null*, booleans, symbols, and the unspecified value.

As in section 4, we write dotted pairs (as in the parameter list of a μ-lambda) with **dot** rather than a period to avoid meta-circular confusion in our PLT Redex implementation.

Section 6 of the R^{5}RS Scheme specification indicates that primitive procedures are bound to names in the initial environment, but that those names can be mutated during the course of a program. To model that, we use special names with *#%* prefixes to indicate the actual built-in procedures, and we bind those values to their *#%*-less names in the initial store:

(**store** ((*list #%list*) (*cons #%cons*) (*car #%car*) (*cdr #%cdr*)
(*pair? #%pair?*) (*null #%null*) (*null? #%null?*)
(*set-car! #%set−car!*) (*set-cdr! #%set−cdr!*)
(+ *#%+*) (− *#%−*) (/ *#%/*) (∗ *#%∗*)
(*call/cc #%call/cc*) (*dynamic-wind #%dynamic−wind*)
(*values #%values*) (*call-with-values #%call−with−values*)
(*eqv? #%eqv?*) (*apply #%apply*) (*eval #%eval*)) · · ·)

There are three different contexts we will make use of: program evaluation contexts, dynamic-wind contexts, and expression contexts. Each program evaluation context (PC) contains a store, and a dynamic-wind context. Each dynamic-wind context (DC) contains a dynamic-wind stack and an expression context. Expression contexts (EC) are the contexts in which program evaluation takes place; they allow evaluation in marked sub-expressions of an application (as in section 2), the test positions of **if** expressions, in **set!** expressions and in the first position in a **begin** (as long as there are at least two expressions in the **begin**). The evaluation context for *#%call−with−values* is explained in section 6.7. The $\text{EC}_\circ$ and EC_* evaluation contexts and *inert* work like $C_\circ$ and C_* and *inert* from section 3.

The *dws* non-terminal corresponds to one frame of dynamic-wind context information and its use is explained in section 5. The *sv* non-terminal generates values that appear in the store.

S-expressions (*s* and *nss*) and s-expression contexts (SC) correspond to s-expressions and s-expression contexts from section 4. There are more possible s-expression contexts in the full language because there are more possible syntactic forms.

Finally, the *x* nonterminal represents both program variables and binding locations, and the *pp*, *cp*, and *mp* nonterminals represent pointers to pairs, fixed-arity procedures, and variable-arity procedures, respectively. The *ptr* non-terminal is a short-hand for terms that index into the store. One subtle point here is that the *v* production produces *pp*, *cp*, and *mp* but not *x*. Those variables are not included because free variables are not values and bound variables have to be dereferenced before use, so neither qualifies as an irreducible value.

6.2 Relations

In the remaining figures, we will make heavy use of various reduction relation symbols. The basic reduction relation we will use is $\rightarrow$, which indicates that the program term on the left reduces in one step to the term on the right. We also use two other relations to aid in the system's readability, defined in terms of the $\rightarrow$ relation:

- $(e_1 \cdots e_n) \mapsto^\circ e'$ iff $\text{PC}[(e_1^\circ \cdots e_n^\circ)] \rightarrow \text{PC}[e']$
  The application on the left reduces to the term on the right in a program context, assuming that all of the expressions in the application are marked.
- $e \rightarrow^e$ **error:** *s* iff $\text{PC}[e] \rightarrow$ **error:** *s*
  The term on the left signals an error, halting the program immediately.

6.3 Basic syntactic forms

Figure 9 shows rules for the basic syntactic forms. For the **if** form, if the test position evaluates to anything other than *#f*, the term rewrites to its "then" subexpression. If the test position evaluates to *#f*, it rewrites to its "else" subexpression, if present, *unspecified* otherwise. For the **begin** form, the evaluation contexts defined in figure 8 ensure that the first term of a **begin** expression containing at least two expressions is evaluated fully; then these rules cause **begin** expression that consists of a fully-evaluated value followed by one or more expressions to rewrite to a new **begin** expression with the initial value dropped. These rules also specify that a **begin** form with only a single expression reduces immediately to that expression, even if that expression is not yet a value.

Because our model does not take into account R^{5}RS Scheme's numeric tower, we model its numeric operations in terms of true mathematical functions. We assume that we can identify the true number represented by each numeric term and model each numeric procedure by performing the appropriate mathematical operation on those true numbers: + is modeled by summation on the represented numbers, ∗ is modeled by multiplication, and so on.

6.4 Cons and cons-cell mutation

The rules for constructing new *cons* cells are given in figure 10. Since all cons cells are mutable and therefore can be distinguished even when they hold identical values, we cannot allow (*#%cons v v*) to be a value itself. Instead, the *#%cons* rule introduces a new pair into the store and reduces to a pointer to that new pair. The (*#%list* $v_1 \cdots$) rule rewrites to ((**lambda** *x x*) $v_1 \cdots$), taking advantage of the μ-lambda application rules described in the next subsection.

Figure 11 gives rules for *car* and *cdr*. Application of either procedure to a pair pointer rewrites to the contents of the appropriate field in the pair being pointed to. If either selector is applied to a non-pair value, the term rewrites to an error message.

The predicates in figure 12 are similarly simple. The *#%pair?* procedure reduces to *#t* if its argument is identifiable as a pair pointer and *#f* otherwise. The *#%null?* procedure reduces to *#t* if and only if it is supplied with the built-in null value.

Figure 13 gives rules for *set-car!* and *set-cdr!*, for cons-cell mutation. The *#%set−car!* and *#%set−cdr!* rules are the same as the *car* and *cdr* rules, respectively, except that instead of reducing to the current value of appropriate component of the pair being pointed to, they replace that component with the given replacement then rewrite to an unspecified value.

6.5 Procedures and assignable variables

The rules in figure 14 handle variable lookup and variable assignment: a binding pointer is replaced with its value in the store when

$$
\begin{array}{lcl}
\text{PC}[(\textbf{if } v_1\ e_1\ e_2)] & \rightarrow & \text{PC}[e_1] \\
 & & (v_1 \neq \#f) \\
\text{PC}[(\textbf{if } \#f\ e_1\ e_2)] & \rightarrow & \text{PC}[e_2] \\
\text{PC}[(\textbf{if } v_1\ e_1)] & \rightarrow & \text{PC}[e_1] \\
 & & (v_1 \neq \#f) \\
\text{PC}[(\textbf{if } \#f\ e_1)] & \rightarrow & \text{PC}[\textit{unspecified}]
\end{array}
$$

$$
\begin{array}{lcl}
\text{PC}[(\textbf{begin } v\ e_1\ e_2\ \cdots)] & \rightarrow & \text{PC}[(\textbf{begin } e_1\ e_2\ \cdots)] \\
\text{PC}[(\textbf{begin } e_1)] & \rightarrow & \text{PC}[e_1] \\
(+\ \lceil n \rceil\ \cdots) & \mapsto^\circ & \lceil \Sigma\ n\ \cdots \rceil \\
(-\ \lceil n_1 \rceil\ \lceil n_2 \rceil\ \cdots) & \mapsto^\circ & \lceil n_1 - (\Sigma\ n_2\ \cdots) \rceil \\
(-\ \lceil n \rceil) & \mapsto^\circ & \lceil \text{-}n \rceil \\
(*\ \lceil n \rceil\ \cdots) & \mapsto^\circ & \lceil \Pi\ n\ \cdots \rceil \\
(/\ \lceil n_1 \rceil\ \lceil n_2 \rceil\ \cdots) & \mapsto^\circ & \lceil n_1\ /\ (\Pi\ n_2\ \cdots) \rceil \\
(/\ \lceil n_1 \rceil) & \mapsto^\circ & \lceil 1\ /\ n_1 \rceil
\end{array}
$$

Figure 9. Basic syntactic forms

$$
\begin{array}{lcl}
(\textbf{store } ((ptr_1\ sv_1)\ \cdots) \\
\quad \text{DC}[(\#\%cons^\circ\ v_{car}{}^\circ\ v_{cdr}{}^\circ)]) & \rightarrow & (\textbf{store } ((ptr_1\ sv_1)\ \cdots\ (p_i\ (\#\%cons\ v_{car}\ v_{cdr}))) \\
 & & \quad \text{DC}[p_i]) \\
 & & (p_i\ \textit{fresh}) \\
\text{PC}[(\#\%list^\circ\ v_1{}^\circ\ \cdots)] & \rightarrow & \text{PC}[((\textbf{lambda } (\textbf{dot } l)\ l)^\circ\ v_1{}^\circ\ \cdots)]
\end{array}
$$

Figure 10. List constructors

$$
\begin{array}{lcl}
(\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
\qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
\qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
\quad \text{DC}[(\#\%car^\circ\ pp_i{}^\circ)]) & \rightarrow & (\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
 & & \qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
 & & \qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
 & & \quad \text{DC}[v_{car}]) \\
(\#\%car^\circ\ v_i{}^\circ) & \rightarrow^e & \textbf{error:} \text{ can't take car of non-pair} \\
 & & (v_i \notin pp) \\
(\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
\qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
\qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
\quad \text{DC}[(\#\%cdr^\circ\ pp_i{}^\circ)]) & \rightarrow & (\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
 & & \qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
 & & \qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
 & & \quad \text{DC}[v_{cdr}]) \\
(\#\%cdr^\circ\ v_i{}^\circ) & \rightarrow^e & \textbf{error:} \text{ can't take cdr of non-pair} \\
 & & (v_i \notin pp)
\end{array}
$$

Figure 11. List accessors

$$
\begin{array}{lcl}
(\#\%null?\ \#\%null) & \mapsto^\circ & \#t \\
(\#\%null?\ v_i) & \mapsto^\circ & \#f \\
 & & (v_i \neq \#\%null) \\
(\#\%pair?\ pp) & \mapsto^\circ & \#t \\
(\#\%pair?\ v_i) & \mapsto^\circ & \#f \\
 & & (v_i \notin pp)
\end{array}
$$

Figure 12. List predicates

dereferenced, and mutation of a binding pointer is represented by replacing the value pointed to by the update. **lambda** is the only binding form in this semantics, so the rules for procedure calls are the only ones that introduce new bindings. Procedure calls are modeled by two features: closure introduction and procedure application.

The rules in figure 15 govern the introduction of closure values into the store. Like cons cells, procedures are not values, but pointers to them are; procedures are modeled this way so that we can model *eqv?* more accurately. The allocation rule for fixed-arity procedures is straightforward. The allocation for μ-lambda procedures always puts two procedures into the store: a stub μ-lambda procedure whose body contains a call to an ordinary procedure, and an ordinary procedure that contains the original μ-lambda's body expressions.

The reason for arranging the system this way is so that when a μ-lambda procedure is applied, we can rewrite it into a corresponding call to the fixed-arity code pointer and thereby use the same reduction for both kinds of applications. The rules in figure 16 show this and the rest of the rules for application in detail. The first rule shows how marks are placed in applications, which is just as in section 2. Application of a procedure pointer to arguments is modeled by creating one new binding pointer in the store per formal argument where the value being pointed to by each pointer is the argument supplied in the appropriate position, and rewriting to the procedure's body with these new bound-variable pointers substituted for occurrences of the formal arguments.

Application of a μ-lambda allocates a list for its extra arguments, applies the initial portion of the arguments as usual, and applies the extra arguments into the last argument of the procedure that actually contains the body expressions. The function $\mathcal{L}$ used here is a metafunction that builds the syntax of a *cons*-list from its arguments:

$$
\begin{array}{lcl}
\mathcal{L}\ [\![\ x\ y\ \cdots\ ]\!] & = & (\#\%cons\ x\ \mathcal{L}\ [\![\ y\ \cdots\ ]\!]) \\
\mathcal{L}\ [\![\ ]\!] & = & \#\%null
\end{array}
$$

The last rules specify the behavior of Scheme's *apply* procedure which accepts a procedure and an arbitrary number of arguments, the last of which must be a list. It calls the procedure with the arguments and the contents of the list as subsequent arguments. To model it, the first two *#%apply* rules flatten out the argument list and, when the list is exhausted, reduce to a normal application.

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \quad (pp_i\ (\#\%cons\ v_{new}\ v_{cdr})) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\#\%set{-}car!^{\circ}\ pp_i{}^{\circ}\ v_{new}{}^{\circ})]) & & \ \mathrm{DC}[\textit{unspecified}])
\end{array}
$$

$$
(\#\%set{-}car!^{\circ}\ v_1{}^{\circ}\ v^{\circ}) \to^{e} \textbf{error:}\ \text{can't set-car! on a non-pair } (v_1 \notin pp)
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \quad (pp_i\ (\#\%cons\ v_{car}\ v_{new})) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\#\%set{-}cdr!^{\circ}\ pp_i{}^{\circ}\ v_{new}{}^{\circ})]) & & \ \mathrm{DC}[\textit{unspecified}])
\end{array}
$$

$$
(\#\%set{-}cdr!^{\circ}\ v_1{}^{\circ}\ v^{\circ}) \to^{e} \textbf{error:}\ \text{can't set-cdr! on a non-pair } (v_1 \notin pp)
$$

Figure 13. Cons cell mutation

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (x_i\ sv_i) & & \quad (x_i\ sv_i) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[x_i]) & & \ \mathrm{DC}[sv_i])
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (x_i\ sv_i) & & \quad (x_i\ v_{new}) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\textbf{set!}\ x_i\ v_{new})]) & & \ \mathrm{DC}[\textit{unspecified}])
\end{array}
$$

Figure 14. Variable mutation and lookup

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots) & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots\ (cp_i\ lam_i)) \\
\ \mathrm{DC}[lam_i]) & & \ \mathrm{DC}[cp_i]) \\
 & & (cp_i\ \textit{fresh})
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots) & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\ \mathrm{DC}[(\textbf{lambda}\ (x_1\cdots\ \textbf{dot}\ x_r)\ e_1\ e_2\cdots)]) & & \quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\ \textbf{dot}\ x_r)\ (cp_i\ x_1\cdots x_r))) \\
 & & \quad (cp_i\ (\textbf{lambda}\ (x_1\cdots x_r)\ e_1\ e_2\cdots))) \\
 & & (mp_i,\ cp_i\ \textit{fresh})
\end{array}
$$

Figure 15. Procedure introduction

$$
\mathrm{PC}[(inert_1\cdots e_i\ inert_{i+1}\cdots)] \to \mathrm{PC}[(inert_1\cdots e_i{}^{\circ}\ inert_{i+1}\cdots)]
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (cp_i\ (\textbf{lambda}\ (x_1\cdots)\ e_{body1}\ e_{body2}\cdots)) & & \quad (cp_i\ (\textbf{lambda}\ (x_1\cdots)\ e_{body1}\ e_{body2}\cdots)) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots \\
\ \mathrm{DC}[(cp_i{}^{\circ}\ v_{arg1}{}^{\circ}\cdots)]) & & \quad (x_{arg2}\ v_{arg1})\cdots) \\
 & & \ \mathrm{DC}[(\textbf{begin}\ e_{body1}\ e_{body2}\cdots)[x_1\cdots/x_{arg2}\cdots]])] \\
 & & (\#x_{arg} = \#v_{arg}\ , x_{arg2}\cdots\ \textit{fresh})
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr\ sv)\cdots & \to & \textbf{error:}\ \text{arity mismatch} \\
\quad (cp_i\ (\textbf{lambda}\ (x_1\cdots)\ e\ e\cdots)) & & (\#x_{arg} \neq \#v_{arg}) \\
\quad (ptr\ sv)\cdots) & & \\
\ \mathrm{DC}[(cp_i{}^{\circ}\ v_{arg1}{}^{\circ}\cdots)]) & &
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\ \textbf{dot}\ y)\ (cp_t\ x_1\cdots y))) & & \quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\ \textbf{dot}\ y)\ (cp_t\ x_1\cdots y))) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(mp_i\ v_{n1}{}^{\circ}\cdots v_R{}^{\circ}\cdots)]) & & \ \mathrm{DC}[(cp_t\ v_{n1}{}^{\circ}\cdots\mathcal{L}\ [\![\ v_R{}^{\circ}\cdots\ ]\!]\ )]) \\
 & & (\#x = \#v_n)
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr\ sv)\cdots & \to & \textbf{error:}\ \text{too few arguments} \\
\quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\ \textbf{dot}\ x)\ (cp\ x\cdots))) & & (\#x_{arg} < \#v_{arg}) \\
\quad (ptr\ sv)\cdots) & & \\
\ \mathrm{DC}[(mp_i{}^{\circ}\ v_{arg1}{}^{\circ}\cdots)]) & &
\end{array}
$$

$$
(nonfun^{\circ}\ v^{\circ}\cdots) \to^{e} \textbf{error:}\ \text{can't apply non-function}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\#\%apply^{\circ}\ v_f{}^{\circ}\ v_{arg1}{}^{\circ}\cdots pp_i{}^{\circ})]) & & \ \mathrm{DC}[(\#\%apply^{\circ}\ v_f{}^{\circ}\ v_{arg1}{}^{\circ}\cdots v_{car}{}^{\circ}\ v_{cdr}{}^{\circ})])
\end{array}
$$

$$
(\#\%apply\ v_f\ v_{arg1}\cdots\#\%null) \mapsto^{\circ} (v_f\ v_{arg1}\cdots)
$$

$$
(\#\%apply^{\circ}\ v_f{}^{\circ}\ v_{arg1}{}^{\circ}\cdots v_{last}{}^{\circ}) \to^{e} \textbf{error:}\ \text{apply must take a list as its last argument } (v_{last} \notin pp \cup \{\#\%null\})
$$

Figure 16. Procedure application

(**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[(*#%call/cc*$^\circ$ *v*$_1{}^\circ$)]))
$\rightarrow$ (**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[(*v*$_1$ (**lambda** (**dot** *args*)
(**throw** *x*$_k$ *dws*$_1$ $\cdots$
EC$_1$[(**begin** *x*$_k$ (*#%apply #%values args*))])))])
(x, x$_k$ fresh)

(**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[(*#%dynamic−wind*$^\circ$ *cp*$_1{}^\circ$ *cp*$_2{}^\circ$ *cp*$_3{}^\circ$)]))
$\rightarrow$ (**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[(**begin** (*cp*$_1$)
(**push** (*x*$_1$ *cp*$_1$ *cp*$_3$)
((**lambda** (*x*$_2$) (**pop** (**begin** (*cp*$_3$) *x*$_2$)))
(*cp*$_2$))))]))
(x$_1$, x$_2$ fresh)

Figure 17. Call/cc and dynamic-wind

(**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[(**push** *dws*$_2$ *e*$_{next}$)]))
$\rightarrow$ (**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_2$ *dws*$_1$ $\cdots$)
EC$_1$[*e*$_{next}$]))

(**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ *dws*$_2$ $\cdots$)
EC$_1$[(**pop** *e*$_{next}$)]))
$\rightarrow$ (**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_2$ $\cdots$)
EC$_1$[*e*$_{next}$]))

(**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[(**throw** *x*$_k$ *dws*$_2$ $\cdots$ EC$_2$[*e*$_2$])]))
$\rightarrow$ (**store** ((*ptr*$_s$ *sv*$_s$) $\cdots$)
(**dw** (*dws*$_2$ $\cdots$)
(**begin** $\mathcal{T}$ $[\![$(*dws*$_2$ $\cdots$), (*dws*$_1$ $\cdots$) $]\!]$
EC$_2$[*e*$_2$]))

Figure 18. Call/cc and dynamic-wind support

PC[*v*$_1$]$_*$	$\rightarrow$	PC[(*#%values*$^\circ$ *v*$_1{}^\circ$)]
PC[(*#%values*$^\circ$ *v*$_1{}^\circ$)]$_\circ$	$\rightarrow$	PC[*v*$_1$]
PC[(*#%values*$^\circ$ *v*$_1{}^\circ$ $\cdots$)]$_\circ$	$\rightarrow$	**error:** wrong number of values (*#v*$_1 \neq$ *1*)
(*#%call−with−values v*$_{vals}$ *v*$_{fun}$)	$\mapsto^\circ$	(*#%call−with−values*$^\circ$ (*cwv-mark* (*v*$_{vals}$)) *v*$_{fun}{}^\circ$)
PC[(*#%call−with−values*$^\circ$ (*cwv-mark* (*#%values*$^\circ$ *v*$_{arg}{}^\circ$ $\cdots$)) *v*$_{fun}{}^\circ$)]	$\rightarrow$	PC[(*v*$_{fun}{}^\circ$ *v*$_{arg}{}^\circ$ $\cdots$)]
(*#%call−with−values*$^\circ$ *v*$_i{}^\circ$ $\cdots$)	$\rightarrow^e$	**error:** arity mismatch (*#v*$_i \neq$ *2*)

Figure 19. Multiple values and call-with-values

(*#%eqv? pp*$_i$ *pp*$_i$)	$\mapsto^\circ$	*#t*
(*#%eqv? cp*$_i$ *cp*$_i$)	$\mapsto^\circ$	*#t*
(*#%eqv? number*$_1$ *number*$_1$)	$\mapsto^\circ$	*#t*
(*#%eqv? v*$_1$ *v*$_1$)	$\mapsto^\circ$	*#t*
PC[(*#%eqv?*$^\circ$ *v*$_1{}^\circ$ *v*$_2{}^\circ$)]	$\rightarrow$	PC[*#f*] (*v*$_1 \neq$ *v*$_2$)
(*#%eqv?*$^\circ$ *v*$_1{}^\circ$ $\cdots$)	$\rightarrow^e$	**error:** arity mismatch (*#v*$_1 \neq$ *2*)

Figure 20. Eqv and equivalence

6.6 Call/cc

Our technique for modeling *call/cc* and *dynamic-wind*, shown in figures 17 and 18, is essentially the technique from section 5. Apart from the change of using procedure pointers rather than the literal source text of procedures as required to model equality (see section 6.8), the only substantial change is that the continuation procedures in this model accept any number of arguments. The trimming metafunction $\mathcal{T}$ is the same function defined in section 5.

6.7 Multiple values and call-with-values

Multiple values in the full language are nearly identical to multiple values in section 3, and in particular the context arrangement and promotion and demotion rules are the same. Furthermore, even though the present system is much larger than the system presented in section 3, the rules for multiple values are still completely orthogonal to the rules that implement the other features.

There is one twist, though, since rather than the **apply-values** primitive given in section 3, R^{5}RS Scheme provides *call-with-values*, so we model it directly. To do so, we have to use the mechanisms described in section 3, along with a new context containing *cwv-mark*. A term of the form (*#%call−with−values thunk f*) reduces to (*#%call−with−values* (*cwv-mark* (*thunk*)) *f*); that is, it places a special mark around the application of the thunk to no arguments. At that point the evaluation contexts defined in figure 8 will apply and reduce the applied thunk in a multi-value context. When that reduction sequence yields a result (which will be a multiple-values expression), the entire *call-with-values* expression reduces to the application of the second procedure to those produced values.

6.8 Eqv? and equivalence

Figure 20 shows the rules for *eqv?*. Since all mutable values (and procedures) are allocated in the store, *eqv?* is a simple matter

(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
DC[(*#%eval*° v_1°)])
→
(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
DC[$\mathscr{R}$ ⟦((*ptr*$_1$ *sv*$_1$) $\cdots$), v_1⟧])

(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[(**quote** (s_1 s_2 $\cdots$))]]))
→
(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[(**ccons** (**quote** s_1) (**quote** (s_2 $\cdots$)))]]))

(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[(**quote** ())]]))
→
(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[*#%null*]]))

(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[(**quote** *number*$_1$)]]))
→
(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[*number*$_1$]]))

(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$)
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[(**ccons** v_1 v_2)]]))
→
(**store** ((*ptr*$_1$ *sv*$_1$) $\cdots$ (*pp*$_1$ (*#%cons* v_1 v_2)))
(**dw** (*dws*$_1$ $\cdots$)
EC$_1$[SC$_1$[*pp*$_1$]]))
(*pp*$_1$ *fresh*)

Figure 21. Quote and eval

of checking that the two values supplied have identical syntactic structure (which we indicate here, as PLT Redex does, by repeating the same subscript for both arguments to the *eqv?* procedure to indicate that the two subterms must be identical).

6.9 Quote and eval

The rules for *#%eval* and **quote** in figure 21 are essentially the same as the rules for *eval* and **quote** in section 4. The main difference is that the rewriting rules for replacing quote are nested an SC context inside an EC context. This only matters when using *#%eval*. In particular, if the call to *#%eval* is in some marked context, SC will not match properly, due to the marks. In the smaller calculus, we could get away with just using SC, since it also encompassed evaluation contexts, but here we must be explicit. The reify function ($\mathscr{R}$) used here is as defined in section 4.

7. Related Work

Reduction semantics has been used to model large programming languages many times and in many different ways. Felleisen's dissertation [3], which introduced context-sensitive reduction semantics, gives a formulation of a substantially smaller language than the one we present here that he calls "idealized Scheme," and Felleisen extends that model into the λ-v-CS calculus in later work [4]. Since then, reduction semantics have been used to model the cores of many languages including Emacs Lisp [19], MultiLisp [7], Java [9], ML [13, 24] and Concurrent ML [21] among many others. Harper and Stone present a formal semantics for Standard ML that includes a dynamic semantics encoded using a variation on Wright and Felleisen's notation; it is the largest example of a programming language semantics given in a variant of reduction semantics we have found in the literature (with the possible exception of our own semantics for R^{5}RS Scheme).

There has also been extensive work on the semantics of Scheme. Clinger presented an operational semantics for a core Scheme in the development of the notion of space efficiency [2]. Gasbichler, Knauel, Sperber, and Kelsey have presented operational and denotational semantics for *dynamic-wind* [12]. Ramsdell presented a structural operational semantics for Scheme aimed at fixing the unspecified order of argument evaluation problem we discuss in subsection 2 [20]. His model is less complete than ours (for instance, it does not include multiple return values) and is tied much more closely to the R^{5}RS Scheme formal semantics. Van Straaten has written an interpreter based on the R^{5}RS Scheme denotational semantics [23], but we know of no formal correspondence between his program and the denotational semantics itself.

8. Conclusion

We have presented a semantics for R^{5}RS Scheme using context-sensitive reduction semantics developed using PLT Redex. To the best of our knowledge, it formalizes more of the language than any other semantics for the language. In addition it shows how to model R^{5}RS Scheme-style multiple return values in an small-step operational semantics setting for the first time, and gives a new model for unspecified sequential evaluation orders that uses nondeterministic choice. In the process, we have introduced several new techniques for modeling programming language features with term rewriting.

PLT Redex and the source code for all the models presented in this paper, including our executable model of R^{5}RS Scheme, are available for download at

http://www.cs.uchicago.edu/~jacobm/r5rs/

Acknowledgments

Thanks to Kent Dybvig and Matthew Flatt for helpful discussions of the technical details presented here and the inner workings of Chez Scheme [1] and MzScheme [8]. Thanks also to John Reppy and Dave MacQueen and the anonymous reviewers for their helpful suggestions.

References

[1] Cadence Research Systems. *Chez* Scheme Reference Manual, 1994.

[2] William D Clinger. Proper tail recursion and space efficiency. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.

[3] Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State In Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[4] Matthias Felleisen. Lambda-v-CS: and extended lambda-calculus for Scheme. In *Proceedings of the Conference on LISP and Functional Programming*, 1988.

[5] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Available online: http://www.cs.utah.edu/plt/publications/pllc.pdf, 2003.

[6] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical*

Computer Science, 102:235–271, 1992. Original version in: Technical Report 89-100, Rice University, June 1989.

[7] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9:1–31, 1999.

[8] Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. http://www.plt-scheme.org/software/mzscheme/.

[9] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

[10] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *Proceedings of the ACM Conference Principles of Programming Languages*, 1985.

[11] Daniel P. Friedman, Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. Scheme 84 interim reference manual. Technical Report 153, Indiana University Computer Science, 1985.

[12] Martin Gasbichler, Eric Knauel, Michael Sperber, and Richard A. Kelsey. How to add threads to a sequential language without getting tangled up. In *Proceedings of the 2003 Scheme Workshop*, 2003.

[13] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proceedings of the ACM Conference Principles of Programming Languages*, 1993.

[14] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 16–27, New York, NY, USA, 2004. ACM Press.

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

[16] Jacob Matthews. Operational semantics for Scheme via term rewriting. Technical Report TR-2005-02, University of Chicago, 2005.

[17] Jacob Matthews, Robert Bruce Finder, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*, 2004.

[18] Robert Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4), 1992.

[19] Matthias Neubauer and Michael Sperber. Down with Emacs Lisp: Dynamic scope analysis. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2001.

[20] John D. Ramsdell. An operational semantics for Scheme. *Lisp Pointers*, volume 2, April–June 1992.

[21] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[22] Gerald Jay Sussman and Jr Guy Lewis Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, 1975.

[23] Anton van Straaten. An executable denotational semantics for Scheme. http://www.appsolutions.com/SchemeDS/.

[24] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

Commander S — The shell as a browser

Martin Gasbichler Eric Knauel
Universität Tübingen
{gasbichl,knauel}@informatik.uni-tuebingen.de

Abstract

Commander S is a new approach to interactive Unix shells based on interpretation of command output and cursor-oriented terminal programs. The user can easily refer to the output of previous commands when composing new command lines or use interactive viewers to further explore the command results. Commander S is extensible by plug-ins for parsing command output and for viewing command results interactively. The included job control avoids garbling of the terminal by informing the user in a separate widget and running background processes in separate terminals. Commander S is also an interactive front-end to scsh, the Scheme Shell, and it closely integrates Scheme evaluation with command execution. The paper also shows how Commander S employs techniques from object-oriented programming, concurrent programming, and functional programming techniques.

1. Introduction

Common Unix shells such as `tcsh` or `bash` make no effort to understand the output of the commands and built-in commands they execute on the behalf of the user. Instead they simply direct the output to the terminal and force the user to interpret the text own her own. As subsequent commands often build on the output of previous commands, the user needs to enter text that has been output by previous commands. As an example, consider a user that wants to terminate her browser because it hangs once again. She only knows the name of the executable (`netscape`) but not the process ID. Hence she first executes the `ps` command:

```
# ps
  PID    TIME COMMAND
  704  0:00.30 tcsh
 1729  6:01.35 xemacs (xemacs-21.4.17)
 1740  8:10.03 netscape
 5823  0:00.07 tcsh
```

From the output, she learns that the process ID of the browser is 1740. Now she can issue the `kill` command:

```
# kill 1740
```

Even though the previous `ps` command already emitted the process ID 1740, the user has to enter the number manually and double-check to get the right one. Killing processes by name is so common that there is a wide-spread Perl program called `killall` that terminates all running processes with a given name. However, `killall` is not appropriate if multiple processes with the same name exist but only one of them is to be terminated.

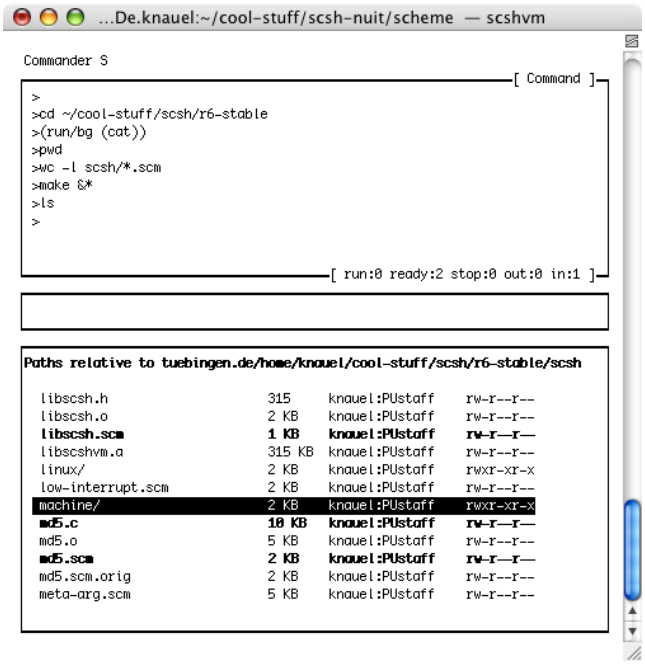


Figure 1. Commander S

Commander S takes a different approach to the concept of an interactive Unix shell: Commander S tries to understand the output of the commands it executes and present it to the user in such a way that the user can easily refer to the output of previous commands. To that end, Commander S draws a user interface on the terminal using the ncurses library. It divides the screen into three areas as shown in Figure 1: The upper half of the screen occupies the *command window* where the user enters the command line. The command line provides the usual line editing facilities such as cursor movement. Below is a small window, called the *current command window*, which shows the last command being executed. The *result window* covers the rest of the screen and contains the output of the last command. The crucial point of the result window is that Commander S presents—for an extensible set of known commands—the result of the commands not simply as text but as structured data. The user can change the focus from the command buffer to the result buffer and *explore* the result. This means that through various key-bindings, the user can invoke other commands that apply to the data presented in the result window. Furthermore, the user can *paste* the data from the result window into the command window to complete the next command line.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

In the case of the example above, Commander S knows that the result of the `ps` command is a list of processes. It presents this list in the result window as follows:

```
  PID  TIME COMMAND
  704   0:00.30 tcsh
 1729  6:01.35 xemacs (xemacs-21.4.17)
 1740  8:10.03 netscape
 5823  0:00.07 tcsh
```

The result window shows the first line with inverted colors because it is the *focus object*. Some key-bindings modify the focus object only, while others affect the entire result window. Of course, the user can also change the focus object with key strikes. For the list of processes, she needs to press the up and down arrows. To return to the task of killing the browser, user needs to press the down key twice and can then press the key for sending the focus object to the command window. Now, she only needs to add the `kill` command to the command line and press the return key to invoke it. If the user were to kill several processes, she would have to mark them for selection by making one after the other the focus object and pressing the marking key. Then the key for pasting the selection will send them to the command window. Sometimes it is desirable to build the command line not only from the results for the most recent command but from one or more commands that were executed earlier. To support this, Commander S maintains a history for the result buffer in which the user can go backwards and forwards as necessary. This history makes the old results immediately available and the user does not need to use the scrolling facility of the terminal if a command with a larger amount of output happened to be before the result the user is searching for. The current command window always informs the user, which command line produced the output in the result window.

Commander S is also an interactive front-end to scsh, the Scheme Shell. This is realized by a second mode, called *Scheme mode*, for the command window, to which the user can switch from the standard *command mode* with a single key press. The interaction between result window and command window also works for the Scheme mode, but the representation of the pasted objects are s-expressions in this case. The combination of both modes enables the user to combine the power of Scheme with the brevity of shell commands.

In addition, Commander S extends the job control features of common Unix shells. First, the job control facility displays the list of current jobs in the result buffer with key-bindings for the common commands such as putting a job into foreground or background. Second, Commander S uses the ncurses library to continuously display the status of the all current jobs. Finally, Commander S can execute a background job with a separate terminal and allows the user to switch to the terminal, view the running output, or enter new input. To that end, Commander S provides a terminal emulation which stores the output of the process.

1.1 Overview

Section 2 explains some programming techniques and particular libraries used for implementing Commander S. Section 3 gives an overview on Commander S's kernel and describes the implementation of some central features of the user interface. Section 4 describes the interface for writing new viewers. Section 5 pictures some standard viewers such as the process viewer and the directory viewer. Section 6 provides details on the job control implemented by Commander S. Section 7 lists some related work, and Section 8 concludes and presents future work.

2. Preliminaries

This section explains some programming techniques and libraries used to implement Commander S. A reader familiar with the particular techniques may choose to skip the corresponding sections.

2.1 Object-oriented Programming in Scheme

The *viewers* described in Section 5 and Section 4 undertake the task of displaying the result of a command according to its structure. Viewers are implemented in terms of object-oriented programming (Section 4 motivates this design decision). We used the object system proposed by Adams and Rees [2] as a foundation. This system is elegant, easy to implement and very powerful. The complete machinery needed for the object system is given by functions shown in Figure 2. The system represents an object as a procedure that binds the instance variables in its closure and accepts a message (a symbol) as its sole argument. It dispatches on the message and returns the corresponding method as a procedure (see `get-method`). All methods accept the object as their first argument to ensure that overridden methods always get the correct object. Hence, `send`, the construct for calling a method, calls `get-method` first to acquire the actual method and calls that method with the object itself plus the arguments passed to `send`.

2.2 Concurrent Programming using the Concurrent ML API

Commander S is implemented as a concurrent application spawning various threads. To synchronize the threads, Commander S employs a Scheme implementation of the Concurrent ML (CML, for short) concurrency functionality [7]. The implementation is given as a library that is part of *Sunterlib*, the Scheme Untergrund Library [1]. This section provides a short introduction to the subset of the CML API used throughout the implementation of Commander S.

CML offers a collection of data-structures for the communication between threads. For the implementation of Commander S, *synchronous channels* and *placeholders* are important. A channel offers a `send` operation that posts a value to channel and a `receive` operation that reads a value posted to the channel. The communication is synchronous, thus, a `send` operation returns exactly at the time when another thread tries to `receive` a value from the channel (and vice versa). A placeholder is an updateable cell, allowing exactly one assignment. A thread reading the value of a placeholder with `placeholder-value` blocks until another thread updates placeholder with a value using `placeholder-set!`. Updating a placeholder already containing a value yields an error.

The CML frameworks allows the decoupling of describing a synchronous operation from actually performing the operation. Thus, synchronous operations become first-class values, called *rendezvous* in the CML notation. The `receive` operation on a synchronous channel, for example, is composed of generating a rendezvous that describes synchronous operation (e. g. "receive a message on a channel") and waiting till the rendezvous actually occurs. Thus, `receive` is implemented as follows:

```
(define (get-method object message)
  (object message))

(define method? procedure?)

(define (send object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message))))
```

Figure 2. Machinery for the object system.

```
(define (receive channel)
  (sync (receive-rv channel)))
```

In which `receive-rv` is a constructor for rendezvous that describe a receive operation on a synchronous channel and `sync` is the function that blocks the thread until a rendezvous actually takes place, or phrased in CML terminology, becomes *enabled*. `Send` and `placeholder-value` may be decomposed in the very same way using `send-rv` and `placeholder-value-rv`.

CML provides combinators that combine multiple rendezvous to a more complex rendezvous. The most important combinator is `choose`, which waits for the first rendezvous from a given list of rendezvous to become enabled. Commander S frequently uses `select`, the synchronous variant of `choose`. The `wrap` combinator allows associating a *post-synchronization action* in form of a function with a rendezvous. When the rendezvous becomes enabled, the associated action is carried out. Function that serve as the post-synchronization actions accept one value — the value that becomes available upon synchronization of the corresponding event. For a receive operation, for example, the value given to the action function is the value received via the channel. The following example code illustrates `select` and `wrap`:

```
(select
 (wrap (receive-rv channel-1)
       (lambda (value)
         (placeholder-set! p value)))
 (wrap (placeholder-value-rv q)
       (lambda (value)
         (send channel-2 value))))
```

Here, `select` combines two rendezvous associated with post-synchronization actions and blocks until the first rendezvous becomes enabled. The first rendezvous in question describes a receive-operation on a synchronous channel named `channel-1`. `Wrap` associates a function with this rendezvous that places the value received via `channel-1` in a placeholder `p`. The second rendezvous describes the synchronous operation of waiting for the value of placeholder `q` becoming available. This rendezvous is also associated with a post-synchronization function which takes the value that just became on-hand and sends it to `channel-2`.

2.3 The ncurses library

Ncurses [3] is a C library that provides a high-level interface to terminal control. In practice a multiplicity of terminal emulations, each having their own control sequences, is in use. Thus, even small tasks like placing the cursor at a certain position on the screen become complex. To assure that an application is portable, the applications needs to know the escape codes of many terminal emulations. Ncurses relieves the programmer of this task. Given a standardized abstract description of a terminal emulation, a so-called *terminfo* entry, usually provided by the maker of the operating system, ncurses learns a particular terminal emulation. The high-level interface of ncurses provides functions for creating overlapping windows, outputting text, controlling the color of output, and placing the cursor. Ncurses also offers a function `wgetch` for reading input from the terminal that decodes the control sequences the terminal emulation uses to encode special keys (such as cursor movement), to a standard representation. We set aside a survey of the ncurses functions used and instead explain their functionality where occur in the following sections.

A separate library for scsh, called `scsh-ncurses`, provides Scheme bindings for all ncurses functions using scsh's foreign function interface. Writing the stubs needed to encode and decode C and Scheme values and calling the ncurses functions is almost straightforward. Just `wgetch` requires special attention. The `wgetch` function reads a character from the terminal, decodes the control sequence if necessary, and returns an integer key code. If no input is available, the behavior of `wgetch` depends on a global mode: in *delay mode*, the function blocks the process until the input becomes available, whereas in *non-delay mode* the functions yields an error. From the perspective of a scsh user, either mode is unfavorable. Calling `wgetch` in delay-mode blocks the whole scsh-process and subsequently all Scheme threads.[1] In non-delay mode, a Scheme thread waiting for input would have to wait busily, thus waste processor time. A preferable mode of operation is to block solely the Scheme thread calling `wgetch`. To achieve this behavior, `scsh-ncurses` calls `wgetch` in non-delay mode at first. If `wgetch` yields an error, `scsh-ncurses` calls scsh's `select` on the terminal to block the Scheme thread calling `select` until the terminal becomes available for reading. Scsh uses the Unix `select` call internally to wait for the file and socket descriptors associated with Scheme ports to become ready for reading and writing. Scsh also offers `select` as Scheme function, which adds the Scheme ports supplied as arguments to the list of file descriptors to watch with the internal `select`.

3. Commander S's kernel

The introduction left unspecified how Commander S recognizes the meaning of a command's output. The idea is not to execute the program directly, but hand over this task to a function that runs the program and parses its output. In the notion of Commander S this function is a *command plug-in*. A command plug-in registers itself as a wrapper for the execution of a certain program. Displaying the parsed output in the result buffer is not in the field of duty of the command plug-in. Instead, *viewer plug-ins* present the output in a structured way. A viewer plug-in registers itself as the presenter for results of a certain type. Command plug-ins are expected to produce a result value of a distinguishable type. Thus, Commander S decouples command evaluation from presentation of the output.

The kernel of Commander S may be regarded as a read-eval-print-loop. Basically, a central event loop processes the input, invokes a command plug-in or executes an external program, and chooses the viewer plug-in to present the result in the result buffer. In Scheme mode, usual Scheme evaluation takes place, but the result is displayed using viewer plug-ins as well. Thus, the evaluation of Scheme expressions also benefits of the power of viewer plug-ins. This section describes the crucial parts of Commander S's kernel.

3.1 Event loop

A central event loop receives all input of the terminal and decides what to do. Basically, the decision depends on two factors: which window has the focus and whether the key pressed has special meaning.

Keys with special meaning, such as the `return` key, are treated by the kernel. The `return` key triggers the evaluation of a command. `Cursor-up` and `page-up` or `cursor-down` and `page-down` keys move through the command history and result history, respectively (see Section 3.6). The key sequence `Control-x` is treated as a prefix, and thus modifies the meaning of the next key press. The sequence `Control-x o` switches the buffer currently focused. `Control-x p` and `Control-x P` paste the current selection and the current focus value, respectively, into the command buffer (see Section 3.3).

If the command window has the focus and the key has no special meaning to the kernel, the key event is passed to the function implementing line-editing (see Section 3.5), which interprets the key accordingly and updates the command buffer. Before continuing in the event loop, the command window needs to be updated to reflect the new state of the command buffer. Thus, the event loop calls a function to repaint the affected part of the command window.

[1] Scsh employs a user-level thread system

⟨command-line⟩ ::= ⟨cmd⟩ (⟨comb⟩ ⟨cmd⟩)* ⟨job⟩?
⟨cmd⟩ ::= ⟨prog⟩ ⟨arg⟩* ⟨redir⟩*
⟨redir⟩ ::= (> | < | >>) ⟨fname⟩
| << ⟨s-expr⟩
⟨comb⟩ ::= | | && | || | ;
⟨job⟩ ::= & | &*
⟨prog⟩ ::= ⟨str⟩ | ⟨unquote⟩
⟨fname⟩ ::= ⟨str⟩ | ⟨unquote⟩
⟨unquote⟩ ::= ,⟨s-expr⟩ | ,@⟨s-expr⟩
⟨str⟩ ::= ⟨scheme-string⟩
| c^+ $c \notin$ {&, |, <, >, ,}

Figure 3. Command language

If the result window has the focus, the key event is passed to the viewer currently visible in the result window. Thus, except for the key sequences listed above, a viewer gets all key events.

3.2 Executing commands

How Commander S executes a command depends on whether the command has been entered in Scheme mode or command mode. If the command buffer is in Scheme mode, the kernel expects the line entered to be a Scheme expression and evaluates it using `eval`. The command mode, in contrast, works akin the prompt of a traditional shell.

The commands entered in the command mode must conform to the *command language* of Commander S. Figure 3 shows a grammar for the command language. Except for some minor differences, this language largely accords to the syntax for commands that users are accustomed to by traditional shells. A notable difference concerns strings, which Commander S models like scsh. While a shell like `tcsh` distinguishes strings in single quotes, double quotes, and backward quotes (for using the output of a command as a string), strings in Commander S's command language are always Scheme strings. The command language is implicitly quasiquoted. Thus, in contexts where a string is expected, the user may use unquote and specify a Scheme expression to be evaluated. Results of the evaluated expression may be a string, a symbol, or an integer. This way, the `tcsh` command `kill ‘cat /var/run/httpd.pid‘` that employs backward quotes to use the contents of the file `/var/run/httpd.pid` as an argument for `kill` may be written as `kill ,(run (cat /var/run/httpd.pid))` in Commander S's command language.

Scsh already supplies a mechanism for running an external program: the `run` macro. This macro expects a specification for the program to run and the redirections of the input and output channels as its arguments. The specification has special syntactic notions called *process forms* and *extended process forms* [9]. Commander S includes a little compiler, which translates a command language command to a process form suitable for the usage with `run`. Thus, when a user submits a command, the compiler generates a corresponding process form and Commander S calls `eval` to actually run the program as specified.

However, the compiled process form demands some preparations before it may be evaluated by `eval`: the `run` macro doesn't substitute shortcuts symbols widely-used by traditional shells. These shortcuts include the tilde, which denotes the user's home directory, environment variable names, and *glob-patterns*. A glob-pattern specifies a list of files by a regular expression. The glob pattern {`/var/tmp,/tmp`}`/*.scm`, for example, specifies a list of all files with names ending in `.scm` in the directories `/var/tmp` and `/tmp`. Thus, Commander S inserts an expansion pass before evaluating a command that searches the compiled command for shortcuts symbols and replaces them. To implement globbing, Commander S uses the C shell compatible implementation of `glob` that is part of the scsh API.

The evaluation of Scheme expressions takes place in a separate environment called the *shell environment*. The basis for this environment is the module definition of the *shell module* which imports `scheme-with-scsh`, a module providing R^5RS and the whole scsh API. The Scheme 48 module system facilitates turning a module into an environment suitable as an argument for Scheme's `eval` function. Thus, evaluating Scheme expressions boils down to calling `eval` and using the shell environment as the environment for evaluation.

The shell module redefines a choice of scsh functions to return a value with a distinguishable type. `Directory-files` serves as an example; if called without arguments, this function returns the contents of the current working directory as a list of strings. This representation is very handy when writing scripts. However, this representation of directory contents is indistinguishable from an arbitrary list of strings. This poses a problem: the viewer to be used to display a result is selected by examining the result. Thus, the shell module introduces a new record type `fs-object`, which encapsulates a file-system object, and redefines `directory-files` to return a list of `fs-object`s. The redefinition of `directory-files` calls the original definition of `directory-files`, imported with a different name, and wraps the resulting filenames in `fs-object` records. So far the shell module only redefines a few functions that return filenames. An aim of future work is to apply this technique to other parts of the scsh API as well.

3.3 Focus value table

Pasting values into the command window running in Scheme mode requires an external representation of the value. This severely restricts the set of values usable for pasting. For example in scsh records, continuations, and procedures have no external representation. Thus, Commander S allows pasting objects as a reference into a global table called the *focus value table*. View plug-ins may register a value in the table using `add-focus-object` which returns an integer index. The function `focus-value-ref` returns the stored value at a given index. Hence, the viewer plug-in may avoid converting a value to an external representation and return a call to `focus-value-ref` instead.

3.4 Command plug-ins

Command plug-ins undertake the task of running a particular external program, parsing the program's output and representing the result as an distinguishable type. The command plug-in for `ps` exemplifies this. If a user enters `ps` to see the list of running processes, in the command mode of Commander S, this invokes the `ps` plug-in. The `ps` plug-in runs the actual `ps` program provided by the operating system and parses its output. The result is represented as a list of `process` records, thereby making the result distinguishable from an arbitrary list of strings and enabling viewers to recognize the type of the result.

The function `register-plugin!` registers a new plug-in with Commander S. The constructor `make-command-plugin` creates a new command plug-in record which contains three entries: A name for invoking the plug-in, a completion function that calculates completions for the arguments (see Section 3.7), and the *plug-in function*. The kernel calls the plug-in function to run the command, parse the output, and produce the result value. Instead of executing an external program, a plug-in function may also call a scsh function. The following code shows the command plug-in for `printenv` as example. `Printenv` returns a list of all environment variables:

```
(register-plugin!
 (make-command-plugin "printenv"
```

```
no-completer
(lambda (command args)
  (env->alist))))
```

The `no-completer` is a completion functions that offers no completions for a command (see Section 3.7). The scsh function `env->alist` returns all environment variables as an association list.

3.5 Line-editing

The feature users miss most when using scsh in an interactive session is line-editing. Line-editing involves making the backspace key work as expected, allowing the user to move the cursor using the cursor keys, inserting text at an arbitrary position of the command line, and some extra features the user is accustomed to from text editors. The scsh REPL does not provide line-editing because it applies `read` directly to standard input to read from the terminal. However, the command buffer of Commander S offers a line-editing functionality with the features mentioned above and feeds the input into `read` (or the parser for the command language) only after the user has pressed the `return` key. The line-editing functionality is implemented in terms of the ncurses (see Section 2.3), thus is portable and involves no emulation specific code.

3.6 Command and result history

Like conventional shells, Commander S offers a so-called *command history*. A command history provides a way to access the prior commands entered during the session. Most Unix shells bind the cursor keys to a function that cycles through the list of commands and displays prior commands at the prompt. This feature is especially useful when the user executes a series of similar commands.

Besides the command history Commander S also provides a *result history*. The motivation for this novel feature is a limitation of traditional Unix shells that don't provide a method to access the output or result of a prior command execution. In this case the user falls back on a feature of her terminal emulation program. These programs usually buffer the output of the terminal session, thus, the user may scroll up and view the output of commands issued afore. To reuse a prior result the user copies the text to the command prompt using a copy and paste mechanism provided by the terminal program. This method, although exercised by numerous users, has at least two drawbacks. First, it may be hard to find the wanted result — there may be lots of output to search through and the wanted output may even be mingled with another processes output (see Section 6). Second, there is only access to a textual representation of the result.

Commander S saves the result objects created during a session in the result history. Thus, the user may go back in the result history at any time and continue to use a saved result object. The result history facilitates the task of finding the desired result — each command is associated clearly with the result it produced. While cycling through the result history, the active command window shows the command used to produce the result shown in the result buffer.

In the notion of Commander S, a result history is easy to implement. Having the viewer objects (see Section 4) instanced, the kernel stores the object along with the corresponding command in a list that serves as the history. Thus, going back and forth in the history selects an existing viewer object that is set as the the current result object. Subsequently, the kernel clears the result window and sends a `paint` message to the new current result object to make the object visible.

3.7 Programmable completion

Most shells offer an automatic completion for commands and arguments entered partially at the prompt. Usually pressing the tabulator key while editing a command line at the prompt triggers a *completion function*. This function considers the token of the command line the user is currently editing (that is, the token where the cursor is) and finds a set of strings to which the partially entered token is a prefix. This set depicts the set of possible completion for the token. If there is more than one possible completion, most shells simply display the possible completions and expect the user to continue editing the token until the prefix becomes unambiguous. Depending on the position of the token in the command line, the token denotes a program to be executed or an argument to a program. Thus, only executable files come into question as completions for the command token, whereas, intuitively there no such constraint for argument tokens. Most shells accommodate this observation by using different completion functions for the particular tokens of a command line.

Popular shells like `tcsh`, `bash`, and `zsh` offer a *programmable completion function* which allow users to write completion functions tailored to syntax of arguments of a specific command. The file transfer program `ftp`, for example, expects a host name to connect to as its first argument. The following `tcsh` commands establishes an appropriate completion function for `ftp`:

```
> set preferred_ftp_hosts=(ftp.gnu.org ftp.x.org)
> complete ftp 'p/1/\$preferred_ftp_hosts/'
```

This example specifies a completion for the first argument only.[2] The possible completions for this argument are given as a list specified in the variable `preferred_ftp_hosts`.

Commander S provides a similar programmable completion function for the command mode. If the user presses the tabulator key a general completion function calls the parser for the command language and identifies the token the cursor is pointing at. This token is considered for completion. Depending on the position of this token a more specific completion function is selected. The completion function for command tokens is a built into Commander S and uses the union of executables available in the paths listed in `PATH` and the set of registered command plug-in names as possible completions. However, the user may wish to specify an executable by entering a complete path. In this case the command completion function calls `complete-with-filesystem-objects` to build the list of completions. This function checks whether there is a file or directory that matches the partially entered path. If the token matches a directory name, `complete-with-filesystem-objects` offers the contents of this directory as possible completions. Otherwise the parent directory of the partial names is searched for completions.

If a completion function returns a single possible completion, Commander S may replace the token on the command line with this completion and repaint the command prompt. However, if there is more than one completion, Commander S uses the result buffer to display the list of completions. The user may use the list as an aide to memory and continue to type the token, or by pressing the tabulator key a second time switch the buffer focus and select a completion using the cursor keys directly.

The general completion functions also handles the completion of arguments. Unless a specific a completion function for the current command token is specified, it calls `complete-with-filesystem-objects` to complete the argument. Specific completion functions are tied to command plug-ins. Thus, to provide a special completion function, the user adds a command plug-in. The following command plug-in for the `ftp` command provides such a completion function.

[2] `p/1` stands for "position one"

```
(register-plugin!
 (make-command-plugin
  "ftp"
  (let* ((hosts '("ftp.gnu.org" "ftp.x.org"))
         (cs (make-completion-set hosts)))
    (lambda (command to-complete)
      (completions-for
       cs (or (to-complete-prefix to-complete) ""))))
  just-run-in-foreground))
```

In this example, the second argument to `make-command-plugin` is the completion function. A completion function has two arguments; the abstract syntax of the command line and the token to completed. The completion function in question uses a built-in list of host names as possible completions. `Make-completion-set` creates a special caching data-structure which speeds up the computation of matching completions. This is especially useful when the set of possible completion is big, for example, when searching the completions for file names. The procedure `completions-for` searches and returns the matching completions for the prefix returned by `to-complete-prefix` in the completion set.

4. Implementing Viewer Plug-ins

In the notion of Commander S a *viewer plug-in* (*viewer* for short) undertakes the task of displaying the result value of a command in a structured fashion. However, a viewer may go beyond just displaying data and implement a small application running in the result window. The predefined file system viewer (see Section 5), for example, not only displays files and directories but also allows navigating through subdirectories.

Given a result value, Commander S tries to find the appropriate viewer. Each viewer comes with a predicate that identifies the result values the viewer handles. Commander S applies the predicates provided by the registered viewers to the result value. The viewer belonging to the first predicate to evaluate to true accepts the bid. Now, Commander S instances a new viewer using the accordant constructor and asks the viewer to paint itself to the result window.

Viewers are implemented using object-oriented programming (see Section 2.1 for an introduction of the object system used). A viewer depicts an object that accepts the messages sent by kernel and encapsulates a state. In this setting, an object-oriented approach appeared to be a natural choice. Commander S sends the following messages to viewer objects:

- `paint` The `paint` message asks the object to paint itself to the result window. This message is sent to objects just created or if an result object becomes the current result object. (i.e., if the user cycles through the result history, see Section 3.6). As arguments, the objects receives the ncurses window to paint in, a result buffer object which contains information about the result window's size, and a boolean indicating whether the result window has the focus.
- `key-press` If the result window has the focus, the current result object receives a `key-press` message whenever the user presses a key. The object receives the key code and a boolean saying whether the special prefix key sequence `Control+x` is active as arguments. The kernel expects this method to return an instance of the viewer and stores this instance in the history. This is a clincher, since this allows a viewer to instantiate and return a different viewer. The viewer responsible for displaying the contents of a user record, for example, uses this case to instance a directory viewer object if the user presses return key on the line displaying the path to a user's home directory.
- `get-selection-as-ref` This message asks the viewer to return the current selection as a reference into `focus-value-table` (see Section 3.3) received as an argument. The message is only available if the command buffer is in Scheme mode, thus, the return value of this method has to be a piece of Scheme code (as a string).
- `get-selection-as-text` This message asks the viewer to return the current selection in a textual representation. If selections don't make sense in context of a result value, this method may return false. A boolean delivered as an argument says whether the selection is to be inserted into the command or Scheme mode. Thus, a viewer may deliver an adequate string (see Section 4.2 for an example). It is conceivable though that representing the selection as a string makes no sense. In this case a viewer may choose to understand the `get-selection-as-text` message as a `get-selection-as-ref` message, hence, requiring a reference to the `focus-value-table`. To facilitate this, the `focus-value-table` is passed as a second argument to the `get-selection-as-text` messages.

4.1 Selection lists

Before giving an example for the implementation of a viewer object, we shall describe *selection lists*. Selection lists are an important user interface widget, akin to menus, used by almost all viewer objects. A selection list displays a given set of entries as sequential lines at an arbitrary position inside an ncurses window. Using the cursor keys, the user may move a selection bar over the lines to focus a particular entry, and mark and unmark entries. Most viewers employ a selection list using marking to facilitate selecting items which are to be processed together. The selection list also determines the area in view if the number of items to display exceeds the space assigned to the selection list.

The constructor `make-selection-list` expects as its argument a list of records of type `element` that denote the items of the selection list, and returns a Scheme record representing the selection list. An `element` record consists of a field that carries the object to be returned if the user marks the accordant line, a boolean saying whether this entry may be marked at all, and the text to be displayed.

The `paint-selection-list-at` operation accepts a selection list, window-based coordinates, and an ncurses window as its arguments and paints the selection list in its current state at the given coordinates to the window. To pass key events to a selection list, viewer objects call the function `selection-list-handle-key-press` which updates and returns the state of selection list accordingly.

Implementing a `get-selection-as-text` method in a viewer frequently boils down to getting the list of marked entries from a selection list using `selection-list-get-selection`, or, if this list is empty because no entries are marked, getting the entry currently focused by the selection bar using `selection-list-selected-entry`. The selection list implementation offers the function `make-get-selection-as-ref-method` which returns a function suitable as an implementation of a `get-selection-as-ref` method. The focus objects returned by methods implemented using this function stand for the return object specified in the accordant `element` record.

4.2 Example: process viewer

As an example for the implementation of viewers, this section describes the implementation of the process viewer from the introduction and sketches the implementation of the command plug-in for `ps`.

The process viewer views the output of the `ps` command. The `ps` command is a command plug-in based on the `portable ps` library from Sunterlib [1]. As the `ps` command is not standardized, the library dispatches on the type of the host operating system and then issues the `ps` command with options chosen to get all processes

and a set of additional information available on all supported platforms. It then parses the output and stuffs it info a record of type `process`. The `ps` command plug-in does not currently support additional options but returns this list unchanged. In the future, the `ps` command should accept arguments to restrict the returned processes and customize the additional information. While argument parsing is certainly more work, a user who often switches operating systems would certainly be happy to use the same set of options on all platforms. Of course, the syntax of the options could easily be made customizable.

Figure 4 contains the implementation of the viewer plug-in for processes. The function `make-process-viewer` is the constructor for process viewer objects. The constructor is called by the kernel, if the predicate for this viewer, `list-of-processes?`, identified a result value as a list of process objects. The kernel supplies the result value in question and the buffer to draw to as arguments to the constructor. The constructor returns a function that given a message name returns a function implementing the method. The instance variables of the object are bound in the closure of this function. The process viewer employs a selection list (see Section 4.1) to display a list of processes. `Make-process-selection-list` formats the process objects and uses `make-selection-list` to create a selection list that fits into the result window leaving one line free for a heading. On a `paint` message, the viewer displays the header and calls the procedure `paint-selection-list-at` to draw the selection list beneath the header. A `key-press` message is also forwarded to the selection list. On a `get-selection-as-text` message, it returns the PIDs of the selected processes for the command mode and a list of PIDs in the Scheme mode.

Finally, the last two lines of the figure register the process viewer plug-in registers as viewer for a list of records of type `process` and hands out the constructor to the kernel.

```
(define (make-process-viewer processes buffer)
  (let* ((processes processes)
         (cols (result-buffer-num-cols buffer))
         (lines (result-buffer-num-lines buffer))
         (sel-list
          (make-process-selection-list
           cols (- lines 1) processes))
         (header (make-header-line cols)))

    (define (get-selection-as-text
             self for-scheme-mode?
             focus-object-table)
      (let* ((marked
              (selection-list-get-selection sel-list)))
        (cond
         ((null? marked)
          (number->string
           (process-info-pid
            (selection-list-selected-entry sel-list))))
         (for-scheme-mode?
          (string-append
           "'" (exp->string
                (map process-info-pid marked))))
         (else
          (string-join
           (map process-info-pid marked))))))

    (lambda (message)
      (case message
       ((paint)
        (lambda (self win buffer have-focus?)
          (mvwaddstr win 0 0 header)
          (paint-selection-list-at
           sel-list 0 1 win buffer have-focus?)))
       ((key-press)
        (lambda (self key control-x-pressed?)
          (set! sel-list
                (selection-list-handle-key-press
                 sel-list key))
          self))
       ((get-selection-as-text) get-selection-as-text)
       ((get-selection-as-ref)
        (make-get-selection-as-ref-method sel-list))
       (else
        (error "process-viewer unknown message"))))))

(register-plugin!
 (make-viewer make-process-viewer list-of-processes?))
```

Figure 4. Implementation of the process viewer (excerpt).

5. Predefined viewers

The previous section already presented Commander S's viewer for processes. In this section, we present further viewers for filesystem objects, user and group information and results of commands related to AFS. In addition, a viewer for inspecting arbitrary Scheme values is described.

5.1 The filesystem viewer

Dealing with files is another common scenario where the user is forced to re-enter text that appeared in the output of a previous command. A common pattern is that the user first issues an `ls` command to list the files within a directory and then uses another command to manipulate certain files. To view the most recent error log file of an Apache web-server, the user could first use `ls -lat`, which prints the files sorted by date:

```
# ls -lt
-rw-r--r--   5543 Jun 15 02:00 error_log.1118275200
drwx--x---    512 Jun 15 02:00 ./
-rw-r--r--  49024 Jun 14 15:04 access_log.1118275200
-rw-r--r--  66312 Jun  8 21:59 access_log.1117670400
-rw-r--r--  11498 Jun  8 21:59 error_log.1117670400
-rw-r--r-- 140048 Jun  1 18:17 access_log.1117065600
-rw-r--r--   4688 Jun  1 05:36 error_log.1117065600
drwx--x---    512 Mar 25  2004 ../
```

Next, she would invoke a viewer such as `less` on the latest file `error_log.1118275200`:

```
# less error_log.1118275200
```

Again, the user has to enter text that appeared in the output of a previous command. Modern shells such as `bash` or `tcsh` will help the user to enter by providing *command line completion*. This means that the shell examines the command line already typed and completes the last token as far as possible or presents the user a set of possible completions. The shell derives the possible completions from the leading command, the default mode is to complete the token as a filename. In the example above, the user could ask the shell to complete the command line `less e`. The shell will expand this to `less error_log.111` and list all error files as possible completions. Now the user needs to inspect the output of the previous `ls -lt` command to learn that the name of the most recent file continues with an `8`. After entering this character, the shell is able to fully complete the filename. However, while command line completion is certainly of great aid for the programmer, the shell again makes no use of the output of previous commands, which contains in our example the files in chronological order. If the example takes place within the `tcsh` shell, this is especially disappointing as there `ls` is a built-in command. This means, the output is not produced by some external command but by the shell itself.

The user could try to save typing by combining entering a command line that extracts the name of the newest error log for the output of `ls` and calls `less` on it:

```
# less `ls -1t err* | head -n 1`
```

While this approach is close in the spirit of the Unix philosophy to combine little tools to perform the work, the command line is rather long and fragile. We would not dare to use such a construction on the command-line for a command such as `rm`. It also requires the user to know in advance that error logs (and only these) start with `err`.

Commander S knows that the result of the `ls -lat` command is a list of files. It presents this list in the result window as follows:

```
Paths relative to /usr/local/svn/logs

-rw-r--r--   5543 Jun 15 02:00 error_log.1118275200
drwx--x---    512 Jun 15 02:00 ./
-rw-r--r--  49024 Jun 14 15:04 access_log.1118275200
-rw-r--r--  66312 Jun  8 21:59 access_log.1117670400
-rw-r--r--  11498 Jun  8 21:59 error_log.1117670400
-rw-r--r-- 140048 Jun  1 18:17 access_log.1117065600
-rw-r--r--   4688 Jun  1 05:36 error_log.1117065600
drwx--x---    512 Mar 25  2004 ../
```

That is, the presentation of a list of files is the list of the file names relative to a directory, which is displayed in the first line. If the focus object is a directory and the user presses the return key, the result window will display the contents of this directory. To return to the task of viewing the latest log file, the user can immediately press the key for sending the focus object to the command window, as the focus object is already the most recent file. Now, she only needs to add the `less` command to the command line and press the return key to invoke it. Pasting files to the command window inserts them as absolute filenames. If the command window is in Scheme mode, pasting inserts filenames as strings.

If the user enters the `ls` command, Commander S does not really invoke the `ls` program and parse its output. Instead, it uses the scsh function *file-info* to obtain the file status information and the function `directory-files` to get the contents of a directory. From this information, it generates a list of records of type `fs-object`. An `fs-object` combines a filename with file status information. The filesystem viewer registers itself as the viewer for `fs-objects` and for lists of `fs-objects`.

As Commander S provides its own binding for the scsh procedure `directory-files`, which returns a list of `fs-objects` instead of a list of strings, and extends the scsh functions which operate on filenames to `fs-objects`, the viewer is also able to present the values of Scheme expressions returning lists of filenames.

The functionality of filesystem viewer could be extended in various aspects: additional key-bindings for renaming, deleting, or copying files, manipulation of file mode bits, invoking of a default application based on the filename suffix, and so forth. However, while we would certainly like to have these features, it is not the focus of our current work as programs like midnight commander or the dired plug-in for Emacs already show the merits of this approach. Instead, Commander S aims combine graphical presentation with command execution and shell programming. Unlike pure front-ends for filesystem browsers, Commander S is also not limited to the presentation of filesystem objects.

5.2 User and group information viewer

User and group information are ubiquitous in Unix. For user information, scsh provides the procedure `user-info` as wrapper for the standard C functions `getpwnam`/`getpwuid` to return the user information from a given login name or UID. It returns a record `user-info` with the fields `name`, `uid`, `gid`, `home-dir`, and `shell` which contain the corresponding entries form the user database (usually `/etc/passwd`). For the group information, scsh analogously provides a wrapper `group-info` for the C functions `getgrnam`/`getgrgid`. The fields of the returned record `group-info` are `name`, `gid`, and `members`, the latter containing the users of the group as a list of strings. Commander S contains viewers for the `user-info` and `group-info` records that present the contents of the records in a selection list. The main feature of these viewers is that the user may navigate through the presented information by selecting an entry and pressing the return key: For the `gid` field, Commander S presents the corresponding group information, for the `home-dir`, it invokes the filesystem viewer from Section 5.1 on the home directory, likewise for the `shell` field, and for the members of a group, Commander S presents the associated user information. Here is an example for the value of the expression `(user-info "gasbichl")`:

```
[0: name] gasbichl
 [uid] 666
 [gid] 4711
 [home-dir] /afs/wsi/home/gasbichl
 [shell] /bin/tcsh
```

If the user presses the return key, Commander S presents the information for GID 4711 as follows:

```
 [name] PUstaff
 [gid] 4711
 members:
 gasbichl
 klaeren
 knauel
```

The viewers are implemented in about 130 lines of code but already provide a nice tool for browsing user and group information. We think that in this style a lot of information in the realm of Unix can be presented and thus enable the user to browse this information very conveniently and fast.

5.3 AFS

This section presents two viewers related to the Andrew File System (AFS for short) as an example for using Commander S for viewing the result of special purpose programs. AFS is a network filesystem based on a client-server model. AFS stores the data on the server in logical partitions called *volumes*. Each volume is mounted at some directory below the global `/afs` root. On the client, a local daemon transparently fetches and stores the contents of the volumes from the server and maps it into the local filesystem. AFS also introduces permissions for directories based on access control list (acl for short) and has its own user management. The user views the permissions with the `fs listacl` command and manipulates them with the `fs setacl` command. For example:

```
# fs listacl .
Access list for . is
Normal rights:
  system:administrators rlidwka
  gasbichl rlidwka
  knauel rl
# fs setacl . knauel rli
```

adds the right to insert files into the current directory for the user `knauel`. Commander S saves the user from entering the username that already occurred in the output by displaying the result of `fs listacl` using a selection list:

```
Access list for . is
Normal rights:
system:administrators rlidwka
gasbichl rlidwka
knauel rl
```

By pressing the key for sending the selection, the user can paste the string `knauel rl` to the command window running in command mode behind a `fs setacl`. Alternatively, the user may paste the entry as a pair while in Scheme mode. This is especially useful to set the rights of several users at once. For example, the following expression grants the right to read, list and insert files to a list of such entries which the user would paste from the result window at the place of `...`:

```
(for-each (lambda (acl)
            (fs setacl "." (car acl) "rli")) ...)
```

On the other hand, the viewer for `fs listacl` also supports direct editing of the acl entries. Currently, pressing the deletion key removes an entry from the acl. More features such as direct modification of the rights would be desirable but requires functionality beyond the current capabilities of the selection list.

Commander S also supports management of volumes. The command `fs listquota` takes as argument a directory and prints the quota information for the volume the directory resides in. This is also a convenient way to obtain the name of the volume needed the most volume-related commands. Commander S prints the result of `fs listquota` as

```
Volume Name: home.gasbichl
Quota:       1000000
Used         899724
% Used       90%
Partition    28%
```

From here, the user can either paste the volume name into the command window or press the return key to execute the `vos examine` command on the volume. A future version will also support direct editing of the quota.

The commands for volume manipulation also have command line completion for the volume name argument. Commander S receives the list of all volumes from the command `vos listvldb`. Executing this command may take some time, therefore it is not desirable to initialize this list during startup. Fortunately, command completion is completely programmable in Scheme and during startup the corresponding plug-in can simply spawn a thread which issues `vos listvldb` and initializes the volume list. This way, the user has to wait only if she wants command completion for `vos` before the thread finishes its work.

5.4 Value inspector

The domain of viewers is not limited to the results of Unix commands. In fact, the user may add viewers for any kind of Scheme value. Scheme 48 already comes with an inspection facility to browse arbitrary Scheme values. We have lifted the inspection facility into our ncurses-based framework and use it as the default viewer for exceptions which effectively implements a debugger.

We briefly review the inspection facility in Scheme 48: Its command processor provides a command `,inspect` that takes as its argument a Scheme expression, evaluates it and presents the outermost structure of the resulting value in a menu. There is a menu entry for every immediate sub-value. For a list, the sub-values are the entries of the list, for a record the sub-values are the components of the record, for a continuation the contents of the stack frame makes up the sub-values. A menu entry consists of a number for selection by the user, an optional name for reference, and the external representation of the sub-value. The source of the name depends on the kind of value being inspected: for records it is the name of the record field, for environment frames it is the name of the variables. List or vector entries do not have names. After the presentation of the menu, the user may enter the number of a menu entry to continue inspection with the corresponding sub-value or press the `u` key return form the inspection of a sub-value. For continuations, the `d` key selects the parent continuation. If there are more than 14 sub-values, the `m` key switches the presentation of the menu to the next 14 sub-values and so on. Finally, the `q` key ends the inspector and sets the focus object of the command processor to the last value that has been inspected. The command processor also comes with a `,debug` command which inspects the continuation of the last exception that occurred. As inspection of a continuation displays an excerpt of the source code of the corresponding function call before presenting the menu, this is enough to implement a very useful debugger.

For Commander S we implemented a viewer, called *inspector*, which shows the sub-values of an arbitrary Scheme value in a selection list. The user may select a sub-value by moving the selection bar to it and pressing the return key. In addition, we have adopted the key-bindings for `u` and `d` from Scheme 48.

For the implementation of the inspector, Commander S mainly reverts to the procedure `prepare-menu` from the implementation of the `,inspect` command. The procedure takes as its argument a Scheme value and returns the list of its sub-values as pairs of a name (or `#f`) and the sub-value. Commander S turns these pairs into `element` records for a selection list: The object to be returned on marking is the sub-value itself, all elements are markable, and the text is the external representation shortened to the width of the window. For the latter, we make use of `limited-write`, another utility from Scheme 48 which is a variant of `write` that limits the output to a certain depth and output length. Unfortunately, the single line within a selection list of often not enough space to present complex data structures in a useful manner. Besides the preparation of the selection list, there is not much to do for the inspector: As the `,inspect` command, it prints a source code excerpt for continuation in a header line and being able to return from a sub-value requires the viewer to maintain a stack of visited values. Invoking the inspector on a sub-value pushes the current value on the stack and the `u` key pops a value from the stack and makes it the current value. The `,inspect` command in Scheme 48 proceeds likewise.

We could use the inspector to display any value but we have currently only registered it for the continuations of exceptions, but this may be extended for arbitrary values.

6. Job control

Most Unix shells allow the user to run multiple processes simultaneously. In shell terminology these processes are called *jobs*. A shell usually provides commands to stop and continue jobs, view the list of jobs and their status, and the job's access rights to the terminal. All processes share a single terminal as their standard output and input. The POSIX job control interface [5] enables the shell to control which process may read or write to a terminal. Traditional shells pursue the following policy: A single foreground job has read and write access to the terminal and all background jobs are allowed write to the terminal only. If a background job tries to read from the terminal, the shell suspends the execution of the job until the job becomes the foreground job.

Thus, running multiple background jobs, which write to the terminal yield a mingled output. Basically, the user has two choices to avoid this: redirecting the output of each job to a separate file, or make the shell's job control stopping processes that attempt to write to the terminal. However, both options are disadvantageous. A job control policy with exclusive write access may stop the computation of a background job completely just because there is output available. This not appropriate in all cases, for example when running a daemon from the command line. On the other hand, redirecting the output requires extra effort for setting up the

redirections for standard output and standard error, viewing the file, and deleting the temporary files afterwards.

Commander S adds a third method, not provided by traditional shells, to the picture; so-called *console jobs*. The standard input and output of a console job are connected via a separate pseudo terminal to Commander S. A thread continuously reads the pseudo terminal to ensure that writing to the terminal does not block. A *console* record stores the pseudo terminals and the buffered output of a job. The viewer plug-in for this record type displays the output of the job in the result buffer and updates it continuously as new output arrives. Thus, the user may review the output of a command at any time. Section 6.3 discusses console jobs in detail and presents the implementation at a glance.

Beside console jobs, Commander S offers job control as known from traditional shells. The implementation, however, diverges from traditional implementations. We present a elegant concurrent implementation in the CML framework in the following sections.

Section 6.1 presents the POSIX job control facilities at a glance. A reader familiar with these facilities and their mode of operation may choose to skip this section. Section 6.2 describes how Commander S runs jobs without a separate console. Section 6.3 explains the execution of console jobs. Section 6.4 describes the implementation of the job list, a data structure that maintains the informations on jobs centrally.

6.1 Traditional job control

The POSIX API contains functions for implementing job control which are widely-used by traditional shells. Scsh already provides bindings to these functions. Thus, it was not necessary extend scsh to implement Commander S's job control. This section explains the basics of POSIX job control using scsh's names for the POSIX functions.

Process groups are the basis for job control — a process group is a set of processes, which share a common process group id. Each process is member of exactly one process group. When a process forks, the child process inherits the process group id of the parent — the process is said to *join* the parent's process group. A process may also *open* a new process group by calling `set-process-group`. Each terminal device is associated with one process group, named the *foreground process group*, all other process groups are called *background process groups*. A process group makes itself to the foreground process by calling `set-tty-process-group`. In contrast to processes of background process groups, processes of the foreground process group are granted read and write access to the terminal. If a background process tries to read from the terminal, the kernel terminal driver suspends the job using the `SIGTTIN` signal. Depending on the configuration of the terminal a background job writing to the terminal may also be suspended using the `SIGTTOU` signal. Using `wait`, a parent process may watch if a child gets suspended.

6.2 Jobs without console

Jobs without a separate console are either foreground or background jobs and work akin to jobs in a traditional shell. To execute a foreground job, Commander S temporarily escapes the curses mode and hands the control on the screen over to the foreground job. Once the foreground jobs terminates (or gets suspended by a signal), Commander S reobtains control. Commander S expects a background job neither to read from nor write to the terminal. If the job tries to read or write, however, the job gets suspended and Commander S notifies the user (see Section 6.4). In this case the user may choose to continue the job in foreground. Vice versa, a user may also explicitly stop a foreground job and continue the job in background.

```
(define-syntax run/fg
  (syntax-rules ()
    ((_ epf)
     (run/fg* '(exec-epf epf)))))

(define (run/fg* s-expr)
  (debug-message "run/fg* " s-expr)
  (save-tty-excursion
   (current-input-port)
   (lambda ()
     (def-prog-mode)
     (clear)
     (endwin)
     (restore-initial-tty-info! (current-input-port))
     (drain-tty (current-output-port))
     (obtain-lock paint-lock)
     (let ((foreground-pgrp
            (tty-process-group (current-output-port)))
           (proc
            (fork
             (lambda ()
               (set-process-group (pid) (pid))
               (set-tty-process-group
                 (current-output-port) (pid))
               (eval-shell-env s-expr)))))
       (let* ((job (make-job-sans-console s-expr proc))
              (status (job-status job)))
         (set-tty-process-group
           (current-output-port) foreground-pgrp)
         (newline)
         (display "Press any key to return...")
         (wait-for-key)
         (release-lock paint-lock)
         job)))))
```

Figure 5. Running a job in foreground.

The machinery for running jobs is built on top of scsh's `run` form. The form (`run/fg` *epf*) executes the extended process form *epf* as a foreground job. To specify a program to run and the corresponding redirections of the input and output channels scsh uses a special syntactic notation: process forms and extended process forms. Thus, `run` and `run/fg` are implemented as macros not as functions.

Figure 5 shows the implementation of `run/fg`. Applications of `run/fg` expand into a call to `run/fg*`; a function that expects a piece of Scheme code as a s-expression as its argument. The Scheme code is supposed to actually run the process using scsh's basic `exec-epf` facility. Unlike `run`, `exec-epf` does not fork the process before running the program. `Run/fg*` calls `eval-shell-env` to evaluate the Scheme code in the shell environment. It is important that the evaluation takes place in the shell environment since an extended process form is implicitly backquoted, thus, by using unquote, a user may embed Scheme code in an extended process form. Carrying out the evaluation in the shell environment ensures, for example, that the user may refer to variables defined interactively in the Scheme mode or use focus values.

Before running the process using `eval-shell-env`, `run/fg*` calls a sequence of ncurses functions to save the current screen, clear it and finally escapes the curses mode temporarily using `endwin`. This yields an empty screen called the *result screen*. This avoids that the Commander S screen is garbled with the output of the process. To execute the process, `run/fg*` forks the process, opens a new process group, and makes this process group the new foreground process group. The parent process calls `make-job-sans-console` to create a new job record with the process object returned by `fork`. The parent process uses `job-status`; a wrapper version of `wait` for jobs. Thus, the parent waits until

the child process exits and makes itself the foreground process group again. Afterwards, the parent process waits for a key press to give the user time to read the child's output. It is essential to ensure that no output occurs during the time Commander S is a background process — otherwise the terminal driver would suspend Commander S. To enforce this condition `run/bf` obtains the `paint-lock` which prevents other threads, such as the thread that updates the job status indicator (see Section 6.4), from painting onto the screen.

Running jobs in background works alike using a function `run/bg*`. There, the code for escaping from the curses mode and setting the foreground process drops out. On start-up, Commander S configures the terminal to stop background processes that try to write to terminal, thus, a background cannot garble the screen. Commander S offers two functions for continuing suspended jobs without a console: `Continue/fg` puts a stopped job into the foreground and continues the job, `continue/bg`, vice versa, continues a job as a background job. The implementation of this functions is derived from the implementation of `run/fg*` and `run/bg*`. However, instead of forking and calling `exec-epf`, the functions send the process group of the job a `SIGCONT` signal, thus, the processes continue to execute.

6.3 Console jobs

The implementation of console jobs is more complex than the implementation of jobs without console. While there is no extra effort needed to display the output of job without console — it is only visible on the separate result screen — the output of console jobs causes more effort. The output of a job must be read by Commander S continuously to keep the job running. However, displaying the output in the result buffer as it occurs is not reasonable — the job would behave like an ordinary foreground job.

Here, the concept of viewer plug-ins comes into play. The output of a console job is represented by a *console* record. An accompanying viewer plug-in for this record type displays the output and updates the result buffer as new output arrives. To the kernel a console is conceptually just another value with a predefined viewer plug-in. Each console is accompanied by a thread that reads the pseudo terminal of the process and sends the characters read into a synchronous CML channel. Thus, this thread lifts I/O events into the CML framework.

To actually paint the contents of the output buffer to the screen, the console viewer plug-in uses a so-called *terminal buffer*. The heart of a terminal buffer is a thread spawned by the function shown in Figure 6. The terminal buffer is connected via the synchronous `pty-channel` to the thread that reads the console's output. Depending on whether the console is currently visible in the result buffer or not, the terminal buffer either buffers the new output (by calling `terminal-buffer-add-char`) or buffers it and immediately repaints the result buffer. The decision whether to update the result buffer or not is left up to the console viewer plug-in, which uses `resume-console-output` or `pause-console-output` to stop and continue the updates, respectively.

The terminal buffer performs a second task hidden in the function `terminal-buffer-add-char`. Basically, this function implements a terminal emulator for a small subset of VT100 control codes. The terminal emulation is necessary to restrict the effects of terminal escape codes generated by the running job to the result buffer only. Forwarding the escape codes rawly to terminal Commander S is running on yields undesirable effects. If the running job outputs the escape code to clear the screen, for example, this escape code would be interpreted by the terminal emulator for the terminal Commander S is running on, and clean the entire screen— including the command buffer. Alas ncurses offers no solution to this problem.

```
(define (spawn-console-loop
         pause-channel resume-channel
         window termbuf pty-channel)
  (spawn (lambda ()
    (let lp ((paint? #t))
      (select
       (wrap (receive-rv pause-channel)
             (lambda (ignore)
               (lp #f)))
       (wrap (receive-rv resume-channel)
             (lambda (ignore)
               (lp #t)))
       (wrap (receive-rv pty-channel)
             (lambda (char)
               (cond
                ((eof-object? char) (lp paint?))
                (else
                 (terminal-buffer-add-char
                  termbuf char)
                 (if paint?
                   (begin
                     (curses-paint-terminal-buffer
                      termbuf window)
                     (wrefresh window)))
                 (lp paint?)))))))))))

(define (pause-console-output console)
  (send (console-pause-channel console) 'ignore))

(define (resume-console-output console)
  (send (console-resume-channel console) 'ignore))
```

Figure 6. Updating a `terminal-buffer` and painting it.

6.4 Job status and job list

A job is in one of the following run states: running, finished, stopped, waiting for input, or waiting with output (the latter applies to background jobs without a console only). Traditional shells notify the user either immediately or before drawing the next prompt if the status of a job changes. Both methods have drawbacks: a prompt notification means that the shell prints the notification directly to the terminal at point of time the status change occurs, thus garbling the terminal output. Waiting for the next prompt avoids a garbled screen, but the user has to issue (empty) commands from time to time to see if a status change occurred. A graphical user interface produces relief for this problem.

Commander S's command buffer displays a small gauge, the *job status indicator*, in the lower right corner of the command window (see figure 1). The job status indicator displays the current number of processes in each of the possible state. Whenever the status of a jobs changes, a thread updates the job counts immediately without disrupting the user.

Commander S uses a central *job list* to maintain a list of all jobs. The job list serves two purposes. First of all, it is needed to implement the `jobs` command, which prints a list of all jobs and their current state. As a second task, the job list registers all status changes of a job and informs the job status indicator about the change.

The implementation of the job list was tricky — there are several sources of events that modify the state of the job list: A user may submit a new job at the prompt, stop or continue a job, or a background job may interrupt or finish its execution. Thus, the job list needs to observe several diverse sources for events at once. First of all, user commands such as submitting, continuing, or stopping a job need to inform the job list about the job status changes. The termination or suspension of a background jobs is the second source for events that trigger changes in the state of

the job list. To notice these changes the job list needs to call `wait` for each background job and update the job list. Using the CML framework these diverse sources for events may easily be represented uniformly as rendezvous. Thus, one central `select` synchronously waits for the occurrence of any of the named events.

Figure 7 shows an excerpt from the implementation of the job list. The function `spawn-joblist-surveillant` starts the thread that maintains the job list and returns the `statistics-channel`. This channel connects the job list with the job status indicator — whenever the state of the job list changes in a relevant way, the job list posts the updated job counts to this channel and the thread accompanying the job status indicator updates the gage. The thread spawned by `spawn-joblist-surveillant` executes an infinite loop that uses `select` to choose a rendezvous from the possible sources of events affecting the job list. The job list consists of lists for each run state that are bound locally in the thread. The loop variable `notify?` indicates whether an update of the job status indicator is due. If this is the case, the thread sends the current job counts to `statistics-channel`. The constructor for jobs `make-job-sans-console` and `make-job-with-console` submit the jobs just created to job list using the `add-job-channel`. If a rendezvous on the `add-job-channel` is enabled, the function associated to this event by the `wrap` combinator adds the new job to the list of running jobs and continues the loop. In this case an update of the job status indicator is due, thus the loop function is called with `#t` as the value for `notify?`. Receive rendezvous on the `notify-continue/foreground-channel` indicate that the user issued a `continue/fg` or `continue/bg` command. Thus, a job that is either stopped, waiting with output, or waiting for input changes to the running state. The accordant action for this events deletes the job from the lists for stopped jobs, adds it to the list of running job, and sets `notify?` to true. The `get-job-list-channel` is used by the `jobs` command to get the list of all jobs.

The job list also monitors the status changes of the processes using `wait`. The constructor for jobs spawns a thread that calls `wait` on a job's process object, and fills a CML placeholder with the status value returned by `wait`. The function `job-status-rv` returns the corresponding rendezvous. This way, the status change of a process translates to a CML rendezvous suitable for integration with the job list's `select` call. Thus, the job list surveillance thread includes the `job-status-rv` for all running jobs into the selection of rendezvous by mapping `job-status-rv` on the list of all running jobs. The function associated with each rendezvous adds and removes the affected job to the corresponding lists of jobs in a specific state. The scsh functions `status:exit-val`, `status:stop-sig`, and `status:term-sig` decode the `status` value returned by `job-status-rv`. Depending on whether the process exited, was suspended or terminated abnormally, these functions return `#f` or an integer providing further information on the reason of state change. If the operating system suspend the process, for example, `status:stop-sig` returns the signal number that yielded to suspension.

7. Related Work

There is multiplicity of file managers available that follow the tradition of the abandoned Norton Commander, such as the GNU MidnightCommander [6] or LFM [8]. These applications use most of the screen to display one or two file lists which the user may navigate, use to select files, and perform operations on them. The last line of the screen shows the shell prompt of a traditional shell. Thus, these applications are clearly committed to work with files solely. To Commander S, working with files is just one facet of a more holistic approach for easing the work with a shell. The GNU MidnightCommander comes with job control for background jobs but these "jobs" are merely running copying and moving operations. XEmacs and GNU Emacs ship with `dired`, a special mode for editing directory trees [10]. The GNU screen [4] terminal manager allows users to detach from a terminal and reattach to it later, and offers some text based copy and paste mechanism. This provides a functionality akin to Commander S's console jobs.

8. Conclusion and Future Work

This paper presented Commander S as a browser for UNIX. With the aid of command plug-ins, Commander S parses the output of commands and acquires the contained information. Viewer plug-ins use the ncurses library to present the output information as interactive content. Commander S contains plug-ins for the most common entities in shell interaction, processes, and filesystem contents. The paper shows that it is possible with little effort to extend Commander S to other domains. Through the use of the CML library, the implementation of the job control is very short, even though it is more powerful than in common UNIX shells and even contains a small terminal emulator for running processes in the background while saving their output.

The technique presented in this paper could be used to present other information such as DNS result records, or the contents of NIS or LDAP databases. As Commander S closely integrates an evaluator for Scheme expressions, the user can always fall back to writing small programs if the power of the command language or the viewers does not suffice to accomplish a task.

One conceivable extension of Commander S is the integration with the Orion window manager which is also based on Scsh. In this combination, Orion would start several Commander S instances concurrently, and assign every instance its own pseudo terminal and Xterm window.

Acknowledgments Christoph de Mattia wrote the `scsh-ncurses` bindings and an early prototype of Commander S called `scsh-nuit`.

References

[1] Sunterlib — the Scheme Untergrund library, 2005. Available at `http://www.scsh.net/resources/sunterlib.html`.

[2] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *ACM Conference on Lisp and Functional Programming*, pages 277–288, Snowbird, Utah, 1988. ACM Press.

[3] Eric Raymond, Zeyd Ben-Halim, and Thomas Dickey. *Writing programs with ncurses*, 2004.

[4] Oliver Laumann et al. *GNU Screen 4.0.2 user manual*, 2005. `https://savannah.gnu.org/projects/screen/`.

[5] Donald. A Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., 1994.

[6] Pavel Roskin and Miguel de Icaza. The GNU MidnightCommander, 2005. `http://www.ibiblio.org/mc/`.

[7] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[8] Iñigo Serna. lfm —last file manager, 2004. `http://www.terra.es/personal7/inigoserna/lfm/`.

[9] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber. *Scsh Reference Manual*, 2003. Available from `http://www.scsh.net/`.

[10] Michael Sperber. Dired. `http://www-pu.informatik.uni-tuebingen.de/users/sperber/software/dired/%`.

```
(define (spawn-joblist-surveillant)
  (let ((statistics-channel (make-channel)))
     (spawn (lambda ()
       (let lp ((running '()) (ready '()) (stopped '()) (new-output '())
                (waiting-for-input '()) (notify? #f))
         (cond
          (notify?
           (send statistics-channel ...)
           (lp running ready stopped new-output waiting-for-input #f))
          (else
           (apply select
            (append
             (list
              (wrap (receive-rv add-job-channel)
                    (lambda (new-job)
                      (lp (cons new-job running)
                          ready stopped new-output waiting-for-input #t)))
              (wrap (receive-rv notify-continue/foreground-channel)
                    (lambda (job)
                      (lp (cons job running) ready
                          (delete job stopped) (delete job new-output)
                          (delete job waiting-for-input) #t)))
              (wrap (receive-rv get-job-list-channel)
                    (lambda (answer-channel)
                      (send answer-channel ...)
                      (lp running ready stopped new-output waiting-for-input #f))))
             (map
              (lambda (job)
                (wrap (job-status-rv job)
                      (lambda (status)
                        (cond
                         ((status:exit-val status)
                          => (lambda (ignore)
                               (lp (delete job running) (cons job ready) stopped
                                   new-output waiting-for-input #t)))
                         ((status:stop-sig status)
                          => (lambda (signal)
                               (cond
                                ((= signal signal/ttin)
                                 (lp (delete job running) ready stopped new-output
                                     (cons job waiting-for-input) #t))
                                ((= signal signal/ttou)
                                 (lp (delete job running) ready stopped
                                     (cons job new-output) waiting-for-input #t))
                                ((= signal signal/tstp)
                                 (stop-job job)
                                 (lp (delete job running) ready (cons job stopped)
                                     new-output waiting-for-input #t))
                                (else (error "Unhandled signal" signal)))))
                         ((status:term-sig status)
                          => (lambda (signal)
                               (lp (delete job running) ready (cons job stopped)
                                   new-output waiting-for-input #t)))))))
                running))))))))
    statistics-channel))
```

Figure 7. Excerpt from the implementation of a job list with asynchronous status indication.

Ubiquitous Mail

Erick Gallesio
Université de Nice - Sophia Antipolis
930 route des Colles, BP 145, F-06903 Sophia Antipolis, Cedex, France
Erick.Gallesio@essi.fr

Manuel Serrano
Inria Sophia Antipolis
2004 route des Lucioles - BP 93 F-06902 Sophia Antipolis, Cedex, France
http://www.inria.fr/mimosa/Manuel.Serrano

ABSTRACT

Bimap is a tool for synchronizing IMAP servers. It enables two or more IMAP mirrored servers to be modified independently and later on, synchronized. Bimap is versatile so, in addition to synchronizing emails, it can be used for filtering and classifying emails. For the sake of the example, the paper shows automatic emails classification and white-listing programmed with Bimap.

Bimap is implemented in Scheme. The most important parts of its implementation are presented in this paper with the intended goal to demonstrate that Scheme is suited for programming tasks that are usually devoted to scripting languages such as Perl or Python. With additional libraries, Scheme enables compact and efficient implementation of this distributed networked application because the main computations that require efficiency are executed in compiled code and only the user configurations are executed in interpreted code.

1. Introduction

Low cost computers, ADSL, and wireless connections have made ubiquitous computing a reality. Because the Internet is now available nearly everywhere on the planet, most of us are nearly permanently connected. Many of us use various computers (maybe, one at home, one at work, and a roaming laptop). All these computers ideally use the same synchronized data. Enforcing this synchronization is not always so easy. Hopefully, some dedicated tools such as Unison [4] allow two replicas of a collection of files and directories to be stored on different hosts, modified separately, and then brought up to date by propagating the changes in each replica to the other. However, as convenient as these tools are for file and directory synchronization, they are of little help when considering email synchronization. In this paper, we address the specific problem of synchronizing email.

Bimap is a tool for synchronizing email. It enables emails to be manipulated from different computers and localizations. A user can read, answer, and delete emails from various computers amongst which some can be momentarily disconnected. Bimap automatically propagates the changes to all these computers. As demonstrated in this paper, synchronizing email is a simple problem of synchronizing lists. Functional languages are therefore candidates of choice for implementing such algorithms. Bimap is implemented in one of them, namely Scheme, our favorite programming language.

The rest of this paper is organized as follows. Section 2 presents the *Internet Message Access Protocol* which constitutes the foundation of Bimap. Section 3 shows the limitations of IMAP and it presents the general system architecture used by Bimap. Section 4 presents the synchronization algorithm and its implementation in Scheme. Section 5 presents *white-listing*, a filtering application implemented with Bimap.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

2. IMAP

The *Internet Message Access Protocol* (IMAP [1]) allows a client to access and manipulate electronic email messages on a server. It permits manipulation of mailboxes (remote message folders) as it also provides the capability for an offline client to resynchronize with the server.

In contrast with other protocols such as the *Post Office Protocol* (POP3 [3]), IMAP includes operations for creating, deleting, and renaming mailboxes, checking for new messages, permanently removing messages, creating new ones, etc. Hence, IMAP is a perfect match for our email synchronizer architecture. We have implemented a simple library that binds IMAP facilities in Scheme. It is briefly presented in this section.

Messages in IMAP are accessed by the use of numbers. These numbers are either message sequence numbers or unique identifiers. ¿From a programming point of view, the latter are much more convenient than the former. Message sequence numbers evolve as emails are created of deleted. On the other hand, each email is associated with a unique identifying number (UID henceforth) that remains valid during an IMAP session. Note that while UIDs unambiguously refer to messages in a given session, no provision is made by IMAP to make UIDs pervasive, i.e., the UID associated with an email may change from an IMAP session to another. Hence, it is difficult to implement a synchronization mechanism that relies on IMAP's UIDs. As seen in Section 4 our synchronization tool prefers to use the stamps allocated by the senders than IMAP's UIDs. These stamps are extremely likely to be unique and in practice we have never found a collision.

Instead of tediously presenting all the functions composing the library, we present in Figure 1 a typical example that illustrates the most important functions. The example shows an interactive session where a user logs in and browses some folders and emails.

Note that contrary to some other protocols, IMAP is stateful. The state of an IMAP connection describes the current directory. The command `imap-folder-select` sets it to a new value.

In addition to the functions used in the example, the library offers various functions for accessing the header elements, the attributes, and the body of the messages.

```
;; establish an ssl connection with the IMAP server
(define sock (make-ssl-client-socket "imap.nohwere.org" 993))
;; get logged in with user name and password
(imap-login sock "john doe" "eodjohn")
;; get the folders list
(imap-folders sock)
   -> ("INBOX" "INBOX.-Unknown" "INBOX.foo" ...)
;; go into a folder
(imap-folder-select sock "INBOX.foo")
;; get the list of messages in "INBOX.foo"
(imap-folder-uids sock)
   -> (33612 32977 29895 29132 29018 28958 26938 26937 26129)
;; get the message information about one email
(imap-message-info sock 33612)
   -> ((message-id . <429F0564.9040400@foo.com>)
       (date . 02-Jun-2005 15:16:09 +0200)
       (size . 6025) (flags \Seen) (uid . 33612))
;; remove one of these messages
(imap-message-delete! sock 33612)
;; create a new folder
(imap-folder-create! sock "INBOX.bar")
;; create a new message in the folder "INBOX.bar"
(imap-message-create! sock "INBOX.bar" "subject: foo\n\nFoo is not bar")
   -> 32739
;; get the whole header of message 32739
(imap-message-header sock 32739)
   -> '((subject . "foo"))
;; get the message 32739 body
(imap-message-body sock 32739)
   -> "Foo is not bar"
;; copy a mail accross server (usually the servers differ)
(imap-message-copy! sock 29132 sock "INBOX.bar")
;; move it into folder "INBOX.foo"
(imap-message-move! sock 32739 "INBOX.foo")
;; log out
(imap-logout sock)
```

Figure 1. An IMAP session in Scheme

3. The general architecture

IMAP is a distributed platform that allows clients to access email servers. IMAP is a big step in the direction of ubiquitous email because IMAP servers can be accessed by basically any computer connected to the Internet. In the first place, we have thought that one IMAP server would be sufficient to satisfy our needs. We have discovered that it is not. The difficulties are two fold. First we occasionally happen to manage our email using disconnected computers such as in-flight laptops. Even if some IMAP-capable email readers maintain a local copy of the emails and are able to work offline, we are seeking a neutral architecture that does not impose a specific mail reader. Second, we also happen to access our distant incoming IMAP server using Web email clients. These clients directly run on the computer hosting the server, modifying its state. This introduces a de-synchronization between the server and the local state of a disconnected laptop.

Generally, IMAP providers enforce quotas and maximal sizes for attachments. These IMAP servers do not provide enough resources for storing all emails. In consequence some emails have to be moved in local folders (e.g., user file system). The location and the format used for these locally stored emails highly depend on email readers. So, they are difficult to synchronize with general synchronization tools.

We came up with a solution that uses several IMAP servers: a main server (or *incoming server*) where the emails first arrive and one local server per computer. All the email clients only access the IMAP server that runs locally. Any modification to an email (deletion, access, ...) is thus local. On a regular basis all the servers are synchronized which maintains a coherent global state. Note that the emails on the *incoming server*, which is generally on a machine which is not under our control, can be a subset of the emails that are stored on our personal computers.

The Figure 2 illustrates a possible use of Bimap synchronization. For the sake of the example let's assume two servers receiving email (*provider1.edu* and *provider2.com*). Both servers only offer IMAP accounts. Let's also assume three machines used to read email (*Classic*, *Desktop* and *Laptop*). *Classic* is not synchronized. It remotely accesses *provider1.edu* and *provider2.com* via a direct IMAP connection. *Desktop* and *Laptop* have local folders that are synchronized with the servers. These two machines expose a set of folders which is the union of the emails located on the two servers and the local folders. In addition to synchronizing *Desktop* and *Laptop* with *provider1.edu* and *provider2.com*, Bimap is also in charge of synchronizing *Desktop* and *Laptop* together in order to ensure a correct synchronization of the email stored locally (and to back them up on another machine too).

4. The synchronization

In this section we present the synchronization algorithm. First, the principles are exposed. Then the specific parts of the synchronization are presented, as well as a sketchy presentation of their implementation.

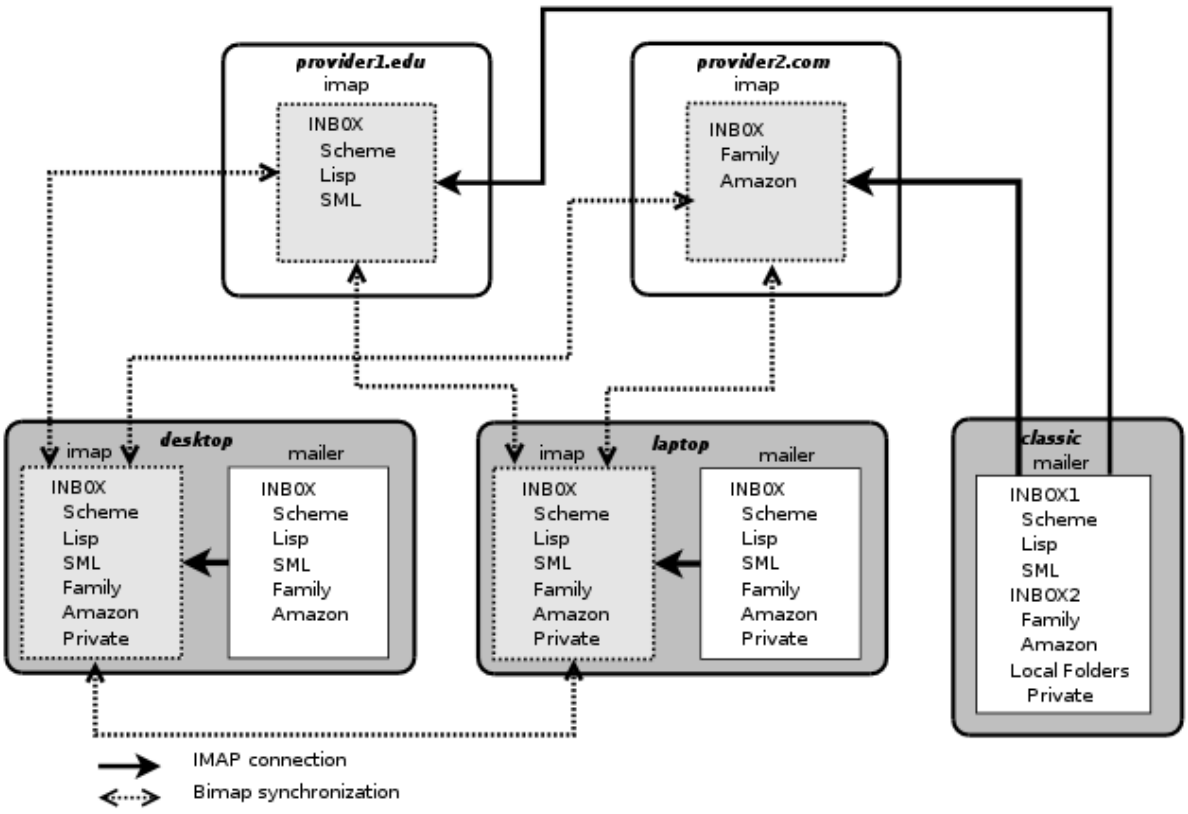


Figure 2. General architecture

4.1 The synchronization principles

The email synchronization relies on the `MESSAGE-ID` header fields [2]. Each of these fields contains a unique identifier which refers to *one* version of *one* email. The uniqueness of the email identifier is guaranteed by the host which generates it. The email identifier pertains to exactly one instantiation of a particular email. In the synchronization process presented here, two emails are considered equal (i.e., the same) if and only if their `MESSAGE-ID` is the same.

`MESSAGE-ID` are not guaranteed to be globally unique hence one could think to replace them with another identification mechanism. As exposed in Section 2, we have decided not to use IMAP's `UID` because they are not pervasive. Alternatively we could have used a checksum mechanism applied to the entire email. This would have had the nasty drawback of enlarged network traffic because the whole email body would need to be transfered for identification. Using the `MESSAGE-ID`, only one field is transfered.

The process of synchronizing email consists in switching from one synchronization state (*sync-state* in short) to another. A sync-state is an abstract notion. In Bimap, it is represented by a sync-table which is made of a list containing the `MESSAGE-ID` of the emails that have been synchronized. When a new synchronization takes place between two servers `S1` and `S2` with a sync-table `SYNCT`, first the lists `L1` and `L2` of emails present in `S1` and `S2` are computed. Then the emails of `L1` are compared to `SYNCT`. An email present in `L1` and absent in `SYNCT` is a new email that has to be copied in `L2` (and vice-versa). An email present in `SYNCT` but absent in either `L1` or `L2` is a deleted email. The implementation of this synchronization is presented in figures 5 and 6.

For the sake of the example, here is an example of a sync-table:

```
(("INBOX"
 ("<OFBF04EB.82F939-O125705F@us.ibm.com>" ((\Seen)))
 ("<1000.4779.9924@gardenia.artisan.com>" ((\Seen)))
 ("<169.62.5348.1708@gardenia.artisan.com>" ((\Seen)))
 ("<11280737.6750.7.camel@localhost.localdomain>" ())
 ("<2508011658.j71GTF002229@sea.inria.fr>" ((\Seen)))))
```

The sync-table is an association list whose `car` is a folder name (`"INBOX"` in this example) and the `cdr` is the list of synchronized messages. For each synchronized message, we retain its `MESSAGE-ID` as well as the list of its IMAP flags. Usage of sync-tables is explained in Section 4.5.

The synchronization only deletes emails that are present in the sync-table. So, if for any reason, such as network failures, the sync-tables are lost, the synchronization algorithm will resurrect emails deleted on only one server but it will never erroneously delete emails. When a message is copied from a server `S1` to a server `S2`, it preserves its `MESSAGE-ID`. Hence, if sync-tables are missing, the synchronization will not erroneously duplicate emails. Consequently, Bimap synchronization is conservative.

4.2 The synchronization state

Instead of being implemented as files in the user file space, sync-tables are stored in the IMAP servers they describe. Hence, users cannot easily modify sync-tables which are private to Bimap. This ensures the coherence of the sync-tables with respect to the servers they describe. In addition, this technique also avoids cluttering user

home space with various private files. Since IMAP servers only contains emails, a sync table is implemented as an email. Since only folders can be named in IMAP, the sync-table is stored as the single message of a well-known folder.

Here is the complete implementation of the sync-table shown in Section 4.1:

```
Subject: bimap Tue Aug 23 06:27:37 2005

(("INBOX"
 ("<OFBF04EB.82F939-O125705F@us.ibm.com>" ((\Seen)))
 ("<1000.4779.9924@gardenia.artisan.com>" ((\Seen)))
 ("<169.62.5348.1708@gardenia.artisan.com>" ((\Seen)))
 ("<11280737.6750.7.camel@localhost.localdomain>" ())
 ("<2508011658.j71GTF002229@sea.inria.fr>" ((\Seen)))))
```

When synchronizing servers `S1` and `S2` which are both referenced by a socket, the name of this folder on `S1` is the name of `S2` concatenated with the user login name on `S1`. The function `sync-folder-name` implements this naming:

```
(define (sync-folder-name s1 s2)
   (let ((n (string-replace
             (socket-hostname s1)
             (string-ref (imap-separator s2) 0)
             #\-)))
      (string-append (bimap-folder-name)
                     (imap-separator s2)
                     n "+" (socket-login s1))))
```

Some special attention has been paid to produce a legal name for the folder name. Since IMAP servers reserve one character (`"."` or `"/"`) as a folder separator, this character cannot be used in folder names. The function `imap-separator` returns the character used on the server. The function `server-sync-table-folder-select` goes into the folder of `S1` containing the sync-table of `S2`.

```
(define (server-sync-table-folder-select s1 s2)
   (let ((f (sync-folder-name s2 s1)))
      (and (folder-exists? s1 f)
           (and (= (folder-length s1 f) 1)
                (imap-folder-select s1 f)))))
```

The function `server-sync-table-get` reads either `S1`'s sync-table for `S2` or `S2`'s sync-table for `S1` if the former is absent.

```
(define (server-sync-table-get s1 s2)
   (cond
      ((server-sync-table-folder-select s1 s2)
       (with-input-from-string
             (imap-message-body
              s1 (car (imap-folder-uids s1)))
          read))
      ((server-sync-table-folder-select s2 s1)
       (with-input-from-string
             (imap-message-body
              s2 (car (imap-folder-uids s2)))
          read))
      (else
       '())))
```

The default case of `server-sync-table-get` is to return an empty sync-table. Remember from Section 4.1 that the algorithm is conservative. That is, if it happens that the sync-table is lost on both servers, the synchronization algorithm will resurrect emails deleted on only one server but in no case will it erroneously delete emails.

4.3 Synchronizing servers

The synchronization of two IMAP servers `S1` and `S2` is parameterized by `folders`, a list of folders to be synchronized. The function `synchronize-servers!`, presented in Figure 3, scans all the folders in the list. For each folder it checks if the folder is new, deleted, or to be synchronized in each server. This function updates the new sync-table in order to reflect the synchronizations that took place.

When all folders are synchronized, the new sync-table is stored on both servers. The function `server-sync-table-store!` accepts three parameters: the socket `S1` and `S2` accessing the servers and the new sync-table `nsync`. It computes the name of the folder where the sync-table on both servers (see `sync-folder-name` in Section 4.2) is to be stored.

When a folder `F` is missing on one server, it has to be determined first if `F` is a freshly created folder or an older one which has been deleted. In order to simplify the understanding of the source code, contrary to the actual implementation, we have duplicated the cases where a folder is either absent on `S1` or `S2`. The code, here duplicated, can easily be merged into a single function. The function `new-folder?` answers this question. A folder is new if at least one of the following conditions is met:

- It is not present in the sync-table (see Figure 4, line *3*).
- It contains sub-folders. Since, synchronization first checks sub-folders, if `F` is old, all its sub-folders would have been previously deleted (line *4*).
- In order to enforce conservativeness, a folder containing new emails (i.e., at least one non-synchronized email) is also considered new (line *5*).

4.4 Synchronizing folders

When a folder is present in both servers (Figure 3, line *25*), each email in this folder is inspected by `synchronize-folders!` defined Figure 5.

The function `sync-table-folders-find`, whose code is not given here, retrieves the information available in the sync-table about the folder `F`. That is, it searches the association list presented in Section 4.1. A hash table is built (line *8*) for improving the performance of the algorithm. It enables fast access to the sync-table.

4.5 Synchronizing messages

The last step of the algorithm is the synchronization of an email. The function `synchronize-message!` synchronizes a message `M1` localized in the folder `F` on server `S1` according to the sync-table `synct`. In addition to copying and deleting emails, this function also propagates the *flags* that are associated with emails. As specified by IMAP these flags denote meta-informations about emails such as *an email is read*, *an email is answered*, ... While not absolutely required, synchronizing flags is important. It enables coherent views of the email on all servers. In order to synchronize flags, Bimap stores the flags of synchronized emails in the sync-tables. That is, for each synchronized email, the sync-table denotes its flags on the servers at the moment of the last synchronization. The function `hashtable-message-flags` returns the flags stored in the sync-table for a given email. It returns either the list of the email flags or `#f` if it is out of synchronization. In the seldom situation where two servers have separately modified the flags of one email, Bimap randomly selects one server for synchronizing flags, loosing the modifications applied on the other server.

5. Filtering and classifying emails

Email has escaped the professional IT sphere. One now emails to colleagues as he does to relatives. Electronic merchandising also generates emails. Electronic billing and confirmation numbers are frequently sent by email. Many administrative procedures can also be completed with the Web and email. All in all this represents a huge number of emails that are sent (and also received) every day.

```
 1: (define (synchronize-servers! s1 s2 folders)
 2:    (let ((sync (server-sync-table-get s1 s2))
 3:          (folders1 (imap-folders s1))
 4:          (folders2 (imap-folders s2)))
 5:       (let loop ((folders folders)
 6:                  (nsync '()))
 7:          (cond
 8:             ((null? folders)
 9:              (server-sync-table-store! s1 s2 nsync))
10:             ((not (memq (car folders) folders1))
11:              ;; folder absent in s1
12:              (let ((f (car folders)))
13:                 (if (new-folder? sync s2 f)
14:                     (begin
15:                        (imap-folder-create! s1 f)
16:                        (let ((s (synchronize-folders! sync s1 s2 f)))
17:                           (loop (cdr folders) (cons s nsync))))
18:                     (begin
19:                        (imap-folder-delete! s2 f)
20:                        (loop (cdr folders) nsync)))))
21:             ((not (memq (car folders) folders2))
22:              ;; folder absent in s2, symmetric to absent in s1
23:              ...)
24:             (else
25:              ;; the folder is present in both servers
26:              (let* ((f (car folders))
27:                     (s (synchronize-folders! sync s1 s2 f)))
28:                 (loop (cdr folders) (cons s nsync))))))))
```

Figure 3. Server synchronization implementation

```
1: (define (new-folder? sync s f)
2:    (let ((dsync (sync-table-folders-find sync f)))
3:       (or (not dsync)
4:           (pair? (imap-subfolders s f))
5:           (any? (lambda (i) (not (assoc (message-id i) dsync)))
6:                 (begin
7:                    (imap-folder-select s f)
8:                    (map imap-message-infos (imap-folder-uids s)))))))
```

Figure 4. Is a folder new?

```
 1: (define (synchronize-folders! sync s1 s2 f)
 2:    ;; go into the synchronized folders
 3:    (imap-folder-select s1 f)
 4:    (imap-folder-select s2 f)
 5:    (let* ((l1 (map imap-message-infos (imap-folder-uids s1)))
 6:           (l2 (map imap-message-infos (imap-folder-uids s2)))
 7:           (fsync (or (sync-table-folders-find sync f) '()))
 8:           (synct (sync->hashtable fsync)))
 9:       ;; synchronize mails
10:       (for-each (lambda (m1)
11:                    (let ((m2 (find-mid (message-id m1) l2)))
12:                       (synchronize-message! synct f m1 s1 m2 s2)))
13:                 l1)
14:       (for-each (lambda (m2)
15:                    (let ((m1 (find-mid (message-id m2) l1)))
16:                       (synchronize-message! synct f m2 s2 m1 s1)))
17:                 l2)
18:       ;; returns a new synchronization state
19:       (let ((fsyncn (hashtable-map synct list)))
20:          (cons f fsyncn))))
```

Figure 5. Folder synchronization implementation

```
1:  (define (synchronize-message! synct f m1 s1 m2 s2)
2:     (let* ((mid (message-id m1))
3:            (uid1 (message-uid m1))
4:            (flags1 (message-flags m1))
5:            (flags (hashtable-message-flags synct mid)))
6:        ;; if flags is false (the message is not in the sync table)
7:        ;; then the message is un-synchronized
8:        (cond
9:           ((and (not flags) m2)
10:           ;; an un-synchronized mail, presents in s1 and s2
11:           ;; (e.g., sync-table lost)
12:           (let ((flags2 (message-flags m2))
13:                 (uid2 (message-uid m2)))
14:              (imap-message-flags-change! s2 uid2 flags1)
15:              (hashtable-put! synct mid (list flags1))))
16:          ((not flags)
17:           ;; an un-synchronized mail which does not exists in s2
18:           (imap-message-copy! s1 uid1 s2 f)
19:           (hashtable-put! synct mid (list flags1)))
20:          ((not m2)
21:           ;; a synchronized mail, removed from s2
22:           (imap-message-delete! s1 uid1)
23:           (hashtable-remove! synct mid))
24:          (else
25:           ;; a synchronized mail, present in s1 and s2
26:           ;; when flags differs they have to be synchronized
27:           (synchronize-message-flags! sync m1 m2 s1 s2)))))
```

Figure 6. Message synchronization implementation

Those of us that are used to communicate via the Internet are so overwhelmed by emails that tools are needed for reading, filtering, and classifying emails. In addition to synchronizing email, Bimap can easily be adapted to these tasks, in the spirit of tools such as `procmail`.

Variables declared via the macro `define-parameter` are called *Bimap parameters*. The puspose of `define-parameter` is threefold: it declares a variable, a function named after the parameter that returns the value of the variable, and a function that stores a new value in the variable. Here is an example of a parameter declaration and use:

```
(define-parameter bimap-verbose 0)

(for-each (lambda (o)
             (if (string=? o "-v")
                 (bimap-verbose-set!
                  (+ 1 (bimap-verbose)))))
          (command-line-arguments))
```

When started, Bimap loads a user configuration file that specifies which IMAP servers and folders have to be synchronized. This file can also contain various definitions that are used for email classification and email filtering. Instead of inventing a new little language for implementing these customizations, Scheme augmented with the IMAP binding library is used. In consequence, a Bimap execution uses compiled Scheme code for running the synchronization algorithm and interpreted Scheme code for running the user configuration code. This blending of compilation and interpretation enables high expressiveness without jeopardizing performance.

5.1 Classifying emails

Bimap can be adapted to enable automatic folder selection. Slightly modifying Bimap enables user customizations that automatically deliver incoming emails into dedicated folders. For instance, one may choose to archive emails emitted for a mailing list into dedicated folders or another user may choose to split professional email from personal email. This customization is specified in the new Bimap parameter, `bimap-folder-rewrite`. The value of this parameter is a procedure that accept four parameters: the connection to the IMAP server, the folder in which the email is currently stored, the message info, and its header fields.

Email classification takes place when a new email is detected on only one of two servers. Instead of copying the new email in the folder of synchronization (line *18*, Figure 6) it is copied into a folder whose name is computed by `bimap-folder-rewrite`. That is, line *18* is replaced with:

```
18: (let* ((hd1 (mail-header->list
19:              (imap-message-header s1 uid1)))
20:        (fdest ((bimap-folder-rewrite) s2 f2 m1 hd1)))
21:    (imap-message-copy! s1 uid2 s2 fdest))
```

The new email is copied in the `FDEST` directory (which defaults to `F2`). No other treatment is needed. Since this email is marked as synchronized as any other email, the next time the two servers are synchronized, the message will be moved in `S1` from folder `F1` to `Fdest`.

The following user configuration example illustrates how the parameter `bimap-folder-rewrite` is used to store the emails sent to a mailing list into a dedicated folder.

```
(let ((old-rewrite (bimap-folder-rewrite)))
   (bimap-folder-rewrite-set!
    (lambda (sock folder msg header)
       (let ((to (message-header-field header "to")))
          (if (equal? to "bigloo@sophia.inria.fr")
             "Bigloo"
             (old-rewrite sock folder msg header))))))
```

The email classification requires no extra synchronization treatment. That is, no provision is taken to ensure the synchronization of an email e that is stored into a re-written folder F. The next time a synchronization takes place, if the folder F on the list of synchronized folders, the message e will be automatically synchronized too. This framework require no implementation effort but it introduces a delay in synchronization. It takes two server synchro-

nizations to correctly classify such an email and propagate the classification to the servers.

5.2 Surviving Spam

Spam email is a plague. They clutter our mailboxes, threatening email usefulness. Spams are more and more numerous. The Google Gmail accounts of the authors of this paper are cluttered with approximatively 3000 to 6000 spam emails per month. That is, between 100 and 200 spam emails are received each day! The 20 to 30 legitimate emails that are received are literally lost in this ocean of ineptitude. Spam emails are terribly annoying because they are cumbersome, distracting, and polluting. So, it is a popular challenge to stop spams. Many research labs have started projects on this topic. Anti-SPAM software is widely available. The best of these systems do an impressively good job at stopping spams. They use more and more clever methods to decide, according to its content, if an email is spam or not. Bayesian filtering is one of them. Unfortunately, as good as these systems are, as with anti-viruses software, they are bound by their very nature to be late on spam: anti-spam filters cannot anticipate new spamming techniques. Even more pessimistically, we think that content-based filtering is a partial solution that could only produce middling results. What can be reasonably expected from such filters when applied to emails like:

```
 .oooo.o  .ooooo.  oooo    ooo oo.ooooo.
d88(  "8 d88' `88b  `88b..8P'   888' `88b
`"Y88b.  888ooo888    Y888'     888   888
o.  )88b 888    .o  .o8"'88b    888   888
8""888P' `Y8bod8P' o88'   888o  888bod8P'
                                888
                               o888o
```

Spammers can use other techniques for obfuscating emails. A lot of them attempt to fool Bayesian filters by introducing meaningless texts. This ranges from c*h*a*n*g*i*n*g the space character to replacing letters with numb8rs and n0nsense 4ccents. Presumably the most intriguing fooling technique swaps the letters composing the words. Aoccdrnig to rscheearch at Cmabrigde uinervtisy, it deosn't mttaer waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteres are at the rghit pclae. The rset can be a tatol mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae we do not raed ervey lteter by itslef but the wrod as a whole.

Content-based filtering is not good enough, it lets too much spams entering our mailboxes. To work around this problem, we have coupled content-based filtering with a more drastic approach: white-listing. This well known technique consists of accepting incoming emails only from authenticated users. We are using a very straightforward technique. We save all the email address of our correspondents into in a big database which compose our white-list. When a new email goes through the content-based filter, the address of the email sender is checked against the white list. If the sender is unknown, the email is moved into a special folder. Otherwise, it is directly delivered to the regular mail box folder. This technique is extremely effective. In our personal setting, white-listing succeeds in detecting 99.9% of spam and only a few legitimate emails go into the spam-dedicated folder. A vast majority of legitimate emails are correctly handled and are no longer lost in a forest of spam. The dedicated folder of unknown senders can be checked once in a while when time permits.

Implementing white-listing exercises email pre-filtering as described above. White-listing is trivial to implement because it only requires a hash table. In the following we assume that the email addresses are stored in the local file `~/.bbdb` and are organized according to the Emacs' Big Brother Data Base format [5]. Since this code takes place in the user configuration file, it can be easily adapted to satisfy everyone's needs.

```
(define *white* (load-bbdb "~/.bbdb"))

(define (unknown-mail? header)
   (not (hashtable-get
         *white* (header-field 'from header))))

(let ((old-filter (bimap-filter)))
  (bimap-filter-set!
   (lambda (sock folder msg header)
     (if (unknown-mail? header)
         (imap-message-move! sock msg "INBOX.-Unknown")
         (old-filter sock folder msg header)))))
```

6. Summary and Conclusion

We have presented Bimap, a tool for synchronizing IMAP servers. We have shown that with very few modifications to the synchronization algorithm, Bimap is also able to filter and classify email. As such, Bimap could be a potential replacement for `procmail`. This is highly convenient because it enables email filtering with simple small Scheme scripts. Two such scripts have been presented: one for classifying emails that belong to mailing lists and a second one for implementing white-listing. Each of these scripts is no more that a few lines of Scheme code.

In order to ease the reading of the present paper, a simplified version of the synchronization algorithm has been presented. Contrary to the code presented here, the actual implementation supports folder re-writing. That is, it enables synchronizing folder `F1` of server `S1` with folder `F2` of server `S2` without imposing name equality between `F1` and `F2`. This is convenient for managing different IMAP accounts intended for different purposes. This incurs a small additional implementation complexity, such as an indexing with two folders name in the sync-table (so the functions `new-folder?`, `synchronize-folders!`, `synchronize-message!`, and `sync-table-folders-find` no longer take only one folder name as parameter but two), but the main principles of the implementation stay the sames.

We are now permanently using Bimap for our own email. We have found that email classification and white listing coupled with Bayesian filtering (that only runs on our incoming email server) is highly effective to filter out nearly all illegitimate emails. Without pretending to have rediscovered the pleasure and excitement of answering our first 80's emails, we claim that Bimap significantly reduces the modern burden of coping with email. We are no longer disturbed by irrelevant emails arriving continuously in our mailboxes. This makes our professional life significantly nicer!

Bimap is not yet the perfect tool. It still needs improvement. In particular, IMAP does not support locking. This is quite unfortunate, because lacking locks makes it impossible to prevent situations where two servers simultaneously attempt to synchronize against a shared third server. In such a situation, inconsistencies might occur in the IMAP responses that cause Bimap to fail. As stated in Section 4.1 this is not critical because the only consequence of corrupted sync-tables is emails resurrection. In no case could it lead to erroneous email deletion.

Bimap is a realistic, yet simple, application written in Scheme. It benefits from the expressiveness of this language and, more importantly, it uses a feature that is frequently available in Scheme implementations: the blending of compiled and interpreted programs. For efficiency, the tree comparisons are compiled. For usability and convenience the user scripts are interpreted. Few other languages present these capabilities.

[1] Crispin, M. – **Internet Message Access Protocol** – RFC 3501, The Internet Society, 2003.
[2] Crocker, D. – **Standard for the format of ARPA Internet text messages** – RFC 822, Dept. of Electrical Engineering, University of Delaware, 1982.
[3] Myers, J. and Rose, M. – **Post Office Protocol - Version 3** – RFC 1939, Carnegie Mellon and Dover Beach Consulting, Inc., 1996.
[4] Pierce, B. and Vouillon, J. – **What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer** – MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.
[5] Zawinski, J. – **The Insidious Big Brother Database** – 20th century.

Implementing a Bibliography Processor in Scheme

Jean-Michel Hufflen
LIFC (FRE CNRS 2661)
University of Franche-Comté
16, route de Gray
25030 BESANÇON CEDEX
FRANCE
hufflen@lifc.univ-fcomte.fr

Abstract

We report an experience of implementing the MlBibTeX bibliography processor, a re-implementation of BibTeX with particular focus on multilingual features. First we describe the behaviour of this software and explain why we chose Scheme to implement the first public version. Then we give the broad outlines of our implementation and show how we took as much advantage as possible of the main features of Scheme. We also explain what we really missed and suggest some ways to improve these points.

Keywords MlBibTeX, bibliography processor, medium-sized programming in Scheme.

1. Introduction

This article reports an experience of implementing medium-sized software in Scheme. The 'philosophy' related to the definition of this programming language is that 'a very small number of [basic] rules [...] suffice to form a practical and efficient programming language,' as mentioned at the introduction of the current revision of Scheme [24]. So this article is an attempt to show how software can be developed using 'a very small number of basic rules.' Anyway, we do not regret to have developed our software in Scheme, but have some criticisms: we think they are constructive.

The program we have written using Scheme is a *bibliography processor*. More precisely, this is a re-implementation of BibTeX [36], the bibliography processor associated with the LaTeX word processor [29]. Reading this paper does not require precise knowledge of LaTeX and BibTeX, but we provide a brief introduction in order to make our purpose more precise. LaTeX is not an interactive word processor: first, users type a *source file*, then LaTeX processess this source file and produces an output file that can be displayed on a screen or printed on a laser printer. LaTeX, which uses TeX as typeset engine[1] [28], produces high-quality print outputs. In particular, it is able to hyphenate words correctly [28, App. H].

[1] TeX, defined by Donald E. Knuth [28], provides a general framework to format texts. To be fit for use, the definitions of this framework need to be organised in a *format*. There are several formats, the most well-known being LaTeX.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

Items of the bibliographical information cited throughout an article typeset with LaTeX can be denoted by an identifier, e.g.:

```
\cite{ziemianski2002a}
```

[33, § 12.2.1]—from a syntactic point of view, LaTeX commands begin with '\' and braces are used to surround arguments—and when BibTeX is used to build the 'References' section of the article, it searches *bibliography files* containing *entries*, e.g.:

```
@INPROCEEDINGS{ziemianski2002a,
        ...}
```

[33, § 13.2] and generates a file containing bibliographical *references* cited throughout the article. When LaTeX runs again, such references are typeset and appear as part of this article.

LaTeX's recent versions have greatly improved multilingual capabilities. For example, the command for specifying the beginning of a chapter and its title is:

```
\chapter{...}
```

[33, § 2.2] and the keyword put by LaTeX defaults to the English one, i.e., 'Chapter,' but can be '*Chapitre*' (resp. '*Kapitel*, ...) for a book written in French (resp. German, ...) In addition, LaTeX is able to switch to accurate patterns for hyphenating non-English words. The babel package[2] [33, Ch. 9] provides multilungal operations for LaTeX. Other packages do that, too, but the most multilingual one is babel, in the sense that this package processes all the natural languages it knows, without giving any privilege to a particular one. Therefore this package is especially suitable for mixing several languages within the same document. For example, if we want to write an article in Polish with some fragments in German, we declare:

```
\usepackage[german,polish]{babel}
```

the last option—here, 'polish'—gives the default language of the document. If we want to write 'Bus to Poznán,' the words 'bus to' being translated in German, we can use the `\foreignlanguage` command of the babel package:

```
\foreignlanguage{german}{Autobus nach} Pozna\'{n}
```

Besides, LaTeX is able to deal with 'foreign' characters, that is, accented letters (e.g., 'ń') and other diacritics, but in this paper, we do not go thoroughly into that, we just use LaTeX commands to produce such characters.

[2] W.r.t. LaTeX's terminology, a **package** is a collection of commands. Something belonging to the Lisp world and close to this notion is the system of *modules* in Common Lisp, controlled by the `*modules*` variable and the functions `provide` and `require` [47, § 11.8].

However, BibTeX's present version does not provide as many multilingual features as LaTeX's, even if the insertion of some multilingual aspects has been put into action [33, pp. 733–734 & § 13.5.2]. In fact, the commands of the babel package can be used within the values of BibTeX fields—as we showed in the previous example—and BibTeX will copy these values onto the files it generates. However, this method seems to us to be bad, because users of such bibliographical entries have to load the babel package with all the accurate options in any document. This package operates statically, that is, all the languages possibly used throughout a document must be declared as options of the `usepackage` command, located at the beginning of the document. In other words, using languages that are not selected when the babel package is loaded causes errors. That may be the case if files of bibliographical references use such language identifiers, put down in the bibliography data bases. In addition, using such LaTeX command in bibliography data bases is just a hack and obviously obstructs the generation of bibliographies for output formats other than LaTeX. Given these considerations, we started a new implementation, called MlBibTeX (for '**M**ulti**L**ingual BibTeX'). We roughly describe the behaviour of this program in Section 2 and explain why we have developed it in Scheme. Section 3 presents MlBibTeX's architecture and gives the guidelines of its development. Section 4 reviews what we enjoyed in Scheme and what we missed.

2. MlBibTeX

2.1 Purpose

We sketched BibTeX's behaviour in the introduction. Now we explain why MlBibTeX can be viewed as a 'better BibTeX,' especially for multilingual features.

Let us consider the entry given in Figure 1, concerning a novella included in an anthology. This novella, written in Polish by a Polish writer, is entitled '*Autobus nach Poznán*,' let us remark that two words of this title belong to the German language. If this novella is cited in an article written in Polish, the corresponding reference should look like:

> [1] Andrzej Ziemiański, *Autobus nach Poznań*. [W:] *Zajdel 2002*. Fabryka słów; Lublin 2002; strony 165–238.

as an item belonging to a `thebibliography` environment [33, § 12.1.2]. A *plain* bibliography style is used above, that is, items are labelled by numbers, and first names are not abbreviated. (Other styles exist—for example, within *alpha* styles, the label of an item is formed from the author's name and the year of publication—various examples are given in [33, Table 13.4].) Besides, let us recall that this reference is supposed to be put at the end of a document written in Polish. Let us have a look at the same reference, but within the bibliography of a document in English and showing the items of this bibliography according to English-speaking conventions as far as possible:[3]

> [1] Andrzej Ziemiański. Autobus nach Poznań. In *Zajdel 2002*, pp. 165–238, Lublin, 2002. Fabryka słów. No English translation.

That is, '*[W:]*' is replaced by 'In,' '*strony*' by 'pp.' for 'pages.' It is easy to see that such simple cases can be processed by means of *substitutions*, that is, by means of LaTeX commands for generating keywords—`\bblin`, `\bblpp`, ...—whose effect is language-dependent—'*in*' and '*pp.*' in English. Other cases are subtler. First, the order is not the same: for example, the address of the publisher

[3] W.r.t. MlBibTeX's terminology, such a convention is called *document-dependent approach* [18, § 4].

```
@INPROCEEDINGS{ziemianski2002a,
        AUTHOR = {Andrzej Ziemiański},
        TITLE = {[Autobahn nach] : german {Poznań}},
        BOOKTITLE = {Zajdel 2002},
        EDITION = 1,
        PAGES = {165--238},
        PUBLISHER = {Fabryka Słów},
        ADDRESS = {Lublin},
        NOTE = {[No English translation] ! english},
        YEAR = 2002,
        LANGUAGE = polish}
```

Figure 1. Example of MlBibTeX entry.

is given before its name in an English-speaking bibliography, after it in a Polish-speaking one. Second, the value associated with the `NOTE` field (see Figure 1), the '`[...] ! english`' notation means that the string surrounded by square brackets is put only if the language of the reference is '`english`.' Users could build an entry for a document, usable when the reference is to be put within an English-speaking bibliography, another entry for the same document, but usable within a French-speaking bibliography, and so on. As a consequence, the information common to these entries would be duplicated. The '`[...] ! ...`' construct avoids such a behaviour; more technical details about such switches are given in [18, § 2.3].

To show some difficulty related to the generation of multilingual bibliographies, let us go back to the Polish version of our reference and recall that the title of the `zemianski2002a` entry uses some German words, which are expressed by the '`[...] : german`' notation. To ensure that these words will be properly hyphenated if need be, we can generate the following text:[4]

```
\bibitem{zemianski2002}Andrzej Ziemia\'{n}ski,
\newblock \emph{\foreignlanguage{german}{Autobus
nach} {Pozna\'{n}}}. \bblin\ \emph{Zajdel 2002}.
\newblock Fabryka s{\l}\'{o}w; Lublin 2002;
\bblpp\ 165--238.
```

provided that the document uses the babel package, with at least the german option. This document's author may think that he does not need to write in German even if a work using German words in its title is cited. In such a case—the german option has not been selected—MlBibTeX does not put the `\foreignlanguage` command:

```
\bibitem{zemianski2002}Andrzej Ziemia\'{n}ski,
\newblock Autobus nach {Pozna\'{n}} ...
```

but some words might be hyphenated incorrectly. Besides, the babel package is not the only way to write texts in Polish: there exists a polski package [4, § F.7], in which case other commands should be used. More precisely, here is the text that will cause the 'foreign' (non-Polish) words to be hyphenated correctly when this package is loaded:

```
\bibitem{zemianski2002}Andrzej Ziemia\'{n}ski,
\newblock \emph{{\selecthyphenation{german}Autobus
nach} {Pozna\'{n}}} ...
```

These examples aim to give some idea about the complexity of MlBibTeX's task. The management of the language specifications (`LANGUAGE` field, '`[...] ...`') and multilingual packages is explained in more detail in [21]. Of course, such problems are

[4] Notice the use of the commands `\bblin` and `\bblpp` for the keywords used in bibliographies. We show how to manage them in [20]. The `\emph` command is for texts to be emphasised, the `\newblock` command is used by some document styles [33, Table 7.2, § 12.2.1].

unknown for 'old' BibTeX. As shown in Figure 1, MlBibTeX's syntax extends BibTeX's. Square brackets are syntactic markers in MlBibTeX, 'normal' characters in 'classical' BibTeX. Likewise, the `LANGUAGE` field, specifying the language of an entry for MlBibTeX, is ignored by BibTeX since unused fields are ignored.

2.2 Requirements

When LaTeX processes a document, it produces an output file and puts the information about bibliographical citations in an *auxiliary* file. For example, processing the '`\cite{zemianski2002a}`' citation will cause the '`\citation{zemianski2002a}`' string to be put into an auxiliary file. This file should contain the specification of the bibliography style to be used: e.g., '`\bibstyle{plain}`' for the 'plain' bibliography style. In fact, these auxiliary files are not TeX source files in the sense that they do not contain texts to be typeset, but the tokens these files use are the same from a syntactic point of view (cf. [33, § 12.1.3] for more details). Here is what is to be done by MlBibTeX:

(i) look into an auxiliary file for the keys cited throughout a document and the bibliography style to be used for this document;

(ii) search bibliography files for corresponding entries;

(iii) look into the beginning of the source file in order to get information about the multilingual packages used and try to determine the document's language;[5]

(iv) sort them (the sort used depends on the bibliography style,[6], it may also depends on the document's language);

(v) arrange them according to the bibliography style chosen.

Tasks (i) and (iii) require a TeX parser, whereas Task (ii) requires a parser of bibliography files. (Because of the compatibility mode for 'old' bibliography styles of BibTeX [16], another parser is required for such files, written using the bst language [35].) The directives for Tasks (iv) and (v) are put in bibliography style files. We see that such a bibliography processor has to manage several formalisms.

2.3 A bit of story

When we designed and implemented MlBibTeX's first version [13], we decided to develop it in C [25], for sake of efficiency and portability. In fact, we confess that we were surprised when TeX had been reimplemented as a new system $\mathcal{NTS}$[7] [43], programmed in Java [23]: it resulted in a program over 100 times slower than TeX [48]. We also were trying to propose an alternative for a program with good reputation of efficiency. We put into action a precise modular decomposition and a precise terminology to name our functions and variables for this first version [14], so using C to develop a program supported by precise methodology seemed to us to be good compromise between efficiency and maintainability.

This first version—we reported the experience we got in [14]—was able to deal with substitutions, that is, commands such as `\bblin` and `\bblpp` (cf. § 2.1). It was also able to process constructs such as '`[...] : ...`' and '`[...] ! ...`.' But when it was ready for use and when we were arranging the interface files—the different values to give to the '`\bbl...`' commands—we became aware that this prototype was not multilingual enough. As an example, putting the different field values concerning the `zemianski2002a` entry in the right order for documents in English and Polish would have led to complicated bibliography styles, very hard to maintain. As we explained in [17], we decided to get rid of the language used by BibTeX for bibliography styles [35]: that is an old-fashioned language, non-modular and only based on handling a stack, as it can be seen in [33, § 13.6]. As abovementioned, a compatibility mode exists [16], but the best way for developing bibliography styles is given by a new language, called nbst, for '**n**ew **b**ibliography **st**yles', close to XSLT [52], the language of transformations for XML[8] documents.

2.4 The nbst language

Within this new framework, parsing a bibliographical entry from a .bib file results in an XML tree, that is, the `zemianski2002a` entry given in Figure 1 can be viewed as the XML tree:

```
<inproceedings id="zemianski2002a"
               language="polish">
  <author>...</author>
  <title>...</title>
  ...
</inproceedings>
```

Processing such a tree can be done this way by using a *template* of the nbst language:

```
<nbst:template match="inproceedings">
  <!-- Putting the reference's label, text omitted. -->
  <nbst:apply-templates select="author"/>
  <nbst:apply-templates select="title"/>
  ...
</nbst:template>
```

Such a template is similar to those used in XSLT, it is invoked when the entry we are processing is rooted by the `inproceedings` element. The two `nbst:apply-templates` elements we mentioned in that sketch aim to look for templates matching the `author` and `title` elements respectively: if such templates exist, they are invoked. The main difference between XSLT and nbst: in the latter, a template can be refined for a particular language:

```
<nbst:template match="inproceedings"
               language="polish">
  ...
  <nbst:apply-templates select="title"/>
  ...
</nbst:template>
```

a template with the `language` attribute having higher priority than a template without. So this template is invoked when we are processing an `inproceedings` element for a Polish-speaking bibliography, whereas the template without `language` attribute is invoked in order to format `inproceedings` elements for bibliographies written in languages other than Polish (more precisely, in languages other than those put in all the `language` attributes of the templates matching `inproceedings` elements). This kind of inheritance is applied whenever we are looking for a template. For example, let us consider the following statement:

```
<nbst:apply-templates select="title"/>
```

When it is run, we are looking for a template whose `match` attribute is `title`. First, we are looking for a template whose `language` attribute is associated with the current language. In particular, if this `apply-templates` statement is run from the template with the `language` attribute associated with `polish`, we are looking for a template with `language="polish"`. If such a template exists, it is invoked. If not, a template matching `title` elements without

[5] BibTeX does not need this step, but MlBibTeX does.

[6] For example, the unsrt bibliography style of BibTeX leaves the entries unsorted: they are put according to the order of appearance within the document.

[7] **N**ew **T**ypesetting **S**ystem.

[8] Reading this article does not require advanced knowledge about XML. Readers interested in this metalanguage can refer to [38].

`language` attribute—that is, a default template—is invoked. Such organisation allows us to build several variants, for English- and Polish-speaking bibliographies, as shown in § 2.1. More technical details are given in [18].

Like in XSLT, the values of `match` attributes belong to the XPath language, used to address parts of an XML document [51]. In fact, our expressions selecting parts of a bibliographical item are very close to XPath's expressions, but we added some functions for operations difficult to perform with the functions provided by the first version of XPath (1.0, the normative document being [51]). For example, using the functions provided by standard XPath to capitalise some words in a title is tedious. In addition, multilingual features require some information included in TEX source files (cf. § 2.2), and are implemented by means of calling external functions: we go thoroughly into this choice in [21]. Obviously, it is preferable for such external functions to be written in a high-level programming language, more precisely, in a language that should ease operations on strings. Such a criterium puts C at a disadvantage: plenty of successful text-processing packages have been written in C, but the memory management is explicit, such operations like concatenation require functions whose use is far from obvious. We cannot require a bibliography style designer to be an experienced programmer in C. So, as we report in [19], we decided to develop MlBIBTEX's first public version (1.3) using Scheme. In particular, this choice allowed us to use the representation of SXML [27] as a Scheme implementation of our XML trees. So the bibliographical entry given in Figure 1 is represented as:

```
(inproceedings (@ (id "zemianski2002a")
                  (language "polish"))
               (author ...) (title ...) ...)
```

In addition, let us recall that `nbst` programs are XML texts. To parse them, we use SSAX [26], its outputs being SXML expressions. Among other tools related to SXML, we have also gained experience by studying the functions implementing SXPath [27], but have given our own implementation, in order to ease the calls of external functions. Likewise, we wholly put into action the implementation of `nbst`, as a 'super-XSLT' processor with a kind of inheritance about the `language` attribute.

2.5 A language accessible by end-users

The choice of a language with XML-like syntax for bibliography styles opens a window towards XML's world and some applications become easier: for example, using `nbst` to build a HTML file [53] from a bibliography file in order to display its entries on the Web. Or generating bibliographies for documents in DocBook, an XML-based system for writing structured documents [54]. But another problem occurs: it is well-known that many end-users puts LATEX commands inside values of BIBTEX fields, because 'old' BIBTEX itself does not have enough expressive power. We already mentioned this fact in the introduction about commands from the `babel` package. In fact, it does not matter if LATEX documents are generated—although it can be told that such behaviour makes difficult the sharing of bibliography files among several users because users have to load the same packages as abovementioned—but may cause errors on other cases. For example:

```
TITLE = {{\textsc{la}} Confidential}
```

In such a case—some letters to be typeset using small capitals—our parser of bibliography files can easily process this title by using an element with accurate attributes:[9]

```
<title>
  <asitis>
    <emph emf="no" scf="yes">la</emph>
  </asitis>
  Confidential
</title>
```

since the `\textsc` command is predefined in LATEX. The problem is more complicated if end-users put commands they have defined themselves, e.g.:

```
TITLE = {{\logo{la}} Confidential}
```

where `\logo` is a user-defined command meaning that its its argument is an acronym. Such a command may be defined as follows [33, § A.1.2]:

```
\newcommand{\logo}[1]{\textsc{#1}}
```

that is, the `\logo` command has the same effect than the `\textsc` command, but it is more readable about its meaning and can be redefined if users wish to change the display of acronyms.

This example shows that if end-users have put some LATEX commands inside values of BIBTEX fields and wish to use MlBIBTEX to output files according to other formats than LATEX, they should be able to specify how their commands have to be processed when bibliography files are parsed and transformed into XML trees. They can do that by means of the `define-pattern` function of MlBIBTEX, some examples being given in Figure 2. The first example shows how the previous `\logo` command can be processed: in this case, it is processed like the `\textsc` command (see the `title` and `emph` elements above).

Hereafter we sketch the effect of the `define-pattern` function, in order to show that end-users can easily customise the transformation of bibliography files into SXML trees. In particular, such a customisation is easy since Scheme allows powerful operations on strings nicely. If a language like C was still used for MlBIBTEX's implementation, this kind of specification would be tedious, or we would have to define a mini-language to do that.

The `define-pattern` function has two arguments. The first is a string viewed as a **pattern**, following the conventions of TEX for defining commands, that is, the arguments of a command are denoted by '`#1`,' '`#2`,' ... (cf. [28, Ch. 20]). If the second argument is a string, it specifies a replacement, the arguments of the corresponding command being processed recursively. The result—that is, the second argument—could be given as an SXML expression, but we wish a particular representation not to occur inside the Scheme code introduced by the `define-pattern` function: that is why we give it as a string whose content is expressed by means of 'usual' XML syntax.

This simple form can deal with many cases, but not all. If we look at the second example, we see how the `\textbf` command of LATEX is replaced by an `emph` element with accurate attributes: using bold face and non-italicised characters. That may be wrong, because `\textit{\textbf{...}}` produces both bold face and italicised characters[10] in LATEX. More expressive power is needed to deal with such cases. In the developed form of the `define-pattern` function, the second argument is a zero-argument function that results in a string, which is the replacement of the pattern. When this form is used, all the operations must be explicit within the body of this zero-argument function. In fact, the form:

[9] Hereafter the `asitis` element means that its contents should not be capitalised or uncapitalised, even if the bibliography style requires that. The `emph` element and its attributes specifies typographic effects, e.g., using small capitals in this example. Readers interested in a description of elements and attributes used within the XML versions of bibliography files can refer to [15]: that is an earlier version, but changes are slight.

[10] Readers interested in the font management in LATEX can refer to [33, Ch. 7].

```
(define-pattern "\\logo{#1}" "<emph emf='no' scf='yes'>#1</emph>")      ; 'scf' is a flag for 'small capitals.'
(define-pattern "\\textbf{#1}" "<emph emf='no' bff='yes'>#1</emph>")    ; 'bff' is for 'boldface flag.'
(define-pattern "\\textit{#1}" (lambda ()
                                 ;; Notice that the emf attribute of the emph element—a switch between roman and italicised characters—
                                 ;; defaults to yes, the other attributes default to no.
                                 (define-pattern "\\textbf{#2}" "<emph bff='yes'>#2</emph>")    ; Local pattern.
                                 (string-append "<emph>" (pattern-process "#1") "</emph>")))
```

Figure 2. Patterns for LaTeX commands in Scheme.

```
(define-pattern p s)
```

—where p and s are strings—is equivalent to:

```
(define-pattern p
  (lambda () (pattern-process s))
```

the `pattern-process` function belonging to MlBibTeX's program. The body of the function that is the second argument of `define-pattern` may include the specification of *local patterns*, as shown in the third example given in Figure 2. Let us consider the last two patterns shown in this figure: when an occurrence of a `\textbf` command is encountered, the local pattern of the third example is applied inside the argument of a `\textit` command, the 'global' pattern of the second example being applied anywhere else.

3. The program

3.1 MlBibTeX's architecture

In the previous section, we introduced to the main modules of MlBibTeX; now we show how they are put together. Figure 3 pictures MlBibTeX's architecture. This figure emphasises the data flow: given some citation keys extracted from an auxiliary (.aux) file, some bibliography (.bib) files are searched and the result is a list of bibliographical entries, given as SXML data. To do that, MlBibTeX's parser is enriched with a module for dealing with patterns. As shown in Figure 3, some patterns are predefined, some—like the pattern matching the `\logo` command in Figure 2—can be user-defined. The analysis of the .aux file also allows us to get information about a bibliography style. If we do not consider the compatibility mode for old .bst files, bibliography styles are written using the nbst language. These files are parsed using SSAX, grouped and 'semi-compiled,' in the sense that templates are rearranged in order to ease the determination of the template to be invoked when we are moving to a particular element. Each template results in a Scheme function after this pre-processing, and the bibliography processor applies such functions.

Like XSLT [52, § 16], nbst supports '`text`,' '`xml`' and '`html`' output modes.[11] There is also a `LaTeX` mode, taking into account some particular points of LaTeX's syntax. So, only the strings to be output are concerned by the differences between `text` and `LaTeX` modes. As examples—`nbst:text` is used to put a string *verbatim*, like the `xsl:text` element in XSLT—:

- `<nbst:text>%</nbst/text>` yields '`%`' in `text` mode, '`\%`' in `LaTeX` mode (in LaTeX, '`%`' introduces a comment [29, § 2.2.1], so it must be escaped to loose this property),
- `<nbst:text>&#163</nbst:text>`—the character numbered 163—yields this character ('£') in `text` mode and the command to produce it ('`\pounds`') [29, § 3.2.2] in `LaTeX` mode, this command being suitable whatever the encoding used by the word processor is.

As mentioned in § 2.4, the programs in .nbst files can use calls to external functions written in Scheme. That is not heretic: this feature—using external procedures—exists in XSLT. We use such external functions in Scheme to implement operations on strings, to program lexical ordering that depend on natural languages, and to search .tex files for information about the multilingual capabilities allowed by the user of the source files, as shown in Figure 3.

We can be asked for a question: 'why two languages: nbst and Scheme? why have we not used Scheme for the whole of a bibliography style?' Such a conventions would have made MlBibTeX close to the stylesheets written in DSSSL[12] [22], associated with SGML texts. But it was told that programming with DSSSL was difficult for style designers that are not experienced programmers. In fact, DSSSL is not declarative enough, if we compare it to XSLT or nbst. Besides, nbst allows refinements to be put into action without modifying an existing style directly. For example, if a Polish style designer finds out that the default version for a style does not fit the Polish requirements for the layout of a reference for an `inproceedings` entry, such requirements can be implemented by developing additional templates whose the `language` attribute is associated with `polish`. External functions written in Scheme should be used for low-level computation, for examples, for operations dealing with the different characters of a string. In fact, we think that style designers will not have to develop such functions, but they can do that if they wish.

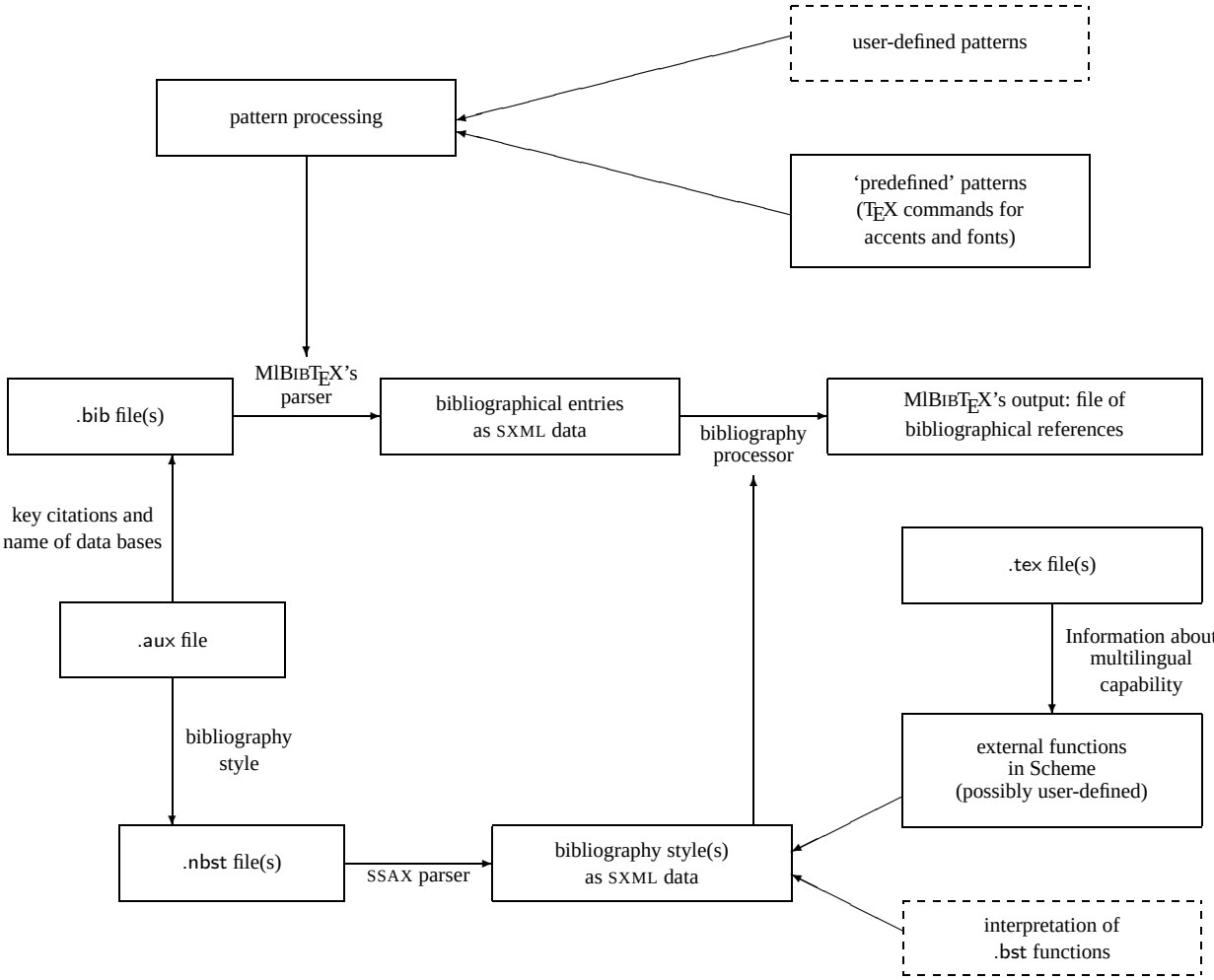
Last but not least, Figure 3 makes precise the parts that are finished presently: all, except that those pictured within a dashed box, they are planned for the next version.

3.2 Our programming

Working about natural languages is an open domain, in the sense that there is no general framework, from a theoretical point of view, that would cover all the natural languages in the world. What is suitable for a particular language may be unsuitable for another. So even if we consider a wide range of natural languages, we have to do experiments and other experiments, reprogram some parts if they have been modelled insufficiently, that is, if some particular cases made fail a general scheme. Only a high-level programming language allows such approach. Besides, the ability for end-users to enrich the program by means of patterns (cf. § 2.5) seemed to us to be a decisive point for choosing Scheme. Let us compare this feature with the Emacs editor, written in Emacs Lisp [31], and customisable by user-defined functions written in this language. Such issues seem to us to justify the choice of a LISP dialect. In addition, when we decided to do a second implementation using another language than C, we were familiar with LISP universe, we have already developed a medium-sized program in COMMON LISP: a rewrite engine for an algebraic specification language [10]. But we noticed that COMMON LISP was too big and heavy. We did not want to accept its complexity, whereas we needed only a

[11] A `html` mode is needed since HTML texts do not fit XML's syntax, stricly speaking.

[12] **D**ocument **S**tyle **S**emantics **S**pecification. This formalism is a side-effect free subset of Scheme, enriched by a library for formatting outputs.



'$A \leftarrow B$' means that A uses B. More precisely, functions or data put in A use functions of B or data from B.

Figure 3. Data flow in MlBIBTEX.

small part of it. We were interested in programming in a simpler LISP dialect, using only a few powerful constructs. In addition, we already have taught Scheme to undergraduate [11] and graduate [12] students.

Here are our rules of programming. Most of them aim to ease maintainability.

- There are precise rules for naming global variables. MlBIBTEX is organised into modules,[13] each module defining a prefix for naming variables. For example, '`pattern-`' is the prefix of the functions dealing with patterns (cf. § 2.5). Here are the exceptions:
  - some general functions and macros, grouped into one file,
  - local variables, that is, variables defined in the body of the special forms `define`, `do`, `lambda`, `let`, `let*`, and `letrec`,
  - *protected variables*, as we will see below, their names always end with '`-pv;`' when they are used in several modules, they do not have any prefix.
- Side effects are only allowed for local variables. In addition, we have carefully followed the recommendation about naming destructive functions in Scheme [24, § 1.3.5]: if a function mutates any of its arguments, its name ends with '`!`'
- Information is retained locally, by means of lexical closure and unlimited extent as far as possible. If several functionalities share the same environment, they are put into action by one function working by *message-passing*. This technique is used for *protected variables*: they are protected since they are enclosed within a lexical environment. For example, the bibliography style, as a path to a nbst program, is managed this way:

```
((bibliographystyle-pv 'see))      ; Get the value.
((bibliographystyle-pv 'set) ...) ; Update.
```

In fact, this technique can be viewed as object-oriented programming in Scheme, as shown in [1, Ch. 2] and [39]. We could have defined a global variable whose value is such a path and setting it whilst MlBIBTEX is running. We could put a syntactic sign inside its name to warn readers of our program that this variable is supposed to be modified. But we have preferred for

[13] From our point of view, these modules only exist in connection to *conception*, we do not use any syntactic feature—e.g., the `module` specification of the Scheme compiler bigloo [40, § 2.2] or PLT Scheme [6, § 5]—for them.

```
(define (parsers-make-launching filename launcher)
  ;; launcher is the function that rules the analysis of the input file. Its arguments are the function going forward through
  ;; the file and the function managing errors.
  (call-with-current-continuation (lambda (parser-exit-c)
                                    (parsers-filename-rp-loop filename launcher parser-exit-c))))

(define (parsers-filename-rp-loop filename launcher parser-exit-c)
  ;; filename being an absolute path to an existing file, opens it, runs a read-and-process loop, and closes the
  ;; corresponding port.
  (let ((input-p '*dummy-value*))
    (dynamic-wind
      ;; Even if the launcher function encounters errors, the input port is closed. The side effect on input-p is allowed
      ;; w.r.t. our conventions, because it is a local variable.
      (lambda ()
        ;; Reenter the middle thunk causes the input file to be open again:
        (set! input-p (open-input-file filename)))
      (lambda () (launcher (make-r-thunk input-p) parser-exit-c))
      (lambda () (close-input-port input-p)))))

(define (make-r-thunk input-p)
  ;; The result is a thunk—zero-argument function—that moves forward through the input file.
  (lambda () (read-char input-p)))

(define (make-x-function parser-exit-c)
  ;; The result is an escape function—'x' is for 'eXit'— that displays an error message, and stops reading through the input file.
  (lambda (msg-idf)
    (msg-manager msg-idf)
    (parser-exit-c #f)))
```

Figure 4. Basic functions to build MlBibTeX's parsers.

all the ways to get the value of such information or update it to be grouped into one function within our program.[14]

We did not use lexer and parser generator like those proposed in [34], analogous to LEX and YACC, which generates C programs [30]. In fact, we could have done that for the bst language, because lexical and syntactic analyses are clearly distinguished in this case. However, there is no distinction between scanner and parser in TeX's language,[15] also used in auxiliary files where information about bibliographies to be build are located (cf. § 2.2). For this language, there is only one analyser, which returns either a whitespace character, or another character, different from '\', or the complete name of a macro. Concerning bibliography files, we can separate lexical and syntactic analysis—we did that in the first version [13, Annex]—but that yields a two-level grammar: a first level for entries ('@...{...}'), a second for values associated with field names. So, we have preferred to develop *ad hoc* parsers for these languages. Last, we use the SSAX parser for nbst programs, since they are XML documents.

We have defined a common framework for the parsers we have built ourselves, the main functions are given in Figure 4. By convention, the arguments of the parser's functions include a zero-argument function to move forward through the input file and an escape function stopping reading through the input file.[16] Since this zero-argument function is our only way to get something from the current input file inside the functions of our parser, we do not 'unread' a character.[17] On the other hand, a parser is reading in advance. The solution put into action is that the functions of our parser return at least two values: the result of processing a fragment of the input file, and the first character belonging to the token after what has just been processed. A simple example is given in Figure 5. These parsers were easy to debug: we replaced the function moving forward through an input file by a function given in Figure 6 and exploring successive characters of a string.[18] as the `read-char` function would do after opening a string port in the sense of SRFI[19] Nr. 6 [3].

Concerning the management of multilinguism, the information related to natural languages used throughout bibliography data bases is organised into a *trie*:[20] see [21] for more details.

4. Scheme as an implementation language

First we developed MlBibTeX's present version with MIT Scheme [2, 9]. Then we study how to put a portable implementation into action with bigloo [40] and PLT Scheme [6, 37]. We carefully grouped non-portable code in one file, so we knew which parts could be difficult to adapt.

[14] In addition, if we consider a variable defined globally and updated at run-time, it can be difficult to detect that it has not been assigned yet to its 'actual' value. We could define it by bounding it to a 'dummy value', but there is no 'universal dummy value.'

[15] That is the case for some early languages.

[16] In particular, this function is called when an error is encountered. There is no error recovery in MlBibTeX—our parsers stop as soon as an error is encountered—but there was not in 'old' BibTeX, either.

[17] In fact, we could use the `peek-char` function of Scheme [24, § 6.6.2] for this operation, but we decided to proceed only ahead, homogeneously.

[18] Besides, this function is used in 'final' MlBibTeX: when an abbreviation, defined by '`@STRING{schw = {Scheme Workshop}}`'—cf. [33, § 13.2.3]—is used, e.g., in '`BOOKTITLE = schw`', MlBibTeX's parser inserts the contents of the string associated with '`schw`' by means of the `make-r-string-thunk` function.

[19] Scheme **R**equest **F**or **I**mplementation. For more details, see the Web page `http://srfi.schemers.org`.

[20] A **trie** is a particular case of a tree for storing strings: there is only one node for every common prefix.

```
(define (s-parse-string-def r-thunk char x)
  ;; 's-' is the prefix for functions parsing bibliography files. Parses '@STRING{<token-0> = <string-value>},' '@STRING' being
  ;; recognised, char being the first character after. r-thunk is the 0-argument function that allows us to move forward through the
  ;; input file, x is the escape function that stops reading and returns #f as the global result of parsing.
  (call-with-values (lambda ()
                      (s-next-bibtex-idf r-thunk
                                         ;; Checking that the token beginning with char is '{' and returning the first character
                                         ;; after, in case of success:
                                         (s-recognise-left-brace r-thunk char x)
                                         x))
    (lambda (token-0 char-0)
      :: token-0 is the abbreviation's name, char-0 is supposed to be '='.
      (call-with-values (lambda () (s-parse-value r-thunk (s-recognise= r-thunk char-0 x) x))
        (lambda (string-value char-1)
          ((s-string-defs-pv 'add) token-0 string-value)   ; Adds the binding token-0 ↦ string-value. Let us notice that
                                                           ; s-string-defs-pv is a protected variable (cf. § 3).
          ;; First, recognising '}' and returning the first character after, then processing next entry, that is, next '@{...}' and
          ;; returning two values:
          (s-next-entry r-thunk (s-recognise-right-brace r-thunk char-1 x)))))))
```

Figure 5. How our parsers use multiple values.

Our only error related to portability was an occurrence of the *false* value inadvertently replaced by the empty list.[21] Another portability problem arose from accented letters typed by using an encoding which extends ASCII:

```
(char-alphabetic? #\é) ⟹MIT Scheme          #t
(char-alphabetic? #\é) ⟹bigloo, PLT Scheme  #f
```

In reality, such a case is unspecified by the standard Scheme since this standard does not specify whether or not a character like '#\é' is a letter and since the `char-alphabetic?` function can only be applied to letters. Anyway, porting MlBibTeX raises a very small number of problems, but difficult, because they were related to features outside the standard Scheme. In fact, most of the issues mentioned hereafter are not MlBibTeX-specific and have already been debated, but we mention them, as a short report of our experience and as additional examples of these problems.

4.1 What we have liked

A common pitfall for Scheme programmers is the order of evaluation of a function's arguments: it is left unspecified by the Scheme reports [24, § 4.1.3] and may vary from an interpreter to another in practice. To be honest, the absence of a fixed order may look strange at first glance, but we think that it is straightforward, it forces programmers to emphasise what is sequential within their programs, most often by using the special forms `let` or `let*`.

As far as possible, we use Scheme as a functional programming language, in the sense that functions can be arguments or results of other functions. Since Scheme has only one namespace, that is, functions are particular values for variables, our program looks homogeneous. In COMMON LISP or other LISP dialects where functions belongs to a particular namespace [47, § 5.2], distinct from the 'other' variables, we would have had to add many occurrences of the `function` special form and the `funcall` function, what would complicate the programming.

Advanced functions like `call/cc` and `dynamic-wind` [24, § 6.4] are used in MlBibTeX (cf. Figure 4). However, let us mention that wherever we use these functions, simplified forms, as they are provided by COMMON LISP would have been sufficient: dynamically-scoped exits, by means of the special forms `catch` and `throw` [47, § 7.11], and the special form `unwind-protect`.

Dealing with multiple values is very common within the source files of MlBibTeX, an example being given in Figure 5, many other examples existing for functions dealing with multilingual information. A new special form such as `let-values`, as suggested by SRFI 11 [8], would simplify these examples.

4.2 What we have missed

The functions dealing with input files, `open-input-file` and `call-with-input-file`, signal an error if the file cannot be opened. But by using only the forms of the Scheme standard, we cannot know this information before trying this operation. The same problem arises from the functions dealing with output files, `open-output-file` and `call-with-output-file`. This can be solved by means of *conditions*—this notion exists in COMMON LISP [47, Ch. 29], but not (yet?) in the standard Scheme[22]—as suggested by SRFI 36 [44].

Some interpreters—MIT Scheme [9, § 5.7], bigloo [40, §§ 4.1.8 & 4.1.10]—allow characters to be processed using Unicode [49], but only partially. That should be added in the future standard, since more and more information will be encoded according to some extensions of the ASCII code: latin-1 (or ISO–8859–1) for West-European languages,[23] latin-2 for East-European ones, ... Unicode precisely redefines what letters, signs are. Proposals for putting these definitions in Scheme are SRFI 14 [41] and 75 [7]. As mentioned at the beginning of this section, some interpreters presently diverge about this point, which should be refined for further versions of Scheme.

We especially missed an interface with the operating system, in the sense of a function that would have launched a command of the operating system, and be able to retain its result displayed on the current output port, this result being a string usable by the functions of Scheme. From a general point of view, we think that in the standard Scheme, such a function would be more useful than special interfaces with specific programming languages like C or Java[24] [40, §§ 15 & 16]. More specifically, software belonging to TeX's world usually call functions of the kpathsea library [50], used for locating files. For example, the bibliography styles used by 'old' BibTeX can be located by means of the `kpsewhich` command:

[21] Let us recall that in MIT Scheme, `#f` and `()` are still the same object [9, § 1.2.5].

[22] ... but some Scheme interpreters incorporate them: e.g., MIT Scheme [9, Ch. 16].

[23] Internally used in MIT Scheme [9, § 5.5].

[24] Besides, this function could be used to run the compiled form of a program written using these languages.

```
(define (make-r-string-thunk string-0)
  ;; Returns a thunk exploring each character of string-0, in
  ;; turn. When the end of this string is reached, #f is
  ;; returned.
  (let ((string-length-0 (string-length string-0))
        (index 0))
    (lambda ()
      (if (< index string-length-0)
          (let ((result (string-ref string-0 index)))
            (set! index (+ index 1))
            result)
          #f))))
```

Figure 6. Moving forward through a string.

```
kpsewhich plain.bst
.../texmf/bibtex/bst/base/plain.bst
```

In order to put a similar feature into action for MlBibTeX, a workaround was to implement a simplified version of this command in Scheme. This implementation is not wholly satisfactory from a point of view related to portabiblity because this command uses *environment variables*, inaccessible directly from standard Scheme functions: `BIBINPUTS`, `TEXBIB` for 'old' BibTeX,[25]. MlBibTeX uses first the environment variable `MLBIBINPUTS`, before considering those of BibTeX [20].

Last, Scheme could include *packages* in the sense of Common Lisp [47, § 11.2], a simpler version being sufficient. If we develop software under the predefined functions of Scheme, a good discipline for naming functions is sufficient to avoid name clashes. But packages would ease software composition. For example, there is no document explaining how functions and macros of SXML have been named. So we had to be very careful to this point when we decided to use this software for dealing with XML documents.

4.3 Proposals

In [42], Dorai Sitaram writes that 'the [IEEE] Scheme standard and the Scheme reports do not define a useful programming language for all platforms. Instead they [...] define a family of programming languages that individual implementors can instantiate to a concrete programming language for a specific platform.' That is true, but what does it mean in practice? That an ambitious program rhas to rely on a particular dialect? Such dependence seems to us to be acceptable for a program using special effects (e.g., graphical parts), but is strange for functionalities related to a simple interface with an operating system (e.g., file existence). Besides, each dialect obviously provides such a function, and most often under the same name: `file-exists?` in MIT Scheme [9, § 15.3], in bigloo [40, § 4.2.2], in PLT Scheme [6, § 11.3.3]. Naming them homogeneously should be possible. Other examples are subtler, because functions are not known under the same name: if we wish to get the values of environment variables set at the operating system level (cf. § 4.2), the function is `get-environment-variable` in MIT Scheme [2, § 2.6], `getenv` in bigloo [40, § 4.2.1] and PLT Scheme [6, § 15.4]. Analogous points can be noticed about the functions passing a command to the operating system level.

In the foreword of [45], Guy L. Steele Jr. wrote: '[...] Small is easy to understand. I like the Scheme programming language because it is small.' But Scheme can include a small interface with basic services of operating systems and be still small. Such a small interface would not give Common Lisp's complexity to Scheme. It may be difficult to decide about the names to be given to the functions of this interface, because some software already use some functions specific to particular interpreters, so it would be tedious to rename them. A workaround could be an additional predefined variable whose value would group the whole information about the present interpreter, its name and version number, the running operating system, etc. The purpose of the zero-argument function `identify-world` of MIT Scheme [2, § 2.1] is close, but such information is only displayed when the function is applied and cannot be retained in a variable since this function does not return any result. Such a variable had been defined in Common Lisp: `*features*` returns a list of *features* characterising a particular implementation [47, § 25.4.2]. Features have also been proposed in SRFI 0 [5], that seems to us to be a promised way. In particular, such ways a variable would ease the writing of a tool like SCMXLATE [42], a software for porting Scheme programs from a dialect to another.

[25] However, we had to consider these environment variables for sake of compatibility with BibTeX.

5. Conclusion

When we teach Scheme to undergraduate students, some of them asks us about using this language in 'real' situations. Our personal opinion is that Scheme is certainly less used than an imperative language like C, or a language in fashion like Java. However, some medium-sized projects have been programmed using Scheme, and often the use of this language in such cases was successful. A good illustration of that is MlBibTeX. Doing the second implementation in Scheme was faster than doing the first in C, and performances are comparable. Surely, it is well-known that the higher the programming language's level, the faster the development. And to be honest, many problems had already been specified and solved for the first version, so often adapting C structures to Scheme ones was sufficient. But on the other hand, the second implementation proposes many more functionalities.

At the time of writing, we are working on MlBibTeX's installation, in order for this program to be able to work with a great number of Scheme interpreters. We think that we could succeed by using GNU tools such as make [46] and autoconf [32].

We enjoyed programming MlBibTeX in Scheme. We hope that we could go on with our implementation. We think that we could do better for future versions, especially about processing Unicode characters, according to an interpreter-independent way. So Scheme will be a modern language, since the localisation of software, including the use of several writing systems, is current challenge. Likewise, we hope that installing software programmed using Scheme will become easier. So Scheme will be not only 'an efficient and practical programming language' [24], but it will be more portable and more suitable for the modern types of strings.

Acknowledgements

I am very grateful to the anonymous referees, who allowed me to improve the first version of this article substantially.Many thanks to Michael Sperber, too, for his patience when he was waiting for this article.

References

[1] Harold Abelson and Gerald Jay Sussman: *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company. 1985.

[2] Stephen Adams, Chris Hanson and the MIT Scheme team: *MIT Scheme User's Manual*, 1st edition. June 2002.

[3] William D. Clinger: *Basic String Ports*. July 1999. `http://srfi.schemers.org/srfi-6/`.

[4] Antoni Diller: *LaTeX wiersz po wierszu*. Wydawnictwo Helio, Gliwice. Polish translation of *LaTeX Line by Line* with an additional annex by Jan Jelowicki. 2001.

[5] Marc FEELEY: *Feature-based Conditional Expansion Construct*. May 1999. http://srfi.schemers.org/srfi-0/.

[6] Matthew FLATT: PLT *MzScheme: Language Manual. Version 299.100*. March 2005. http://download.plt-scheme.org/doc/299.100/mred.pdf.

[7] Matthew FLATT and Marc FEELEY: *R6RS Unicode Data*. July 2005. http://srfi.schemers.org/srfi-75/.

[8] Lars T. HANSEN: *Syntax for Receiving Multiple Values*. March 2000. http://srfi.schemers.org/srfi-11/.

[9] Chris HANSON, THE MIT SCHEME TEAM *et al.*: MIT *Scheme Reference Manual*, 1st edition. March 2002. Massachusetts Institute of Technology.

[10] Jean-Michel HUFFLEN : *Fonctions et généricité dans un langage de programmation parallèle*. Thèse de doctorat, Institut National Polytechnique de Grenoble. Juillet 1989.

[11] Jean-Michel HUFFLEN : *Programmation fonctionnelle en Scheme. De la conception à la mise en œuvre*. Masson. Mars 1996.

[12] Jean-Michel HUFFLEN : *Programmation fonctionnelle avancée. Notes de cours et exercices*. Polycopié. Besançon. Juillet 1997.

[13] Jean-Michel HUFFLEN: "MlBibTeX: a New Implementation of BibTeX". In: *EuroTeX 2001*, pp. 74–94. Kerkrade, The Netherlands. September 2001.

[14] Jean-Michel HUFFLEN: "Lessons from a Bibliography Program's Reimplementation". In: LDTA *2002*, Vol. 65.3 of ENTCS. Elsevier, Grenoble, France. April 2002.

[15] Jean-Michel HUFFLEN: "Multilingual Features for Bibliography Programs: From XML to MlBibTeX". In: *EuroTeX 2002*, pp. 46–59. Bachotek, Poland. April 2002.

[16] Jean-Michel HUFFLEN: "Mixing Two Bibliography Style Languages". In: LDTA *2003*, Vol. 82.3 of ENTCS. Elsevier, Warsaw, Poland. April 2003.

[17] Jean-Michel HUFFLEN: "European Bibliography Styles and MlBibTeX". TUG*boat*, Vol. 24, no. 3, pp. 489–498. EuroTeX 2003, Brest, France. June 2003.

[18] Jean-Michel HUFFLEN: "MlBibTeX's Version 1.3". TUG*boat*, Vol. 24, no. 2, pp. 249–262. July 2003.

[19] Jean-Michel HUFFLEN: "A Tour around MlBibTeX and Its Implementation(s)". *Biuletyn* GUST, Vol. 20, pp. 21–28. In *BachoTeX 2004 conference*. April 2004.

[20] Jean-Michel HUFFLEN: "Making MlBibTeX Fit for a Particular Language. Example of the Polish Language". *Biuletyn* GUST, Vol. 21, pp. 14–26. 2004.

[21] Jean-Michel HUFFLEN: *Managing Languages within MlBibTeX*. Will be presented at PracTeX conference, Chapel Hill, North Carolina. June 2005.

[22] International Standard ISO/IEC 10179:1996(E): DSSSL. 1996.

[23] *Java Technology*. June 2005. http://java.sun.com.

[24] Richard KELSEY and William D. CLINGER, eds.: "Revised^5 Report on the Algorithmic Language Scheme". HOSC, Vol. 11, no. 1, pp. 7–105. August 1998.

[25] Brian W. KERNIGHAN and Denis M. RITCHIE: *The C Programming Language*. 2nd edition. Prentice Hall. 1988.

[26] Oleg KISELYOV: "A Better XML Parser through Functional Programming". In: *4th International Symposium on Practical Aspects of Declarative Languages*, Vol. 2257 of LNCS. Springer. 2002.

[27] Oleg KISELYOV and Kirill LISOVSKY: "XML, XPath, XSLT Implementations as SXML, SXPath, and SXSLT". In: *International Lisp Conference 2002*. San Francisco, California. October 2002.

[28] Donald Ervin KNUTH: *Computers & Typesetting. Vol. A: the TeXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1984.

[29] Leslie LAMPORT: *LaTeX. A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1994.

[30] John LEVINE, Tony MASON and Doug BROWN: *lex & yacc*. 2nd edition. O'Reilly & Associates, Inc. October 1992.

[31] Bill LEWIS, Dan LALIBERTE, Richard M. STALLMAND and THE GNU MANUAL GROUP: GNU *Emacs Lisp Reference Manual for Emacs Version 21. Revision 2.8*. January 2002. http://www.gnu.org.

[32] David MACKENZIE, Ben ELLISTON and Akim DEMAILLE: *autoconf. Creating Automatic Configuration Scripts. Version 2.59*. November 2003. http://www.gnu.org/software/autoconf/manual/.

[33] Frank MITTELBACH, Michel GOOSSENS, Joannes BRAAMS, David CARLISLE, Chris A. ROWLEY, Christine DETIG and Joachim SCHROD: *The LaTeX Companion*. 2nd edition. Addison-Wesley Publishing Company, Reading, Massachusetts. August 2004.

[34] Scott OWENS, MATTHEW FLATT, Olin SHIVERS and Benjamin MCMULLAN: "Lexer and Parser Generators in Scheme". In: *Proc.* ACM SIGPLAN *2004 Scheme Workshop*, pp. 41–52. Snowbird, Utah. September 2004.

[35] Oren PATASHNIK: *Designing BibTeX Styles*. February 1988. Part of BibTeX's distribution.

[36] Oren PATASHNIK: *BibTeXing*. February 1988. Part of BibTeX's distribution.

[37] PLT: PLT *MzLib: Libraries Manual. Version 299.100*. March 2005. http://download.plt-scheme.org/doc/299.100/mzlib.pdf.

[38] Erik T. RAY: *Learning* XML. O'Reilly & Associates, Inc. January 2001.

[39] Jonathan A. REES and Norman I. ADAMS IV: "Object-Oriented Programming in Scheme". In: *Proc. of the 1988* ACM *Conference on Lisp and Functional Programming*, pp. 277–288. Snowbird, Utah. 1988.

[40] Manuel SERRANO: *Bigloo. A Practical Scheme Compiler. User Manual for Version 2.6c*. June 2004.

[41] Olin SHIVERS: *Character-set Library*. December 2000. http://srfi.schemers.org/srfi-14/.

[42] Dorai SITARAM: "Porting Scheme Programs". In: *Proc. of the 4th Workshop on Scheme and Functional Programming, UUCS–03–023*, pp. 69–74. School of Computing, University of Utah, Boston, Massachusetts. November 2003.

[43] Karel SKOUPÝ: "The Software Quality and $\mathcal{NTS}$". *GUST*, Vol. 16, pp. 41–49. 2001.

[44] Michael SPERBER: *I/O Conditions*. June 2003. http://srfi.schemers.org/srfi-36/.

[45] George SPRINGER and Daniel P. FRIEDMAN: *Scheme and the Art of Programming*. The MIT Press, McGraw-Hill Book Company. 1989.

[46] Richard M. STALLMAN, Roland MCGRATH and Paul SMITH: GNU *make. A Program for Directing Recompilation. Version 3.80*. July 2002. http://www.gnu.org/software/make/manual/.

[47] Guy Lewis STEELE, JR.: COMMON LISP. *The Language. Second Edition*. Digital Press. 1990.

[48] Philip TAYLOR, Jiři ZLATUŠKA and Karel SKOUPÝ: "The $\mathcal{NTS}$ Project: from Conception to Implementation". *Cahiers GUTenberg*, Vol. 35–36, pp. 53–77. May 2000.

[49] THE UNICODE CONSORTIUM: *The Unicode Standard Version 4.0*. Addison-Wesley. August 2003.

[50] TUG Working Group on a TeX Directory Structure: *A Directory Structure for TeX Files. Version 0.9995*. CTAN:tex/archive/tds/standard/tds-0.9995/tds.dvi. January 1998.

[51] W3C: XML *Path Language (XPath). Version 1.0*. W3C Recommendation. Edited by James Clark and Steve DeRose. November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[52] W3C: XSL *Transformations (*XSLT*). Version 1.0*. W3C Recommendation. Edited by James Clark. November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[53] W3C: *HyperText Markup Language Home Page*. May 2005. `http://www.w3.org/MarkUp/`.

[54] Norman WALSH and Leonard MUELLNER: *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc. October 1999.

The Marriage of MrMathematica and MzScheme

Chongkai Zhu
mrmathematica@yahoo.com

Abstract

In this paper, I argue that the programming languages provided in current mainstream CASes are not suitable for general purpose programming. To address this problem, I developed MrMathematica. MrMathematica is a connection between Mathematica and PLT-Scheme, which provides the ability to call Mathematica from MzScheme. The two languages share some common ground, but are mostly complementary to each other. MrMathematica enhances Mathematica, and it helps to introduce Scheme to more people (CAS users).

1. Introduction

A Computer Algebra System(CAS) is a type of software package that is used in manipulation of mathematical formulae. The primary goal of a CAS is to automate tedious and sometimes difficult algebraic manipulation tasks. The principal difference between a CAS and a traditional calculator is the ability to deal with equations symbolically rather than numerically. The specific uses and capabilities of these systems vary greatly from one system to another, yet the purpose remains the same: manipulation of symbolic equations. CASes often include facilities for graphing equations and provide a programming language for the user to define his/her own procedures.

CASes began to appear in the early 1970s, and evolved out of research into artificial intelligence (in Lisp), though the fields are now regarded as largely separate. The first popular systems were Reduce, Derive, and Macsyma. The current market leaders are Maple and Mathematica; both are commonly used by research mathematicians, scientists, and engineers.

The programming languages provided in all the current mainstream CASes are not suitable for general purpose programming. To address this problem, I developed MrMathematica, a Scheme based system that keeps the repertoire of Mathematica.

The remainder of this article is organized as follows. Section 2 of this paper discusses why CAS programming language falls and why a real language is needed; Section 3 introduces Mathematica briefly; Section 4 gives details about MrMathematica; Section 5 concludes.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

2. CAS programmers need a real language

A key issue in the design of CAS is the resolution of what is meant by "evaluation" – of expressions and programs in the embedded programming language of the system.

Roughly speaking, evaluation is a mapping from an object (input) and a specified context or environment to another object that is a simpler or more specific object (output). Example: 2+3 evaluates to 5. More specifically and somewhat pedantically, in a CAS, evaluation involves the conventional programming language mapping of variables or names (e.g. x) to their bound values (e.g. 3), and also the mapping of operators (e.g. +) to their actions. Less conventionally, CAS evaluation generally requires resolution of situations in which a variable "has no value" but stands only for itself, or in which a variable has a value that is "an expression". For example, given a context where x is bound to 3, y has no binding or is used as a "free variable", and z is a+2, a typical CAS would evaluate x+y+z+1 to y+a+5.

In simple cases this model is intuitive for the user and efficiently implemented by a computer. But a system design must also handle cases that are not so simple or intuitive. CAS problem-solving sessions abound in cases where the name and its value(s) in some context(s) must coexist. Sometimes, values are not the only relevant attributes of a name: there may be a declaration of "type" or other auxiliary information. For example it might evaluate $\sin^2 x \leq 1$ to "True" knowing only that x is of type "Real".

CAS builders, either by tradition or specific intent, often impose two criteria on their systems intended for use by a "general" audience. Unfortunately, the two criteria tend to conflict.

1. The notation and semantics of the CAS should correspond closely to "common intuitive usage" in mathematics.

2. The notation and semantics of the CAS should be suitable for algorithmic programming as well as (several levels) of description of mathematical objects, ranging from the abstract to the relatively concrete data representations of a computer system.

The need for this first requirement (intuitiveness) is rarely argued. If programs are going to be helpful to human users in a mathematical context, they must use an appropriate common language. Unfortunately, a careful examination of common usage shows the semantics and notion of mathematics as commonly written is often ambiguous or context dependent. The lack of precision in such mathematics (or alternatively, the dependence of the semantics of mathematical notation on context) is far more prevalent than one might believe. While mathematics allegedly relies on rigor and formality, a formal "automaton" reading the mathematical literature would need to accumulate substantial context or else suffer greatly from the substantial abuse of notation that is, for the most part, totally accepted and even unnoticed by human readers. Consider $\cos(n+1)x \sin nx$.

Because the process of evaluation must make explicit the binding between notation and semantics, the design of the evaluation program must consider these issues centrally. Furthermore, evaluation typically is intertwined with "simplification" of results. Here

again, there is no entirely satisfactory resolution in the symbolic computation programs or literature as to what the "simplest" form of an expression means.

As for the second requirement, the need for programming and data description facilities follows from the simple fact that computer algebra systems are usually "open-ended". It is not possible to build-in a command to anticipate each and every user requirement. Therefore, except for a few simple (or very specific, application-oriented) systems, each CAS provides a language for the user to program algorithms and to convey more detailed specifications of operations of commands. This language must provide a bridge for a computer algebra system user to deal with the notations and semantics of programming as well as mathematics. Often this means including constructions which look like mathematics but have different meanings. For example, in Mathematica x = x+1 is programming language assignment statement; x == x + 1 is an apparently absurd assertion of equality. Furthermore, the programming language must make distinctions between forms of expressions when mathematicians normally do not make such istinctions. As an example, the language must deal with the apparently equal but not identical expressions 2x and x + x.

Programming languages also may have notations of "storage locations" that do not correspond simply to mathematical notations. Changing the meaning (or value) of an expression by a side effect is possible in most systems, and this is rather difficult to explain without recourse to notions like "indirection" and how data is stored. For example, in Mathematica, m[[1,1]]= b assigns value to a position in the matrix m.

With respect to its evaluation strategy, each existing CAS chooses its own twisting pathway, taking large and small sometimes controversial stands on different issues, along the way. Let's see an example in Mathematica:

```
i = 0;
g[x_] := x+i/;i++ > x
```

Or put in Scheme syntax:

```
(begin (Set i 0)
       (SetDelayed (g (Pattern x (Blank)))
                   (Condition (+ x i)
                              (> (Increment i) x))))
```

The two allegedly equivalent expressions (list (g 0) (g 0)) and (Table (g 0) (list 2)) result in (list (g 0) 2) and (list (g 0) (g 0)) respectively.

Other CASes sufferes from similar problems. [4] From the author's own experience, when writing big programs in Mathematica (or some other major CAS), such problems can and will arise, resulting in substantial debugging difficulty.

Providing a context for "all mathematics" without making that unambiguous underpinning explicit is a recipe that ultimately leads to dissatisfaction for sophisticated users.

Is there a way through the morass? A proposal (eloquently championed some time ago by David R. Barton at MIT and more recently at Berkeley) [4] goes something like this: Write in Lisp or similar suitable language and be done with it. This solves the second criterion. As for the first criterion of naturalness – let the mathematician/user learn the language, and make it explicit.

But there is nearly no CA library in Scheme, besides the lightweight JACAL. Statistics shows that for those people who want to do symbolic computation with a computer, nearly all are using a CAS, and nearly none is using Lisp, although most of them also want general purpose programming at the same time. What's worse, CASes that are in/with Lisp (such as MACSYMA, Axiom) have only negligible market share.

So I wrote MrMathematica, which lifts and embeds a popular CAS, Mathematica, into Scheme. Although the currently version targets only MzScheme, its design is portable to any Lisp implementation that can be extended using C. Mathematica was chosen because it has the most dynamic language among major CASes; PLT Scheme was chosen because it has a good interface and a large user group.

3. Introduction to Mathematica

In a typical CAS, an internal evaluation program (`eval` for short), plays a key role in controlling the behavior of the system. Even though `eval` may not be explicitly available for the user to call, it is implicitly involved in much that goes on. Typically, `eval` takes as input the representation of the user commands, program directives, and other "instructions" and combines them with the "state" of the system to provide a result, plus sometimes a change in the "state". Mathematica is one CAS that has a single `eval`.

The central data types of Mathematica are just the same as Scheme: numbers, symbols, and lists. The abstract syntax of the two languages is also congruent: every expression is a list-based tree. To accommodate traditional mathematical expression syntax, Mathematica defines several forms: `InputForm`, `OutputForm`, `TranditionalForm`, `FullForm`, and so on. The FullForm is very close to S-exp, and is the internal representation of expression. A FrontEnd is used to convert between ordinary mathematical expression (InputForm, OutputForm, TranditionalForm) and FullForm.

Mathematica has two major difference compared with Lisp. First, Mathematica doesn't have quote. Second, Mathematica uses array (of pointers) instead of Lisp's linked-list.

The underlying strategy for evaluation in Mathematica is based on the notion that when the user types in an expression, the system should keep applying rules (and function evaluation means rule application in Mathematica) until the expression stops changing. (The example in the previous section just violate this strategy!)

To get a detailed introduction of Mathematica language,please refer to part 2 of [2], or [6].

There are additional evaluation rules for numerical computation in which Accuracy and Precision are carried along with each number. These are intended to automatically keep track of numerical errors in computation.

Besides the rule-based language, Mathematica also offers many mathematical functions and methods, including algebraic manipulation, symbolic calculus, plotting, and so on. Part 3 of [2] describes them in detail.

4. Structure and Interpretation of MrMathematica

Scheme is a meta-language and MzScheme is actually an operation system [3], while Mathematica regards itself only as a scientific computation tool. This determines the architecture of MrMathematica: It works as an extension to MzScheme, which calls Mathematica.

Among all possible interface (between Scheme and Mathematica). I choose to implement the simpest one, MathEval, which is exactly the `eval` used by Mathematica. MathEval is provided as a Scheme function, with input and output done in S-exp, making use of the similarity between S-exp and FullForm. MathEval suffices. Even if you want some "better" interface, the right way to implement it is first to define the same MathEval, and then to define your interface based on it. Another merit of MathEval is that it needs explicit quote, which helps distinguighing between algebra expression and other Scheme value.

Mathematica and MzScheme are both implemented in C, so it is natural for MrMathematica to use C as transmitter. But the ma-

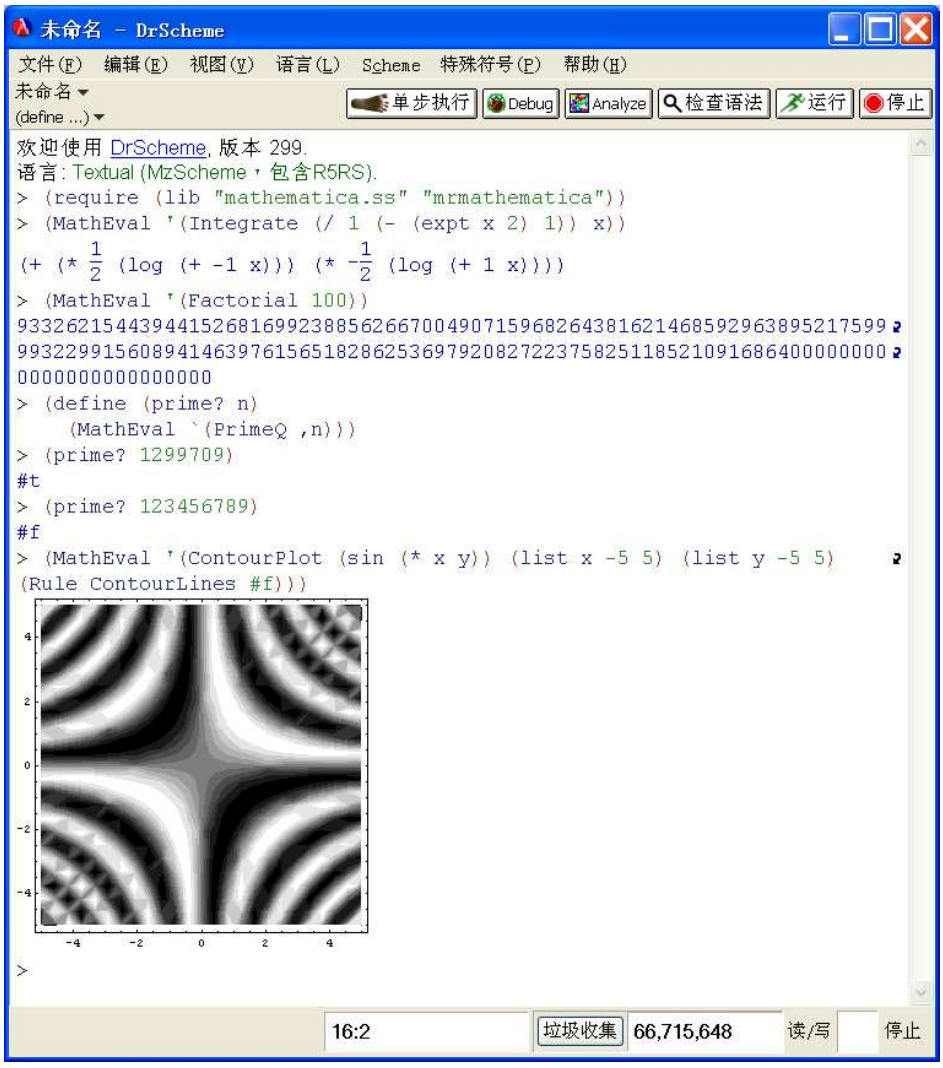


Figure 1. MrMathematica session

jor part of MrMathematica was written not in C but in Scheme. Bottom-up style was used: All needed MathLink (Mathematica's C interface) functions were raised into Scheme in a lower layer implemented as a Scheme module. All the other parts of MrMathematica are written in Scheme, and the final export is the Scheme function MathEval. Compared with the interface provided by MathLink, nearly all the details about the call are encapsulated.

Although the structures of S-exp and Mathematica-expression are similar, the actual keywords are different. The syntax of some pre-defined functions is also distinct. To bridge the gap, I use a separate module in MrMathematica to translate expressions. The result is that a user can write expression just as a Scheme one and send it to Mathematica. In most cases, the output of Mathematica can be directly feed into the Scheme function `eval` or used directly as a Scheme object. The default rules in the translate table are conservative, only dealing with the (exact) common part of Scheme and Mathematica. Programmers can customize the table by new rules.

From the example in Figure 1, we can see that MrMathematica allows every Mathematica Input-Output done in "FullForm" of Mathematica. So CAS users will lose no function from Mathematica, but get the unambiguous, aesthetically appealing, and consistent Scheme. The recommended way to use MrMathematica is, to do all the other programming job in Scheme, and when dealing with mathematical concepts, call the corresponding Mathematica function using MathEval.

You can define your Scheme function that use MathEval, thus using the power of Mathematica with almost no effort. For example, the Mathematica function FactorInteger was raised into Scheme, with exactly the same contract:

```
> (define (factorinteger n)
    (eval (MathEval `(FactorInteger ,n))))
> (factorinteger 111111111111111111)
((3 2) (7 1) (11 1) (13 1) (19 1) (37 1)
 (52579 1) (333667 1))
```

A more efficient version:

```
> (define-syntax factorinteger
    (syntax-rules ()
      ((_ n)
       (map cdr
            (cdr (MathEval `(FactorInteger ,n)))))))
```

For computaion that involves algebra symol(s), explicit quote is used. See the example about integration. To use the return value in Scheme, a explicit call to Scheme's eval is needed:

```
> (define f
    (MathEval
      '(Integrate (/ 1 (+ (expt x 2) 1)) x)))
> f
(atan x)
> (define s (eval `(lambda (x) ,f)))
> (s 0)
0
```

MrMathematica is designed to avoid providing too many features, but also to avoid weaknesses or restrictions. For example, calling multiple or remote Mathematica Kernel(s) is supported; parallel computation is available using PLT's thread utility; variables could all be put in Scheme and the quasi-quote will help transfer their values into Mathematica; Windows, Unix (including Linux), and MacOS are all supported; MrMathematica can render Graphics from Mathematica in DrScheme (this feature needs Scheme and Mathematica running on same machine, which needs further improvement).

Even if your favorite Scheme implementation is not PLT, porting MrMathematica should be easy. There are only three points that are not R5RS and SRFI: the Scheme to C interface, the module system, and the Graphics renderer. MrMathematica uses "Inside MzScheme", the only official C interface for PLT Scheme v20x, as its FFI. As mentioned before, all code that deals with C is in a separate module whose only role is raising C functions. Changing it into different FFI could be done as a routine. The same to module system. To render Graphics from Mathematica in DrScheme, MrEd is used. When using other Scheme implementation, you can easily disable this feature, just as the light-weight version of MrMathematica (designed for MzScheme only instead of full DrScheme) does.

5. Conclusion and Future Work

With MrMathematica, you can use whatever feature you like either from Scheme or from Mathematica. The recommended method to use MrMathematica is to do mathematical compuation in Mathematica and other programming in Scheme. This solves the problem of major CASes: the lack of a good programming language.

Schemers can view MrMathematica as a Computer Algebra library, or a build in term rewriting engine; Mathematica users can view it as a Foreign Language Interface better than that of Java, Perl or Python (string based). After all, the two languages are homologous, thus making the symbiosis.

However, this is only a start of the project. To be really successful, MrMathematica need more applications. Hence this paper. Enjoy hacking with MrMathematica!

For more information about MrMathematica, please visit `http://www.websamba.com/mrmathematica`.

Acknowledgments

Thanks to LinPeng Huang, Matthew Flatt, and Shriram Krishnamurthi for prereading the draft of this paper.

References

[1] PLT Scheme. `http://www.plt-scheme.org/`.

[2] Stephen Wolfram. The Mathematica Book. Wolfram Media, 5th Edition, 2003.

[3] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming Languages as Operating Systems. ICFP 1999.

[4] Richard J. Fateman. Symbolic Mathematics System Evaluators. ISSAC 1996.

[5] Geddes K.O., Czapor Stephen R., and Labahn George. Algorithms for Computer Algebra.Kluwer Academic, 1992.

[6] John Gray. Mastering Mathematica. Academic Press,Inc, 1994.

[7] G J Chaitin. Algorithmic Information Theory. Cambridge University Press, 2004.

[8] Stephen Wolfram. A New Kind of Science. Wolfram Media, 2002.

[9] Olin Shivers. A Scheme Shell. `http://www.scsh.net/docu/scsh-paper/scsh-paper.html`

[10] Aubrey Jaffer. Jacal. `http://swissnet.ai.mit.edu/jaffer/JACAL.html`

[11] Maxima. `http://maxima.sourceforge.net/`

[12] Axiom. `http://savannah.nongnu.org/projects/axiom`

[13] Reduce. `http://www.reduce-algebra.com/`

[14] Mapple. `http://www.maplesoft.com/`

ACT Parameterization Framework

Alan Pavičić
AVL-AST Zagreb, Croatia
alan.pavicic@avl.com

Nikša Bosnić
AVL-AST Zagreb, Croatia
niksa.bosnic@avl.com

Abstract

ACT is a generic parameterization framework used in the development of applications for modeling and parameterization of internal combustion engines. It is developed in Guile. Its two main parts are *Ilm* core of object model built on top of Goops, and *Bee* editor environment providing UI. The core object model supports generic persistence of any object to database, type guardians for different slots, nameservices and object repositories. It also supports *addins*, additional modules which can change the behavior of the entire system as well as any of its parts (e.g. undo/redo functionality, dependencies between objects, event notification, ...). The editor environment for editing Ilm objects includes a library of basic editors, simple composite editors and generic editors. A grading system can be used to dynamically decide which registered editor class is the most appropriate for editing a particular object. Every Bee editor is an Ilm object itself. High level XML descriptions of data models and editors can be compiled to Scheme code defining Ilm classes and Bee editors.

Keywords Lisp, Scheme, MOP, data model, UI, parameterization

1. Introduction

We are working for AVL, a company producing software that simulates parts of internal combustion engines. Most products in our product line are structurally similar. They all consist of two main parts – a part which models and parameterizes some aspects of an engine and a part which actually calculates simulations (solver). Each solver is typically monolithic stand-alone process which reads custom formatted data files from input stream, and after (sometimes very lengthy) calculation stores the result of the simulation to some output stream to be additionally post-processed.

We will concentrate on the part which allows user to model parts of an engine and prepares the input data for solvers in the system.

Such a modeling and parameterizing subsystem needs to be able to define and edit some particular aspects of an engine (depending on the actual ability of the particular solver) and then run the solver. Previously, such a subsystem was implemented in such a way that particular solvers were run from different programs written in C++ which weren't mutually connected. This architecture was drastically slowing down adding new or changing existing aspects of the engine and every change in UI required programmer intervention and rebuilding of the whole application.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

Obviously, we needed more expressive and more efficient system. The first step in the implementation was the analysis of requirements.

In most cases, parameterization is not a very difficult task, because it can be reduced to a relatively small number of statically defined classes of objects which are being parameterized. Connections between such objects are typically trivial, or there are no connections at all. Similarly, editors for such objects can be hand written or just partially automatized.

Sometimes requirements on parameterization can be quite serious. In our case, we have a project where a large number of classes is in play, which are intensively changed during development of program or can be added to system after it has already been deployed.

Also, we have some non typical requirements on objects as they have to know how to persist and depersist themselves (save state to some unspecified medium, such as a file, an internet connection or a relational database, and be able to restore it later, e.g. after the program has been restarted).

Models described by our system can be quite complex themselves and dependencies between objects can be very specific (e.g. relations between mechanic parts of car engine).

Motivated by all of the above, we decided to create a modeling language which allows the same functionality to be added in different ways, depending on the estimation of application developer, rather than to create a rigid tool which should anticipate all possible requirements on classes and relations between them.

Apart from modeling requirements, there are also requirements for additional changes to the functionality of objects. Again, instead of anticipating all possible ways how the behavior of object could be changed, we rather open a way to change the behavior of any class or object during runtime. As we will see later, abstractions for changing metaproperties of an object we will call *addins*.

Similarly, the system has to be able to describe even particular editors for particular types of objects or any other elements of UI. The philosophy should be that the simple editors could be generated automatically and very quickly, but if the application programmer wants to add a very specialized editor for some class or family of classes, that should be possible too. Such approach would guarantee us both – fast development when possible, and tuning anything within the system when necessary.

Finally, it would be nice if even the application itself could be described as a regular object which behaves like the rest of the system.

Such a system, which couples all mentioned elements, would be a parameterization framework for rapid application development in any technical area, not necessarily just engine simulation.

The system described is specific enough that the object model of any typically used OO language (C++, Java, Python, ...) doesn't fit completely. Moreover, since we have requirements that classes can change their behavior (e.g. an object is able to log all changes

of its properties) no fixed object model would serve us completely, no matter how powerful it is.

Creation of such object model from scratch would be a long and expensive task.

Thus, we decided to use the meta object protocol (MOP)[9] which allows us to be independent from any predefined fixed object model and gives us freedom to change the object model on the fly as needed.

The most complete implementations of MOP can be found in Lisp systems, so Lisp was the most obvious choice from the beginning. Because, from a management perspective, the experiment of using Lisp could have failed, Lisp implementation had to be free to reduce possible losses. Because of high number of target platforms, implementation also had to be easy to port. An additional requirement on Lisp implementation was that the chosen implementation has to interface easily to C because of third party libraries we use (GTK+, expat, OpenGL, libuuid, ...).

We chose Guile as the Scheme implementation because it satisfies most of our needs. It is widely used, free and easily portable. It comes with Goops [3] – a complete CLOS-like implementation of MOP. Although it meets all of our requirements, decision to use it is still questionable[1] and implementation of the whole system shouldn't involve anything Guile specific on the conceptual level so everything should be easily portable to any Lisp which has a complete implementation of MOP.

Goops itself has some differences from CLOS, but it is still part of CLOS family. It has slots, generic functions, methods, metaclasses similar to CLOS but it lacks proper implementation of method combinations.

The object system we built upon Goops is named *Ilm*, and the editor system built upon Ilm is named *Bee*. *ACT* is the complete architecture for application development, which along Ilm and Bee contains *Xi* – an XML editor which allows application developers to simply draw definitions and layouts of classes and editors, *Xic* – a compiler from Xi XML formats to Ilm and Bee definitions, and some parts more specific to area of internal combustion engine simulation. Xi is created for the sake of more efficient application development and the fact that most of our application developers do not know Scheme.

2. Ilm

2.1 Ilm Basics

The basic idea of Ilm is to enrich Goops with new features, but to preserve the way the object system is used. That means there should be no difference between using Ilm classes and using classes which are instances of the default metaclass `<class>`.

From user's point of view the basic difference is that a class is defined using `define-ilm-class macro`, which is syntactically the same as `define-class` macro. The class defined in such a way has metaclass `<class-ilm>` and has class `<unique>` added to its list of superclasses. An additional difference is that slots, which do not have getter and setter names defined, will get standardized names for them (prefixing "get-" or "set-" and adding "!" at the end of setter name). We must enforce that access happens only through getters and setters because for some elements of the system to work, one may use only get/set functions to communicate with instances and should never work directly with `slot-ref` and `slot-set!` functions. Similarly, `#:init-keyword` is added if absent.

For example, the code:

```
(define-ilm-class <gas> ()
  specific-heat-capacity
  specific-heat-ratio
  dynamic-viscosity)
```

creates an Ilm class `<gas>` with fully defined slots.

Above definition is expanded to:

```
(define-class <gas> (<unique>)
  (specific-heat-capacity
  #:init-keyword #:specific-heat-capacity
  #:setter set-specific-heat-capacity!
  #:getter get-specific-heat-capacity)
  (specific-heat-ratio
  #:init-keyword #:specific-heat-ratio
  #:setter set-specific-heat-ratio!
  #:getter get-specific-heat-ratio)
  (dynamic-viscosity
  #:init-keyword #:dynamic-viscosity
  #:setter set-dynamic-viscosity!
  #:getter get-dynamic-viscosity)
  #:metaclass <class-ilm>)
```

The class `<unique>` has a single slot `uuid` which is set to unique 128 bit value during instance initialization. To generate that value, the libuuid library is used.

The second class essential for the system is class `<ref>` used for representing references. It is a simple Goops class which contains two slots – the slot `uuid` which keeps the uuid of the object the reference points to, and the slot `obj` which keeps the object itself. The value of the slot `obj` is `#f` if the target object is not loaded.

One of the basic requirements on the system is that every object must be persistable. Knowing that an object in its slot may contain any Scheme value including other objects or collections of objects, it is easy to imagine a situation where we have cycles in the reference graph (in fact this situation is very common when the model is complex).

Class `<ref>` is used for breaking the circularity during recursive persistence of objects. When another Ilm object is found during traversal through object's slots or compound values within a slot, we are persisting a reference to that other object using its uuid as the key rather than the found object itself.

When an object is instantiated or depersisted (loaded) it registers itself with the *object repository*. The object repository is a weak hash table whose keys are uuids of objects, and values are objects themselves. During depersistence, the system again recursively traverses through all object's slots and values. When a reference to an object is found, the system looks for matching a object in the repository and puts it to the proper place. If a matching object is not found (it is not depersisted yet), the system adds a broken link to the hash table and stores a location which should point to the missing object. Eventually, when the missing object is loaded all missing links are removed from the hash table and all pointers are set to their proper values. Such loading strategy enables lazy loading of instances, which is an advantage when large clusters of objects that do not have to reside in memory simultaneously need to be loaded. Obviously, the object repository has to be a weak hash table because if an object is not referenced by some other object (other than the repository itself), it should be collected.

2.2 Persistence

The serialization format of the persisted object does not depend on the database implementation and is always the same. That property allows easy implementation of persistence to a new medium.

Objects are always stored as S-expressions.

The basic writer for objects is the standard generic function `write`[2], with specialized methods for a few additional classes. The main change, with respect to standard write, is that `<ref>` and

[1] performance problems, bugs, module system deficiencies

[2] R^{5}RS [8] the `write` procedure becomes generic function after Goops is loaded

<unique> are written in custom syntax #,(instance ...) that stores the class name of the persisted instance and the keyword list of #:init-keyword value pairs. This syntax is the reason why every slot that needs to be persisted has to have #:init-keyword defined, and why Ilm will add one if omitted. Every database implementation has to provide a port used by write for storing the object.

Analogously, loading of object is implementation independent. Using define-reader-ctor from SRFI-10[6], #,(instance ...) syntax allows us to use the standard function read for reading from given port. If the class whose instance is being read is not yet present in memory, the system will look for its definition on the file system and load it before instantiating the object.

For example, a persisted instance of the above class <gas> could look like:

```
#,(instance <gas>
   #:uuid
   #,(uuid "c6e93456-fef8-44df-9738-d00df8926860")
   #:specific-heat-capacity
   #,(instance <ref>
      #:uuid
      #,(uuid "8426e7f7-1883-48a5-ab4b-43dcf94ba45d"))
   #:specific-heat-ratio
   #,(instance <ref>
      #:uuid
      #,(uuid "75cd206c-d03f-4288-ae1d-109a0e5360bd"))
   #:dynamic-viscosity
   #,(instance <ref>
      #:uuid
      #,(uuid "ac11af8e-0913-4a90-b16b-53b0e7903864")))
```

By default each bound slot with allocation type #:instance and which has #:init-keyword will be persisted. If we do not want to persist such slot, we can use keyword #:nopersist while defining the slot. If the value of the #:nopersist keyword is true, the slot is skipped.

The storage (database) where objects are persisted is named *object pool*, regardless of how it is implemented.

A valid implementation of an object pool is any library that satisfies the following requirements:

- it must invoke the standard read and write on its own ports while loading and saving an object
- it must support shallow loading and saving (i.e. implement load-object and write-object) using standard read and write
- it must support deep loading and saving (i.e. implement load-object-deep and write-object-deep)

Such definition of the object pool provides transparent scalability from trivial object pools (e.g. persistence to the clipboard used for copy/paste) to large databases.

It is recommended that object pool implementation indexes objects by uuid, while other indices are not required[3].

Most object pool implementations will have a symbolic name for their identification.

At the moment, three different object pool implementations exist:

- using the file system – The database name is the directory name, every object is in its own file named after object's uuid. Indexing is done by the file system.
- single file database – Used for embedding Ilm databases into other formats. The database name is the file name and the index is embedded in the file.
- Berkeley DB – Currently in the test phase. A hash table is used for indexing uuids.

Regardless of implementation, an object pool should be garbage collected periodically; otherwise dead objects can remain in it forever. The root set for the object pool garbage collector is the *name service*.

Object pools describe physical representation of the stored object. If we want to arrange objects in logical an hierarchy or we want to give a logical name to an object, we use <name-service>. <name-service> can be considered as analog to file system in Ilm world. A standard way for an application to get some particular object by its name from the the object pool, is using name service (using uuid is considered bad style since uuids should only be used internally and there is no guarantee the object will remain in the database if it is reachable only by uuid).

<name-service> is a standard Ilm class. Therefore, it can be persisted. Since it keeps references to other Ilm objects, placing another named instance of <name-service> within it creates a lower level in hierarchy in the logical sense. The root name service always has to exist and every object pool has to have a function for obtaining it. Typically, that function is named load-obj-from-named-source. If an object is not reachable from the root name service or some other named source it may be considered dead. <name-service> is a simple hash table.

2.3 Metaclasses, Aspects and their Applications

Every Ilm class is an instance of the metaclass <class-ilm>. <class-ilm> is derived from <class>. The initial reason for introducing additional metaclass to the base system is possibility to customize the initialize method for <class-ilm>, which allows us to change the behavior of the class we are defining before it is fully defined.

For example, Goops creates all getters and setters of a class as instances of <accessor-method> class. If we want to combine methods generated with getters and setters with some other methods, instances of <accessor-method> class are not sufficient because they do not support next-method (the form needed to combine methods). That is the reason why we replace all <accessor-method> instances that we get from slots with regular instances of <method> during instantiation of <class-ilm>. The implementation of newly created methods is taken from accesor methods.

The fact that getters and setters are regular methods is intensively used by addins.

We used the ability to modify a class during its creation to introduce several new keywords in slot definitions. #:getter-thunk and #:setter-thunk define post and pre processing procedures respectively which are used to modify default implementations of getters and setters. #:getter-thunk takes two arguments: the object whose getter is invoked and the value received from the default getter. #:getter-thunk that simply returns the value received from default implementation would be implemented as:

(lambda (obj val) val).

#:setter-thunk takes three arguments - the object, the new value and the procedure which would be invoked by default. A simple pass-through #:setter-thunk would be:

(lambda (obj val proc) (proc obj val)).

Slot definitions may omit thunks. One example when thunks should be used is automatic conversion of units (model internally uses SI units while values are provided in arbitrary unit system selected by user). In this case thunks would perform the unit conversion.

The last keyword we added for customizing slot definitions is #:type. It is used as type guardian for particular slot – if one tries

[3] The implementation of a query language is planned.

to assign to a slot a value of wrong type, a runtime exception is raised.

If a slot has both, `#:setter-thunk` and `#:type` keywords, the new value is first passed through the setter and than processed by the type checker. Types can be basic types like integers or strings, enumerated types (elements of a symbol list), other Ilm classes or compound types like type list and type union. A type union allows slot values of one of the specified types for that slot. A type list requires the value to be a list of instances of specified types for that slot, properly ordered. With such compound types, any recursive type can be described.

Example for canonical definition of list of integers without macro usage:

```
(define int-list (make <ilm:type-union>))
(set-types! int-list
            (list (make <ilm:nil>)
                  (make <ilm:type-list>
                        #:types (list
                                  (make <ilm:integer>)
                                  int-list))))
```

and redefinition of the class `<gas>` with a new type guarded slot `ints`:

```
(define-ilm-class <gas> ()
  ...
  (ints #:type int-list))
```

Types are used for better guarantee of correctness of program as well as to enhance introspection capabilities (used by generic Bee editors).

Like the ability to define classes separately from methods, it would be nice if parts of the same class could be defined separately. In practice, it is often a case that some property or a set of properties is defined later on, and that it is added to definitions of some already defined classes. For example, an engineer who describes a cylinder cares only about slots which are related to calculations in some particular simulation, but the class `<cylinder>` can have some additional properties not necessarily related to engine simulations (e.g. the name of the author and some documentation). Such sets of orthogonal properties of a class we call *aspects*. When an aspect is added to a class, new slots are introduced, but the class doesn't change its behavior in any other way. Every slot in the class stores which aspect introduced it. If a slot is supported by several different aspects, it contains a list of all those aspects. If different aspects introduce the same slot with incompatible settings (e.g. `#:init-value` is different), the system raises an error.

Information which aspects are supported by the class are stored in a slot of the metaclass `<class-ilm>`. The class and its slots can be queried and filtered by different aspects. The macro `define-class-aspect` is syntactically similar to the macro `define-ilm-class`, except that it takes the name of the aspect as its second argument. The implementation of aspects is basically redefinition of a class in a way that all already existing slots are kept and new slots are added, taking care about merging of properties of duplicate slots[4].

An example of a macro for adding an aspect to a class:

```
(define-syntax add-name-aspect
  (syntax-rules ()
    ((_ cls) (define-ilm-class-aspect cls #:naming
               (name #:init-values ""
                     #:type (make <ilm:string>))))))
```

and usage of that macro applied to the class `<gas>`:

```
(add-name-aspect <gas>)
```

[4] We are considering implementation of aspects using multiple inheritance that would enable specialization of methods by aspects.

Now `<gas>` has a new slot `name` and supports the naming aspect.

2.4 Addins

The basic behavior of objects (e.g. persistence) is always in the system. When we want to introduce some additional behavior of an object which for some reason (memory usage, speed, pure aesthetics, ...) doesn't need to exist for every class or object, we are introducing special types of modules, named *addins*. An addin can introduce a new behavior which cannot be described by the base system itself.

While aspects introduce new slots and don't change the behavior of the class, addins bring new functionality to existing methods.

Such enriching of the model with new a functionality we call injecting. Important features of addins are that they can be applied to any class or instance and that they can be combined. Number of additional addins which can be added to the base system is unlimited.

We will try to clarify addins through two examples – undo/redo and dependency addin.

A system which would keep track of all changes on all slots of every object all the time would at times be needlessly inefficient (e.g. when it is used by some calculation which is executed from a script where things like undo and redo make no sense). On the other hand, the ability to execute undo and redo actions on some objects and keeping track of all changes chronologically is quite helpful to application developers, who could use the object system without knowing how to implement undoing. Undo addin addresses exactly that issue. Even in an application that needs undo/redo functionality not all objects are undoable. All an application developer has to do to have undo/redo facility in his program is to declare which objects should be undoable or declare classes whose all instances should support that facility.

Injecting an addin means that a new class will appear in the system. The new class will be composed of two – the original class and a class which is introduced by the addin. Composition is done using multiple inheritance. The class introduced by the addin is typically an instance of some addin-specific metaclass, so the composed class will be an instance of the addin's metaclass too. Hence we can additionally customize the composed class in the `initialize` method of the addin's metaclass. In the undo/redo example, we are traversing through all setters, modifying them to register all changes on the global undo/redo stack and to invoke `next-method` which in turn invokes the original setter method, specialized for old class to which addin was injected. That is the reason why we had to convert all getters and setters from `<accessor-method>` to `<method>`. `undo` and `redo` functions are just executing closures stored on a global stack. Of course, changes are captured only when an object is changed through a setter and the object is an instance of an Ilm class with undo/redo addin injected.

If an application programmer knows in advance which addins should be used, and into which classes or objects they should be injected, he could use the composed class name – the name of class concatenated to the name of the injected addin. If we want to make an undoable instance of `<gas>`, we would create an instance of the class `<<undo><gas>>`, where `<undo>` is a the name of addin class.

If we want to inject an addin to an already instantiated object, after its class is composed with the addin, all we have to do is call `change-class` to the newly created class. Since the new class has superset of slots of the old one, all values within old slots will remain untouched. Instead of invoking old methods, such object will have more specialized methods for setters, which are created during composition of classes.

The purpose of the dependency addin is that slot values can be calculated from values of other slots (perhaps from another ob-

ject) by some user defined formula. If we make one slot dependent of other slots the connection will be stored in an instance of class `<dependency-descriptor>`, which also stores the dependency formula. Propagation of change is eager – immediately after some value is changed, the object knows whether it is dirty (needs updateing), but the calculation of the value is lazy and it calculates only parts it actually needs. The persistence of such cluster of objects will not calculate all dirty objects before they are stored. Rather, it will persist current in-memory state including `<dependency-descriptor>` objects. Same as in undo addin, everything what's happening during setting of the slot value and during reading from a slot is defined in `initialize` method of the dependency metaclass `<dep-mc>`. Original getters and setters are invoked using `next-method`.

When an object with an injected addin is loaded, the name of its class is recognized as composed class and after loading of class and addin, additional composition is performed.

For example:

```
(inject-addin-to-class <undo> <gas>)
```

will create a new class `<<undo><gas>>` whose instance will be persisted as:

```
#,(instance <<undo><gas>>
   #:uuid
   #,(uuid "c6e93456-fef8-44df-9738-d00df8926860")
   #:specific-heat-capacity
   #,(instance <ref>
      #:uuid
      #,(uuid "8426e7f7-1883-48a5-ab4b-43dcf94ba45d"))
   #:specific-heat-ratio
   #,(instance <ref>
      #:uuid
      #,(uuid "75cd206c-d03f-4288-ae1d-109a0e5360bd"))
   #:dynamic-viscosity
   #,(instance <ref>
      #:uuid
      #,(uuid "ac11af8e-0913-4a90-b16b-53b0e7903864"))
   #:ints (1 2 3)
   #:name "air")
```

To ejecte an addin can from an object, `change-class` to the original class can be invoked.

The list of possible addins is open ended. In addition to already described addins we implemented an event addin which makes an object notify its listeners when any of its slots change. Implementations of a locking addin which would replace proper setters with dummy setters and debug addin which is able to log all changes within the system are planned.

3. Bee

3.1 Bee Basics

For a complete solution of the parameterization problem, apart from the data model we also need *editors* – UI components dedicated to the interactive modification of objects. The part of our system addressing the task of editing Ilm objects is called *Bee*[5]. Altough Bee does not limit the choice of UI library, all currently created editors are implemented using GTK+ [4].

Every Bee editor is an Ilm object itself. That approach elegantly solves persistence of editors (i.e. UI state), dependencies between editors etc. Additionaly, it also enables creation of meta-editors (Bee editors designed for creation and modification of Bee editors)[6]. Furthermore, since Ilm permits modeling of the data meta-model, Bee editors are also used for editing the data model itself i.e. Ilm class definitions.

An additional source of flexibility of Bee editors is a classical Lisp pattern where an editor accepts a procedure (commonly just a simple lambda expression) as a value of a parameter that specifies or specializes its behavior. Examples of problems solved in such a way are definition of arbitrary hierarchy in the generic tree editor (children of a tree node are returned by a procedure given as a parameter, effectively solving filtering and ordering also) and naming of an entity (name is generated depending on the context and/or translated to the given language).

Since an editor is fully defined just by defining six state-changing actions upon it, a short description of the life cycle of an editor (shown in figure 1) is necessary.

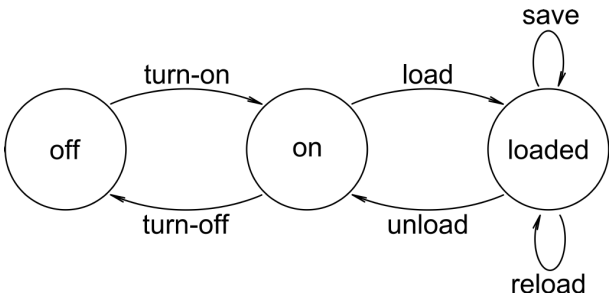


Figure 1. Editor state diagram

- The state *off* is the starting and the ending state. An editor in that state exists as an Ilm object but it still (or again) doesn't have any UI representation. This state is introduced to enable manipulation of properties of the editor which must be defined before the widget (or widget hierarchy) that makes up the editor's UI is created.
- In the state *on*, the static part of the editor's UI representation is created but it is not visible. The static part of UI representation is the part that can be created without knowing exactly which object will be edited and it includes at least the main widget of the editor. We can add an editor in this state to some parent widget and by doing so we can build UI to be shown later all at once.
- The state *loaded* is the "working" state of the editor. Before the editor can enter this state, the object to be edited must be set. UI representation exists in full and is visible, and the editor permits interactive modification of the object.

Actions *turn-on*, *turn-off*, *load* and *unload* switch states of an editor. All transitions shown on the state diagram are allowed (e.g. off-on-loaded-on-loaded-on-off) so the same editor can be used multiple times for editing (even editing different objects) without repeated construction and destruction of the static part of its UI representation.

Each action is implemented as a Goops method that can be invoked by the owner of the editor. Bee provides a simple embedded language[7] for defining editors:

- `specialize-ed-class` macro defines an editor class (using `define-ilm-class`) and overrides the default initial values for slots inherited from its base classes.
- Macros `define-turn-on`, `define-turn-off`, etc. simplify the definition of appropriate methods, provide error checking and ensure state consistency.

[5] short for "**B**ee is an **e**ditor **e**nvironment"

[6] This possibility is not employed in its full strength in the current implementation.

3.2 Basic Editors and Simple Composite Editors

Basic editors cover editing of "atomic" objects and serve as building blocks for construction of complex editors. Typical examples of basic editors are editors for strings, numbers, enumerated values, Boolean values, tabular functions, physical quantities (a pair of a number and a unit from given unit group) etc. Although each basic editor must be manually coded[7], the embedded language described above greatly reduces the effort.

For example, the complete definition of an editor for real numbers is:

```
(specialize-ed-class <real-ed> (<gtk-ed>)
  (layout-hints '(#:hflexible #:small)))

(define-turn-on (ed <real-ed>)
  (set-widget! ed (make <gtk-entry>)))

(define-turn-off (ed <real-ed>))

(define-load (ed <real-ed>)
  (gtk-widget-set-sensitive (get-entry ed)
                            (not (read-only? ed)))
  (load-text ed))

(define-unload (ed <real-ed>)
  (gtk-entry-set-text (get-entry ed) ""))

(define-save (ed <real-ed>)
  (unless (read-only? ed)
    (set-obj! ed
              (string->real (gtk-entry-get-text
                             (get-entry ed))))))

(define-reload (ed <real-ed>)
  (load-text ed))

(define (load-text ed)
  (gtk-entry-set-text (get-entry ed)
                      (real->string (get-obj ed))))

(define get-entry get-widget)
```

Simple composite editors group several (often basic) editors into one whole. Layout creation algorithms have access to *layout hints*, a way for a child editor to express its properties regarding layout. While the current version of Bee includes only a simple single-column composite editor, a table composite editor is under development.

3.3 Grading

To any editor class we can attach one or more *graders* – procedures that, based on properties of the location we want to edit (e.g. allowed types of objects, type of the object currently stored at the location, read-only flag, ...), give a numerical measure of how appropriate an instance of the editor class would be for editing that location. That way we can make the decision about the most appropriate editor class dynamically, without the explicit knowledge about all editor classes in the system and their requirements.

One appropriate grader for the real number editor class could be registered as:

```
(registry-add-type-grader
 (lambda (type)
   (and (or (is-a? type <ilm:real>) (eq? type <real>))
        (cons <real-ed> 11))))
```

A later call to a query function such as `registry-grade-type` would include a pair of the editor class `<real-ed>` and the grade 11 in the returned list if the type given satifies the above condition.

[7] as opposed to automatically generated

By convention, more specific editors are given higher grades. A non-specific "last-resort" editor intended primarily for use by application developers can be used for any location but gets a low grade. On the other hand, an editor created for a specific narrow category of objects (like `<gas-ed>` below) gets much higher grade, but under more selective conditions.

3.4 Generic Editors

The concept of graders opens a door towards *generic editors*. Generic editors use simple composite editors as containers and layout managers for editors created according to the results of the grading of parts[8] of the object being edited. The simplicity of this process enhances scalability with respect to the number of classes in the data model and the number of editor classes in the system, along with resilience regarding data model changes. Furthermore, generic editors enable work on the data as soon as the data model is finished or even during its development.

For example, figure 2 depicts an instance of `<uni-ed>`, a generic editor that uses the described grading and the single-column composite editor, editing an instance of the class `<gas>` defined with the appropriate slot type information. `<uni-ed>` grades each slot of the given object, selects the editor class with the highest grade if any, and adds an instance of the selected editor class to a single-column composite editor serving as a container and layout manager.

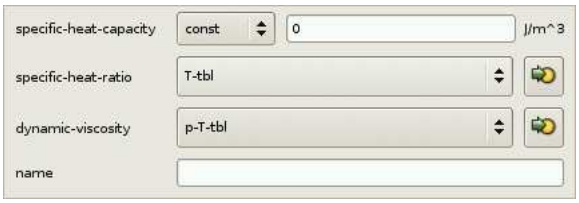


Figure 2. A generic editor

With some minimal specialization generic editors can often replace complex editors built manually by gradual composition of basic editors:

```
(specialize-ed-class <specific-heat-capacity-ed>
  (<multi-type-ed>)
  (slot-namer (make-alist-namer
               '((const . "Constant")
                 (T-tbl . "Table (T)")
                 (p-T-tbl . "Table (p,T)")))))
(registry-add-class-type-grader
  <specific-heat-capacity>
  <specific-heat-capacity-ed> 13)

;;;; omitting similar specialization code for specific
;;;; heat ratio and dynamic viscosity editors

(specialize-ed-class <gas-ed> (<uni-ed>)
  (heading "Gas")
  (slot-namer
   (make-alist-namer
    '((name . "Name")
      (specific-heat-capacity ."Specific Heat Capacity")
      (specific-heat-ratio . "Specific Heat Ratio")
      (dynamic-viscosity . "Dynamic Viscosity")))))
(registry-add-class-type-grader <gas> <gas-ed> 13)
```

The specialized generic editor `<gas-ed>` is shown in figure 3.

[8] typically non-virtual instance slots

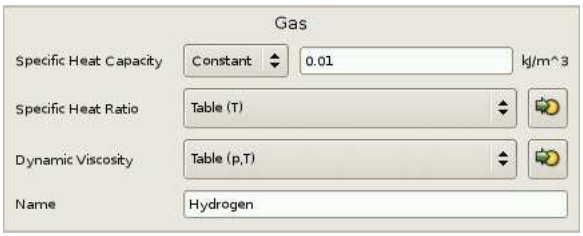


Figure 3. A specialized generic editor

4. Related Work

ASL's [10] components Adam and Eve2 treat problems that are similar to those treated by Ilm and Bee, respectively. We avoid the multiple language approach (C++ for the implementation of libraries, AEL for data and dependency expressions, AVM for the interpretive runtime execution) by using Scheme for all parameterization purposes.

Cells [1] and Cells-Gtk [2] combined also provide a flexible Lisp based parameterization framework.

5. Conclusion

The initial predevelopment experiment was successful and after a short additional development phase we are about to start with deployment, proving once again that Lisp and MOP should be considered in commercial programming at least equally to C++ or Java. Problems we encountered using Lisp weren't of conceptual nature and mostly were related to the selection of a good implementation that covers our specific requirements.

References

[1] *Cells*. http://common-lisp.net/project/cells

[2] *Cells-Gtk*. http://common-lisp.net/project/cells-gtk

[3] *Goops*. http://www.gnu.org/software/guile/docs/goops

[4] *GTK+*. http://www.gtk.org

[5] *Guile*. http://www.gnu.org/software/guile

[6] *SRFI-10*. http://srfi.schemers.org/srfi-10/srfi-10.html

[7] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[8] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). *Revised(5) Report on the Algorithmic Language Scheme*. http://www.schemers.org/Documents/Standards/R5RS

[9] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[10] Sean Parent, Foster Brereton. *Overview of Adobe Source Libraries*. http://opensource.adobe.com/group__asl__overview.html

Javascript to Scheme Compilation

Florian Loitsch

Inria Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex
France

Florian.Loitsch@sophia.inria.fr

ABSTRACT

This paper presents Jsigloo, a Bigloo frontend compiling Javascript to Scheme. Javascript and Scheme share many features: both are dynamically typed, they feature closures and allow for functions as first class citizens. Despite their similarities it is not always easy to map Javascript constructs to efficient Scheme code, and in this paper we discuss the non-obvious transformations that needed special attention.

Even though optimizations were supposed to be done by Bigloo the chosen Javascript-Scheme mapping made several analyses ineffective and some optimizations are hence implemented in Jsigloo. We illustrate the opportunities Bigloo missed and show how the additional optimizations improve the situation.

1. Introduction

Javascript is one of the most popular scripting languages available today. It was introduced with Netscape Navigator 2.0 in 1995, and has since been implemented in every other dominant web-browser. As of today nearly every computer is able to execute Ecmascript (Javascript's official name since its standardization [9] in 1997), and most sophisticated web-sites use Javascript.

Over the time Javascript has been included in and adapted to many different projects (eg. Qt Script for Applications, Macromedia's Actionscript), and it is not exclusively used for webpages anymore. Most of them are interpreting Javascript, but some are already compiling Javascript directly to JVM byte code (eg. Mozilla's Rhino [5] and Caucho Resin [4]).

Javascript is not easy to compile though. Several of its properties make it challenging to generate efficient code:

- Javascript is dynamically typed,
- functions are first class citizens,
- variables can be captured by functions (closures),
- it provides automatic memory management, and
- it contains an *eval* function, which allows one to compile and run code at run-time.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

Scheme has similar features, and Scheme compilers are faced with the same problems. Contrary to Javascript much research has been spent in compiling Scheme, and there exists several efficient Scheme compilers now. By compiling Javascript to Scheme it should hence be possible to benefit from the already present optimizations. Bigloo, one of these efficient compilers, has the supplementary advantage of compiling to different targets: in addition to C, it is capable of producing JVM bytecode or .NET's CLI. A Javascript to Scheme compiler would hence immediately make it possible to run (and interface) Javascript with these three platforms.

When we started the compiler we expected to have the following advantages over other Javascript compilers:

- The compiler should be small. Most of Javascript's features exist already in Scheme, and only few adjustments are needed.
- The compiler should be easy to maintain. A small compiler is easier to maintain than a big, complex compiler.
- The compiler should be fast. Bigloo is fast, and if the translated code can be optimized by Bigloo, the combined compiler should produce fast code. An efficient Javascript to Scheme compiler does not need to create efficient Scheme-code, but code that is easily optimized by Bigloo.
- Any improvement in Bigloo automatically improves the Javascript compiler. New optimizations are automatically applied to the Javascript code, and new backends allow distribution to different platforms.
- Javascript code could be easily interfaced with Scheme and all languages with which Bigloo interfaces.

Many existing Javascript compilers or interpreters already featured some of the listed points, but none combined all these advantages.

Our compiler, *Jsigloo*, takes Javascript code as input, and translates it to Scheme code with Bigloo extensions [1] which is then optimized and compiled to one of the three platforms. Furthermore it is planned to integrate Jsigloo into Bigloo (as has been done for Camloo [16]) thereby eliminating the intermediate Scheme-file.

Section 2 will detail the differences between Javascript and Scheme. In Section 3, the chosen data-structure mapping and typing issues are discussed. Section 4 describes the code generation and how encountered difficulties are handled. Some preliminary performance results are given in Section 5. Section 6 shows why Jsigloo is not yet finished and what needs to be improved in the future. Finally, Section 7 concludes this paper.

[1] Most of the used extensions increase Bigloo's efficiency and could be either omitted or replaced by equivalent (slower) Scheme expressions.

2. Javascript vs. Scheme

Javascript and Scheme share many features, and this section will therefore concentrate on their differences rather than similarities. Even though Javascript is generally considered to be an object oriented language, it bears more resemblance to functional languages like Scheme than to most object oriented languages. In fact Javascript's object system is based on closures which is a feature typically seen in functional languages.

Javascript's syntax resembles Java (or C), and even readers without any Javascript knowledge should be able to follow the provided code samples.

2.1 Binding of Variables

In Scheme, new variables can only be created within certain expressions (eg. `let` and `define`) which ensure that every variable is defined. Javascript however is more flexible:

- Globals do not need to be declared. They can be defined within the global scope (using the same syntax as is used for local variables in functions), but it is also possible to declare them implicitly when assigning an undeclared variable[2]. The inverse - reading from an undeclared variable - is not possible and throws an exception.
- A variable declaration (`var x;`) allows one to declare variables anywhere in a function. The variable is then set to *undefined* at the beginning of the function. Most languages provide blocks to limit the visibility of variables whereas in Javascript blocks do not influence the scoping. But even more surprising the declaration also affects all previous occurrences of the same symbol. In theory one could put all variable-declarations in a block at the end of a function.

This flexibility comes at a price though. When variables share the same name it is easy to accidently reference the wrong variables and produce buggy code. The following example contains several common mistakes.

```
1:  var x = "global";     // global variable
2:  function f() {
3:    x = "local";        // references local x
4:    var someBool = true;
5:    var x = 2;
6:    some_bool = false;  // oops.
7:    if (someBool) {
8:      var x = 1;        // references same x
9:    }
10:   return x;
11: }
12: f();       // => 1
13: x;         // => "global"
14: some_bool; // => false
```

Due to the local declaration of `x` in line *5* **and** *8* the assignment in line *3* does not change the global `x`, but the local one. Line *6* contains another annoying bug: instead of changing the local `someBool` a new global `some_bool` is created and set to `false`.

From a compiler's point of view these differences are mostly negligible though. Only the automatic assignment of *undefined* is of concern, as it makes typing less efficient.

2.2 Object System

Whereas Bigloo uses a CLOS-like [6] object system, Javascript adopted a prototype based system [13]: conceptually objects are ordinary hash tables with an attached prototype field. Whenever a property (Javascript's synonym for "member") is read (`obj.property` or `obj["property"]`) the object's hash table is searched for this entry. If the hash table contains the property the value is returned otherwise the search recursively continues on the object stored in the prototype-field. Either the member is eventually found, or the prototype does not hold an object, in which case *undefined* is returned. Writing on the other hand is always done on the first object (ie. the prototype is completely ignored). If the property did not already exist it will be created during the write.

Methods are just regular functions stored within the object. Every procedure implicitly receives a `this` argument, and when called as method (`obj.method()` or `obj["method"]()`) `this` points to the object (as in line *7* of the next example). If a function is called as non-method (line *4*) the `this`-argument is set to the *global object* which represents the top-level scope (containing all global variables and functions).

```
1: function f() {
2:    print(this);
3: }
4: f();     // 'this' in f becomes the global object
5: var o = new Object();
6: o.f = f;
7: o.f();   // 'this' in f becomes o
```

In Javascript all functions are objects, and while function invocations usually do not access the contained properties, the `prototype`-property is retrieved, when functions are used as constructors. Indeed, constructors too are just functions and do not need to be declared differently. An object creation is invoked by the construct `new Fun()`, which is decomposed and executed in three steps:

- Javascript creates a new object.
- it retrieves the `prototype`-property out of the function object's hash table (which is not necessarily identical to the prototype-field of the same object), and stores the result in the prototype-field of the newly created object.
- it runs `Fun` as if it was invoked as a method on the new object, hence allowing to modify it.

Even though the previous description is not entirely complete (we intentionally omitted some special cases), it is not difficult to show that prototype-based object-systems allow most (if not all) usual Smalltalk [11] or CLOS operations. In particular inheritance, private members or mix-ins [10] are easily feasible. Interested readers are referred to [7] for a more in-depth discussion of Javascript's object-system.

2.3 Global Object

Simply spoken, the *global object* represents the scope holding all global variables (including the functions). What differentiates Javascript from many other languages is the fact, that this object is accessible to the programmer. It is hence possible to modify global variables through an object. Interpreters simply reuse their Javascript-object structure for all global variables. Whenever needed they just provide a pointer to this structure. However for an optimizing compiler the global object is a major obstacle. The following example demonstrates how the global object disallows simple optimizations like inlining.

```
1: function g() { /* do something */ }
2:
3: function f(o) {
4:   o.g = undefined;
5:   g();
6: }
```

Suppose the given functions are part of a bigger program. Function `f` is calling the global function `g`. If `g` is never changed (eg. `g`

[2] Note, there exists a third method involving the "global object".

= some_value;), which is usually easy to detect, a good compiler could inline g. In Javascript it is however more or less impossible to be sure that g is never modified. Even the object passed to f could be the global object, and f could change g. As pointer-analyses are generally very costly and compute only conservative approximations, tracking the global object is not an option.

It is not even possible to avoid the use of global objects (as should be done with the with-construct). The global object is accessed by two ways: it is assigned to the this variable in the global scope (easily avoidable), but it is also passed to every function call, where it becomes the this-variable. Exceptions are all method-calls where the global object is replaced by the object on which the method is executed (Section 2.2 shows an example).

2.4 Variable Arity Functions

Scheme and Javascript both allow variable arity functions, but their approach is quite different. Scheme procedures must explicitly allow supplementary parameters, whereas Javascript functions are automatically prepared to receive *any* number of arguments. Even if a function's signature hints several parameters, it can still be called without passing any argument. The missing values are automatically filled with *undefined*:

```
1: function f(x) { print(x); }
2: f(); // => prints "undefined"
```

If the procedure needs to know the actual number of passed arguments, it can access the arguments-object which is available within any function. Not only does the property size hold the actual number of parameters, it also contains a reference to *all* arguments: arguments[n] accesses the nth argument. Variables in the function's signature are just aliases to these entries. The following example demonstrates the use of arguments. It will print 2, 3 and finally 2:

```
1: function f(x) {
2:   print(arguments.size); // => 2
3:   x = 3; // modify first argument
4:   print(arguments[0]);   // => 3
5:   print(arguments[1]);   // => 2
6: }
7: f(1, 2);
```

2.5 Eval Function

Scheme and Javascript both have the eval function, which allows to compile and execute code at runtime. They do not use the same environment for the evaluation, though. Scheme gives the developer the choice between the *Null-environment*, *Scheme-report-environment* or the *Interaction-environment*. The *Null-environment* and *Scheme-report-environment* are completely independent of the running program and an expression evaluated in them will always yield the same result. The optional *Interaction-environment* however allows to interact with the running program. The visibility of this environment is usually restricted to the top-level of the running program, and it is certainly independent from the location where eval is executed.[3].

Javascript, on the other hand, uses the same environment in which the eval function is executed. The evaluated code has hence access to the same variables any other statement at the eval's location would have. To ease the development of Javascript compilers, the standard gives writers the choice to restrict the use of eval to the form eval(...) (disallowing for instance o.eval(...)) and to forbid the assignment of eval (making f=eval illegal)). It is then possible to statically determine all locations and environments of eval.

3. Data structures and Types

The Javascript specification defines six types: *Undefined*, *Null*, booleans, strings, numbers and Object. This section presents the chosen representation of these types in the compiled code. Javascript's strings and booleans are directly mapped to their Scheme counterparts. As reimplementation of Javascript's numbers would have been too slow and too time-consuming, numbers are mapped to Scheme doubles. This representation does not conform to the ECMA specification[4], but the differences are often negligible. *Undefined* and *Null* are both constants and currently represented by Scheme symbols. As null is generally used for undefined objects we might replace it by a constant object in future versions of Jsigloo to improve typing.

Javascript objects however could not be mapped to any primitive Scheme (or Bigloo) type. In Javascript properties can be added and removed to objects at run-time, and Bigloo's class-system does not allow such modifications. As a result a Bigloo class Js-Object has been written that represents Javascript objects. It contains a hash table as container for these dynamic properties and a prototype-field which is needed for Javascript's inheritance. Several associated procedures simulate Javascript's property accesses and Javascript's objects are now directly mapped to the Js-Object and its methods.

Javascript functions are objects with an additional field containing the Scheme procedure. In our case Js-Function is a Bigloo class deriving from Js-Object, where a new field fun holds the procedure. A function call gets hence translated into a member-retrieval (with-access) followed by the invocation of the received procedure. Figure 1 shows the two classes and the js-call-function executing the call. (a description of this-var and arguments-vec is found in Section 4.4).

```
 1: (class Js-Object
 2:    props  ; hashtable
 3:    proto) ; prototype
 4:
 5: (class Js-Function::Js-Object
 6:    fun::procedure)  ; field of type procedure
 7:
 8: (define-inline (js-call fun-obj this-var arguments-vec)
 9:    (with-access::Js-Function fun-obj (fun)
10:       (fun this-var arguments-vec)))
```

Figure 1. Javascript's objects and functions are mapped to Bigloo classes

Javascript is dynamically typed and variables can hold values of different types during their lifetime. Most of the time programmers do not mix types though, and it is usually possible to determine a small set of possible types for each variable. Bigloo already performs an efficient typing analysis [15], but it cannot differentiate Javascript types that have been mapped to the same Scheme type (*undefined* and *null* become both symbols, objects and functions are both translated to Bigloo objects). Bigloo lacks Javascript-specific knowledge too. Depending on the operands some Javascript operations may return different types. One of these operations is the +-operator. If any operand is a string the result will be a string, otherwise the expression evaluates to a number.

As a result Jsigloo contains itself a typing pass. Contrary to Bigloo Jsigloo only implements an intraprocedural analysis resembling the implementations found in "Compiler Design Implemen-

[3] The standard is rather unclear about what this environment really represents.

[4] Javascript requires -0 and +0 to be different, which is not possible with any Scheme number type in R5RS.

tation" [14], Chapter "Data-Flow Analysis". This choice implies that parameters need to be typed to *top* (i.e. an abstract value denoting any possible type) as is the case for escaping variables: at every function-call the types could change and they need to be set to *top*. Despite these two restrictions the typing pass is able to type most expressions to some small subset. As we will see Javascript does many automatic conversions, and restricting the type-set only a little helps a lot to reduce the impact of them.

4. Compilation

Similar to Bigloo Jsigloo is decomposed into several smaller passes, which respectively execute a specific task. This first part of the section will provide a small overview over Jsigloo's architecture. The remainder of the section will then focus on the code generation. The generic case is handled first, specially treated constructs are then discussed separately. Primarily Scheme-foreign constructs like `with` (Section 4.3) and `switch` (Section 4.2) are examined in their respective subsections, but the important function compilation has its own area (Section 4.4), too. Whenever a generated code is dependent on previous optimizations we will revisit the concerned passes.

A first lexing/parsing pass constructs an abstract syntax tree (AST) composed of Bigloo objects representing Javascript constructs. Bigloo uses a CLOS like object system and it is hence possible to create procedures that dispatch calls according to their type. Jsigloo does not use any other intermediate representation other than this AST. Passes just modify the tree or update the information stored in the nodes.

An early expansion pass then removes some syntactic sugar and reduces the number of used nodes. Immediately afterwards the "Symbol" pass binds all symbols to variables. The following pass continues the removal of syntactic sugar. The optimization passes and typing is then executed before Jsigloo reaches the backend.

The code generator still receives an AST and a simplified version just needs to transform recursively the nodes to Scheme expressions and definitions. Ignoring the previously mentioned special cases and some last optimizations this transformation is straight-forward. Jsigloo just recursively dumps the nodes using generic functions and methods which are dispatched according to the type of their first argument (`define-method`). Figure 2 contains the implementations of the generic method `generate-scheme` for the `Block` and `If` nodes as well as the procedure `generate-indirect-call` used for creating unoptimized function calls.

Javascript and Scheme are very similar, and this can be seen at this level: many implementations of `generate-scheme` just retrieve the members of the node (`with-access`), transform them, and plug them into escaped Scheme lists. Most of the time only minor adjustments are needed. The `If`-method at line *9*, for instance, needs to boolify the condition expression first. That is, in Javascript 0, `null`, `undefined` and the empty string are also considered to be `false`, and conditional expressions need hence to test for these values. As we already know the type (or a superset of possible types) of every expression, some of these tests can be discarded at compile time. Instead of generating adapted code for every boolify-expression Jsigloo uses macros. This way some complexity is moved outside the compiler itself into the runtime library. Macros are still evaluated at compile time, but now within Bigloo. The `js-boolify-generate-scheme` function retrieves all possible types of the given expression and passes them to the `js-boolify_typed` macro (figure 3) as second parameter (the first one being the transformed expression). The macro then automatically discards all impossible configurations[5].

Similar `_typed`-macros are used in many other places. Even though properties of Javascript objects are always referenced by strings (`obj.prop` is transformed into `obj["prop"]`), the expression within the brackets can be of any type. Javascript therefore performs an implicit conversion to string for every access. For instance the `0` in `obj[0]` is automatically converted into `"0"`. `obj[0]` and `obj["0"]` reference hence the same property. The conversion is in this case performed by the `->string_typed`-macro which reduces the tests as much as possible. Another implicit conversion is executed for numeric operators which convert their operands to numbers (`->number_typed`). Generally every conversion has its `_typed` pendant which is used whenever possible.

```
1:  (define-method (generate-scheme b::Block)
2:     (with-access::Block b (elements)
3:        `(begin
4:           #unspecified ; avoid empty begin-blocks
5:           ,@(map generate-scheme elements))))
6:
7:  (define-method (generate-scheme iff::If)
8:     (with-access::If iff (test true false)
9:        `(if ,(js-boolify-generate-scheme test)
10:           ,(generate-scheme true)
11:           ,(generate-scheme false))))
12:
13: (define (generate-indirect-call fun this-arg args)
14:    ; JS ensures left-to-right evaluation of arguments.
15:    (if (or (null? args)         ; 0 arguments
16:            (null? (cdr args))) ; 1 argument
17:        `(js-call ,fun
18:                  ,this-arg
19:                  (vector ,@(map out args)))
20:        (let ((tmp-args (map (lambda (x)
21:                               (gensym 'tmp-arg))
22:                             args)))
23:    `(let* (,@(map (lambda (tmp-name arg)
24:                     (list tmp-name
25:                           (out arg)))
26:                   tmp-args
27:                   args))
28:              (js-call ,fun
29:                       ,this-arg
30:                       (vector ,@tmp-args))))))
```

Figure 2. the `generate-scheme-code` methods for some selected nodes.

Some Javascript constructs need more than just these minor adjustments though. In particular `switch`, `with` and even the well known `while` do not have corresponding Scheme expressions. Due to various optimizations, functions too are not directly mapped to their Scheme counterparts and are therefore discussed in a separate subsection.

4.1 While Translation

The straightforward intuitive compilation of

```
1: while(test) body
```

to

```
1: (let loop ()
2:    (if test
3:        (begin
4:          body
5:          (loop))))
```

[5] The actual `js-boolify_typed` in the Jsigloo-runtime even removes the test for the type, if the expression can only have one single type. The given code sample also misses some other object-tests.

```
(define-macro (js-boolify_typed exp types)
   (let ((x (gensym 'x)))
      `(let ((,x ,exp))
          (cond
             ,@(if (member 'bool types)
                   `(((boolean? ,x) ,x))
                   '())
             ,@(if (member 'undefined types)
                   `(((eq? ,x 'undefined)
                      #f))
                   '())
             ,@(if (member 'null types)
                   `(((eq? ,x 'null)
                      #f))
                   '())
             ,@(if (member 'string types)
                   `(((string? ,x)
                      (not (string=? ,x ))))
                   '())
             ,@(if (member 'number types)
                   `(((number? ,x)
                      (not (=fl ,x 0.0))))
                   '())
             (else #t)))))
```

Figure 3. `js-boolify_typed` used the calculated types to optimize the conversion.

misses an important point: loops in Javascript can be interrupted (`break`) or shortcut (`continue`). These kind of break-outs require either `call/cc` (or similar constructs) or exceptions. Jsigloo uses Bigloo's `bind-exit`, a `call/cc` that can only be used in the dynamic extend of its form:

```
(bind-exit (break)
   (let loop ()
      (if test
          (begin
             (bind-exit (continue)
                (body))
             (loop)))))
```

In the current Bigloo version non-escaping `bind-exits` are not yet optimized though[6] and a bind-exit removal pass has been implemented.

We used `bind-exits` not just in loops, but also for the `switch`-breaks (see next section) or the function-`returns`. In certain cases there is no easy way of avoiding them, but the following transformations are able to remove most of them. The following three samples represent some cases where our analysis allows to eliminate `bind-exits`.

```
(lambda (x)
   (bind-exit (return)
    (if (eq? x 'null) (return 'undefined))
    ;do something
   ))
```

```
(bind-exit (return)
   ; do something
   (if test
       (return 'any)
       (return 'thing))
)
```

[6] Bigloo's `bind-exit` supplies a closure, which, when invoked, unwinds the execution flow to the end of `bind-exit`'s definition (not unlike exceptions caught by a `catch`). Jsigloo uses only a small part of `bind-exit`'s functionality. The supplied closure never leaves the current procedure, and in this case invocations of bind-exit can be transformed into simple `gotos`. Future versions (post 2.7) of Bigloo will contain such an optimization.

```
(lambda (x)
   (bind-exit (return)
    ; do something
    (return result)))
```

All these examples are based on the `return` statement, but similar examples exist with the `continue` keyword of the `while` statement.

Our optimization relies on two observations:

- If an `if`-branch does not finish its execution but is interrupted (`break`, `continue`, `return` or `throw`) any remaining statements following the `if` can be attached to the other branch of the `if`[7].
- Any invocation of the escaping closure, directly followed by the end of the surrounding `bind-exit` is unnecessary and can be removed.

The first observation allows to transform the first example into:

```
(lambda (x)
   (bind-exit (return)
    (if (eq? x 'null)
        (return 'undefined)
        ;do something
     )))
```

Under the assumption that the `returns` have not been used elsewhere in the code, all `bind-exits` can now be removed thanks to the second rule:

```
(lambda (x)
    (if (eq? x 'null)
        'undefined
        ;do something
     ))
```

```
; do something
 (if test
     'any
     'thing)
```

```
(lambda (x)
    ; do something
    result)
```

This optimization removes all but one `bind-exit` from the 33 `bind-exits` found in our test-cases and benchmarks.

4.2 Switch Construct

Javascript's `switch` statement allows control to branch to one of multiple choices. It resembles Scheme's `case` and `cond` expressions, which serve the same purpose. As we will see, neither of them has the same properties as the Javascript construct, and `switch` therefore need to be translated specially.

Javascript permits non-constant expression as `case` clauses and in the following example `expr1`, `expr2` and `expr3` could thus represent any Javascript expression (including function-calls):

```
switch (expr)
   case expr1: body1
   case expr2: body2
   default:    default_body
   case expr3: body3
```

It is therefore not possible to map `switch` to Scheme's `case` which only works with constants. Scheme's `cond`, on the other

[7] If neither branch finishes normally, the remaining statements are dead code, and can hence be removed.

hand, evaluates arbitrary expressions, and if it was not for Javascript's "fall-throughs", a `switch` statement would be easily compiled into an equivalent `cond` expression:

```
1: (let ((e expr))
2:   (cond
3:    ((eq? expr1 e) body1)
4:    ((eq? expr2 e) body2)
5:    ((eq? expr3 e) body3)
6:    (else default_body))
```

As it is, a case-body falls through and continues to execute the body of the next case-clause (unless, of course, it `break`s out of the switch). To simulate these fall-throughs Jsigloo wraps the bodies into a chain of procedures. Each procedure calls the following body at the end of its corps and hence continues the control-flow at the beginning of the next clause's body. `break`s are simply mapped to `bind-exit`s and are not yet specially treated.[8]

The following code demonstrates this transformation applied to our previous example:

```
 1: (bind-exit (break)
 2:  (let* ((e expr)
 3:         (cond-body3   (lambda () body3))
 4:         (cond-default (lambda ()
 5:                         default-body
 6:                         (cond-body3)))
 7:         (cond-body2   (lambda ()
 8:                         body2
 9:                         (cond-default)))
10:         (cond-body1   (lambda ()
11:                         body1
12:                         (cond-body2))))
13:   (cond
14:    ((eq? expr1 e) (cond-body1))
15:    ((eq? expr2 e) (cond-body2))
16:    ((eq? expr3 e) (cond-body3))
17:    (else (cond-default)))))
```

Even though Javascript's default clause does not need to be the last clause, it is only evaluated once all other clauses have been tested. It is therefore safe to use the `cond`'s `else`-clause to invoke the default body, but care must be taken to include its body in the correct location of the procedure-chain.

4.3 With Statement

The access to the property *prop* of a Javascript-objects `obj` is usually either done by one of the following constructs: `obj.prop` or `obj["prop"]`. A third construction, the `with`-keyword, pushes a complete object onto the scope stack which makes all contained properties equivalent to local variables. Within interpreters this operation is usually trivial. The interpreter just needs to reuse the Javascript object type as representation of a scope. When it encounters a `with` it pushes the provided object onto their internal scope-stack. Compilers do not use explicit scope objects though, and pushing objects onto the stack is just not feasible.

Moreover, an efficient compilation of the `with`-statement is extremely difficult. As Javascript is a dynamically typed language it is not (always) possible to determine the type and hence the properties of the `with`-object. Even worse: Javascript objects might grow and shrink dynamically. It is possible to add and remove members at runtime. The following code shows an example where a variable within a closure references two different variables even though the same object is used.

[8] The current transformation has been implemented following a suggestion of a reviewer, and it was not possible to remove the `bind-exit`s in this short time-frame.

```
 1: var o = new Object();
 2: function f(x)
 3: {
 4:    with(o) {
 5:       return function() { return x; };
 6:    }
 7: }
 8: g = f(0);
 9: g();     // => 0;
10: o.x = 1; // adds x to o
11: g();     // => 1;
```

During the first invocation (line *9*) of the anonymous function of line *5* the `o` object does not yet contain `x` and the referenced `x` is hence the one of the function `f`. After we added `x` to `o` another call to `g` references the object's `x` now.

It is therefore nearly impossible to find the shadowed variables when entering a `with`-scope, but a test needs to be done at every access. As a result Jsigloo replaces all references to potentially intercepted variables by a call to a closure which is then inlined by Bigloo. This closure tests for the presence of a same-named member in the `with`-object, and executes the operation (either *get* or *set*) on the selected reference. Note that `with` constructs might be nested, and in this case the operation on the "selected reference" involves calling another function. This transformation (in a simplified version) is summarized in the following code snippet.

```
1: with(o) {
2:    x = y;
3: }
```

becomes

```
1: (let ((x-set! (lambda (val) (if (contains o x)
2:                                 (set! o.x val)
3:                                 (set! x val))))
4:       (y-get (lambda () (if (contains o y) o.y y))))
5:   (x-set! (y-get)))
```

This approach obviously introduces a performance penalty and together with the sometimes unexpected results (like the closure referencing different variables) a widely accepted recommendation is to avoid `with` completely [7].

4.4 Function Compilation

The function translation is the arguably most challenging part of a Javascript to Scheme compiler. Not only is Javascript a functional language where functions are frequently encountered, Scheme compilers usually optimize functions, and a good translation can reuse these optimizations. This section will restate the major differences between Javascript functions and Scheme procedures. We will then discuss each point separately, and detail how Jsigloo handles it. Bigloo is often unable to optimize Jsigloo's generic translation of functions, and the last part of this section presents Jsigloo's optimizations for functions.

Three primary features make the function translation from Javascript to Scheme difficult (for a more detailed discussion see Section 2):

- Every Javascript function can serve as method too. In this case every occurrence of the keyword `this` in the function's body is replaced by the object on which the function has been invoked. Otherwise `this` is replaced by the *global object*.
- It is possible to call every function with any numbers of arguments. Missing arguments are automatically filled with *undefined* and additional ones are stored in the *arguments* object.
- Javascript functions are objects.

Jsigloo's compilation of the `this` keyword is straightforward: When translating functions an additional parameter `this` is added

in front and all call-sites are adjusted: method calls pass the attached object as parameter, and function calls pass the *global object*.

Javascript functions can be called with any number of arguments and an early version of Jsigloo compiled functions to the intuitive form `(lambda (this . args) body)` to use Scheme's variable arity feature. Some measurements revealed that Bigloo was more efficient, if vectors were used instead of the implicit lists. At the call-sites a vector of all parameters is constructed, and then passed as second parameter after the `this`. A translated function is now of the following form: `(lambda (this args-vec) body)`.

Inside the function every declared parameter is then represented by a local variable of the same name. At the beginning of the procedure the local variables are either filled with their correspondent values from the arguments vector, or set to *undefined*. Figure 4 contains a simplified unhygienic version [8] of this process. The same figure shows the result for the declared parameters `a` and `b`.

```
1: `(let* ((len (vector-length vec))
2:         ,@(map (lambda (param-id count)
3:                  `(,param-id (if (> len ,count)
4:                                  (vector-ref vec ,count)
5:                                  'undefined)))
6:                param-list
7:                (iota (length param-list))))
8:    ,body)
```

```
1: (let* ((len (vector-length vec))
2:        (a (if (> len 0) (vector-ref vec 0) 'undefined))
3:        (b (if (> len 1) (vector-ref vec 1) 'undefined)))
4:   body)
```

Figure 4. the Jsigloo-extract at the top generates the code responsible for extracting the values out of the passed `vec`. The code at the bottom gets generated for the parameters `a` and `b`.

After the variable extraction Jsigloo creates the arguments object. As the `arguments`-entries are aliased with the parameter variables (`a` and `b` in the previous example) we use the same technique as for the `with` statement: the entries within the `arguments` object are actually closures modifying the local variables. Additional arguments access directly the values within the vector. Figure 5 demonstrates this transformation.

As has already been stated in Section 3, Javascript functions are mapped to the Bigloo class `Js-Fun`, which contains a field `fun` holding the actual procedure. Jsigloo's runtime library provides the procedure `make-js-function`, which takes a Scheme procedure with its arity and returns such an object. Jsigloo only needs to translate the bodies of Javascript functions, and generate code, that calls this runtime procedure with the compiled function as parameter . The returned object of type `Js-Fun` is compatible with translated Javascript objects. As the compiled function is now stored within an object, function calls are translated into a member retrieval, followed by the invocation of the received procedure.

The overhead introduced by these transformations is substantial: the compilation of the simple Javascript function `function f(a, b) {}` produces a Scheme expression of more than 20 lines, and the applied transformations are extremely counter-productive to Bigloo's optimizations. Storing the procedure in an object efficiently hides it from Bigloo's analyses. The Storage Use Analysis [15] (henceforth SUA), responsible for typing, and Bigloo's inlining pass are both powerless after this transformation. The arguments are then obfuscated by storing them in vectors, where Bigloo's constant propagation can not see them. When building the arguments objects they are furthermore accessed from inside a closure, which makes them slower to access.

```
1: `(let ((len (vector-length vec))
2:        (arguments (make-Arguments-object)))
3:    ,@(map (lambda (param-id count)
4:             `(if (> len ,count)
5:                  (add-entry arguments
6:                     (lambda () ,param-id)
7:                     (lambda (new-val)
8:                       (set! ,param-id new-val)))))
9:           param-list
10:          (iota (length param-list)))
11:   (let loop ((i ,(length param-list)))
12:      (if (> len i)
13:          (begin
14:            (add-entry arguments
15:               (lambda () (vector-ref vec i))
16:               (lambda (new-val)
17:                  (vector-set! vec i new-val)))
18:            (loop (+ i 1)))))
19:    ,body)
```

```
1: (let ((len (vector-length vec))
2:       (arguments (make-Arguments-object)))
3:   (if (> len 0)
4:     (add-entry arguments
5:       (lambda () a)
6:       (lambda (new-val) (set! a new-val))))
7:   (if (> len 1)
8:     (add-entry arguments
9:       (lambda () b)
10:      (lambda (new-val) (set! b new-val))))
11:  (let loop ((i 2))
12:    (if (> len i)
13:        (begin
14:          (add-entry arguments
15:            (lambda () (vector-ref vec i))
16:            (lambda (new-val)
17:               (vector-set! vec i new-val)))
18:          (loop (+ i 1)))))
19:  body)
```

Figure 5. the Jsigloo code at the top is responsible for the `arguments` creation in the emitted result. The bottom is generated for parameters `a` and `b`.

Jsigloo contains some optimizations addressing these issues. A simple one eliminates unnecessary lines: the creation of the arguments object is obviously only needed if the variable `arguments` is referenced inside the function. Otherwise Jsigloo just omits these lines.

In order to benefit from Bigloo's optimizations the indirect function calls need to be replaced by direct function calls wherever possible. Jsigloo's analysis is still relatively simple, but it catches the common case where declared (local or global) functions are directly called. The optimization is not yet correct though, and in its current form it needs to set an important restriction on the input: the given program must not modify any declared functions over the global object or in an `eval` statement. Section 6 discusses the necessary changes for the removal of this restriction.

Single Assignment Propagation (SAP) performs its optimization in two steps. First it finds all assignments to a variable and stores it in a set. Then it propagates constant values (including functions) of every variable that is assigned only once in the whole program.

Computing the definition-set is easy, but not trivial: Javascript automatically sets all local variables to *undefined* at the beginning of a function, and nearly every variable is hence modified at least twice. Once it is assigned to *undefined* and then to its initial value. Declared functions (global and local) are immediately set to their body and are hence treated accordingly. For all others a data-flow analysis needs to determine, if the variable might be used undefined. This analysis is mostly intraprocedural, and only needs one

pass. Some parts are however interprocedural as escaping variables cross function boundaries. Take for instance the following code:

```
1: function f() {
2:   var y = 1;
3:   var g = function() { return x + y + z; };
4:   var z = 2;
5:   g();
6:   var x = 3;
7:   return x + y + z;
8: }
```

Even though within `f` the variable `x` is read only after the definition in line *6*, the call at line *5* still uses the undefined variable. `y` on the other hand is always used after its first (and unique) definition. Usually these cases are difficult to catch, but SAP manages to find at least the most obvious ones: if a variable is defined before an anonymous function has been declared (as is the case for `y` in our example), the analysis does not add the implicit *undefined* definition to the variables definition set. SAP does hence correctly set `y`'s definition set to the assignment in line *2*, but will find two definitions for `z`. At the moment of `g`'s declaration `z` is still undefined, and as it is used within `g` the final definition set of `z` will hold the implicit *undefined*-definition and the assignment at line *4*.

The implicit *undefined* assignments are disturbing Bigloo's optimizations too. Whenever in doubt Jsigloo sets the variable to *undefined* at the beginning of a function. One of the first analyses Bigloo applies is the SUA-analysis, which detects the assignment of *undefined* and types the variable accordingly. Even if Bigloo is able to remove this assignment later on, it will not retype the variable, and misses precious optimization opportunities.

Once the definition-set has been determined, a second pass propagates "single assignments". If a variable has only one assignment in its definition set, and this assignment sets the variable to a constant value or a Javascript function, all occurrences of this variable are replaced by either the constant, or by a reference to this function. In our example the line *7* still uses `x` and `z`, as their definition-sets contain more than one assignment. The optimization transforms our previous example into the following code:

```
1: function f() {
2:   var y = 1;
3:   var g = function() { return x + 1 + z; };
4:   var z = 2;
5:   anonymous_g();
6:   var x = 3;
7:   return x + 1 + z;
8: }
```

Wherever the backend finds direct function-references it is now able to optimize the call. Instead of extracting the procedure from the function object it can use the function-reference. The previous creation of function objects must first be modified to allow access to the procedure:

```
1: (set! direct_f (lambda (this vec) body))
2: (set! f (make-js-function direct_f 2))
```

In our benchmarks and test-cases 27% of all function calls could be replaced by direct function calls after this analysis.

Wherever Jsigloo is able to replace the indirect calls with direct calls it can also improve the parameter passing. The function's signature provides the expected number of arguments, and the parameters do not need to be hidden in a vector anymore. If there are missing arguments, they can already be filled with *undefined* constants at compile-time. An additional `arg-nb` parameter passes the original number of arguments, which is needed for the creation of the arguments object. The last argument finally contains additional arguments, that have not been mapped to direct parameters. They will be used during `arguments` creation, too. Obviously the generic call needs to be adapted too, and the parameter-extraction of figure 4 is lifted into the procedure passed to the `make-js-function`.

Many functions do not use `this` and in this case the first argument can be removed. The same is of course true for `arg-nb` and `rest-vec`, which are only needed, if the function uses the arguments-object. Our running example is finally transformed into the following code:

```
1: (set! direct_f (lambda (a b) body))
2: (set! f (make-js-function
3:             (lambda (this vec)
4:               (let* ((len (vector-length vec))
5:                      (a (if (> len 0)
6:                             (vector-ref vec 0)
7:                             'undefined))
8:                      (b (if (> len 1)
9:                             (vector-ref vec 1)
10:                            'undefined)))
11:                (direct_f a b))
12:            2))
```

Applying these optimizations to the well known Fibonacci function let the size of procedure drop from more than 75 to about 20 lines[9], and reduce execution time by a factor of more than 20.

5. Performance

	Ack	Fib	Meth	Nest	Tak	Hanoi
Jsigloo J	1931	443	185	**898**	28	424
Rhino	1042	666	155	973	55	619
Jsigloo C	**513**	**368**	84	1060	**11**	**368**
Konqueror	-	17183	262	15478	593	21049
Firefox	-	3179	227	1808	79	2762
NJS	-	767	**23**	1481	25	734

Jsigloo is not yet finished, and the given benchmarks (see the appendix for the sources) are therefore just indications. As we wanted to be able to run our benchmarks on most existing Javascript implementations we decided to move the time-measurement into the benchmark itself. This way it was possible to benchmark Internet browsers too. At the same time we lost the start-up overhead, and the more precise measurement of the Linux kernel. All times have been taken under a Linux 2.6.12-nitro on an AMD Athlon XP 2000+, and are expressed in Milliseconds. We used Sun's JDK 1.4.2.09 (HotSpot Client VM, mixed mode) and GCC 3.4.4. We ran every benchmark at least three times, and report the fastest measured time here. Konqueror [2], Firefox [1] and NJS [3] where not able to complete Ack (stack overflows) and do not have a time for this benchmark.

"Jsigloo J" uses Bigloo's JVM backend, whereas "Jsigloo C" targets C, followed by a compilation to native code. "Rhino", in version 1.6R2RC2, compiles Javascript directly to JVM bytecode and competes hence with "Jsigloo JVM". The fastest time on the JVM machine is underlined. "Konqueror" 3.4.2, "Firefox" 1.0.6 and "NJS" 0.2.5 are all interpreters (even though NJS was allowed to precompile the Javascript code into its bytecode format) and are compared to "Jsigloo C". The fastest time is in bold.

During the development these benchmarks have been (and are still) used to pinpoint weak spots of Jsigloo, which were then improved. One of the first benchmarks has been Fibonacci, which explains Jsigloo's good results in some of the other call-intensive benchmarks (Hanoi and Tak). "Nest", as the name hints, increments a number within nested loops. We verified our results and for this benchmark the Java version is actually faster than the native

[9] We are well aware, that this is still far away from a standard 5 lines implementation, but most of the resting lines are redundant `lets`, `begins` or `#unspecified` which are easily removed by Bigloo.

C version. The JVM version of Ackermann is still slower than Rhino's code, but we have pinpointed the source of inefficiency and a generic transformation brings the time for "Ack" down to the same level as Rhino. "Meth" on the other hand makes heavy use of anonymous functions and objects, and this part of Jsigloo is not yet optimized at all.

Note, that Jsigloo is not conformant to the ECMA specification (see Section 6), and has therefore an unfair advantage over the competitors. Some tests showed that Fibonacci's execution time would double if the global object was treated correctly. Other experiences however confirmed, that for instance a fully optimizing Rhino is not conformant either, and especially the *global object* is equally ignored.

6. Future Work

Jsigloo is not finished. Several Javascript features have not yet been implemented and some parts of Jsigloo are not conformant to the ECMA specification. From the more than 10 runtime objects, only two have been written until now (in particular the `Boolean`, `String`, `Number` and `Date` objects are still missing). Due to limitations in the used lexer-generator, some syntactic sugar is missing too (Javascript's automatic semicolon insertion and its regular expression literals).

At the moment Jsigloo does not handle the global object correctly either. It is not possible to modify global variables over an object, and function calls receive a standard `Object` as `this`. We intend to fix this shortcoming by adding two strategies:

- a "correct" solution using a special global object, that holds closures. Whenever a field is modified the closure automatically updates the real global object. (Inversely reading from the global object automatically redirects to the real global variable). A similar strategy is already being used for the `arguments` object and the `with` translation.
- a fast implementation which disallows the use of the global object. Every access to the global object throws an exception.

The `eval` function is missing too. Javascript's and Scheme's `eval` specification are different and incompatible, but Bigloo provides some extensions to Scheme's `eval` which should allow the implementation without too much trickery.

Once either the `eval`-function or the global object is correctly implemented, the SAP optimization of Section 4.4 needs to be adapted. Functions that are visible to `eval` statements and global functions might not be called directly anymore.

Finally the number representation needs to improved. Javascript numbers are mapped to Scheme doubles. In R^{5}RS [12] doubles do not provide enough functionality to correctly represent Javascript numbers, but R^{6}RS will extend Scheme's number specification, and we will revisit this topic once R^{6}RS has been released.

7. Conclusion

We presented in this paper Jsigloo, a Javascript to Scheme compiler we implemented during the last five months. Together with Bigloo it compiles Javascript to Java byte-code, C, or .NET CLI. In the introduction we listed the features Jsigloo should have. We wanted the compiler to be small. Jsigloo is not very big, but with about 30.000 lines of Scheme code Jsigloo is not small anymore. It is still easy to maintain the project, but the effort required is higher than we hoped it would be. Jsigloo's size is explained by the optimizations we integrated in Jsigloo, and preliminary benchmarks show that Jsigloo/Bigloo has the potential to be as fast as the fastest existing Javascript compilers. As Jsigloo uses Bigloo it interfaces with all languages Bigloo interfaces and excels in this area. Furthermore, if Bigloo improves, Jsigloo/Bigloo will improve too.

Despite Javascript's resemblance to Scheme, we could not take full advantage of all Bigloo optimizations and needed to implement additional optimization passes. SAP (Section 4.4) enhances direct method calls, a `bind-exit`-removal pass (Section 4.1) eliminates unnecessary (but currently expensive) `bind-exits`, and typing (Section 3) improves the ubiquitous conversions of Javascript and helps several Bigloo optimization by providing Javascript-specific type information.

Compiling to Scheme and using an efficient existing Scheme compiler did not fulfill all our expectations, but still yielded an interesting compiler. Once all missing features are implemented Jsigloo may be an attractive alternative to all other Javascript compilers.

[1] **http://www.mozilla.org/products/firefox/**.
[2] **http://www.konqueror.org/**.
[3] **http://www.njs-javascript.org/**.
[4] **http://www.caucho.com/articles/990129.xtp**.
[5] **http://www.mozilla.org/rhino/**.
[6] Bobrow, D. and DeMichiel, L. and Gabriel, R. and Keene, S. and Kiczales, G. and Moon, D. – **Common lisp object system specification** – special issue, Sigplan Notices, (23), Sep, 1988.
[7] D. Flanagan – **JavaScript - The definitive guide** – *O'Reilly & Associates*, 2002.
[8] Dybvig, K. and Hieb, R. and Bruggeman, C. – **Syntactic abstraction in Scheme** – Lisp and Symbolic Computation, 5(4), 1993, pp. 295–326.
[9] ECMA – **ECMA-262: ECMAScript Language Specification** – 1999.
[10] Flatt, M. and Krishnamurthi, S. and Felleisen, M. – **Classes and Mixins** – Symposium on Principles of Programming Languages, Jan, 1998, pp. 171–183.
[11] Goldberg, A. and Robson, D. – **Smalltalk-80: The Language and Its Implementation** – *Addison-Wesley*, 1983.
[12] Kelsey, R. and Clinger, W. and Rees, J. – **The Revised(5) Report on the Algorithmic Language Scheme** – Higher-Order and Symbolic Computation, 11(1), Sep, 1998.
[13] Martin Abadi and Luca Cardelli – **A theory of objects** – *Springer*, 1998.
[14] Muchnick, S. – **Advanced Compiler Design Implementation** – *Morgan Kaufmann*, 1997.
[15] Serrano, M. and Feeley, M. – **Storage Use Analysis and its Applications** – 1fst Int'l Conf. on Functional Programming, Philadelphia, Penn, USA, May, 1996, pp. 50–61.
[16] Serrano, M. and Weis, P. – **1+1=1: an optimizing Caml compiler** – ACM Sigplan Workshop on ML and its Applications, Orlando (Florida, USA), Jun, 1994, pp. 101–111.

8. Appendix

Benchmarks

Ackermann

```
1: function ack(M, N) {
2:     if (M == 0) return(N + 1);
3:     if (N == 0) return(ack(M - 1, 1));
4:     return(ack(M - 1, ack(M, (N - 1))));
5: }
6:
7: ack(3, 8);
```

Fibonacci

```
1: function fib(i) {
2:    if (i < 2)
3:       return 1;
4:    else
5:       return fib(i-2) + fib(i-1);
6: }
7: fib(30);
```

Method Calls

```
function methcall(n) {
   function ToggleValue () {
      return this.bool;
   }
   function ToggleActivate () {
      this.bool = !this.bool;
      return this;
   }

   function Toggle(start_state) {
      this.bool = start_state;

      this.value = ToggleValue;
      this.activate = ToggleActivate;

   }

   function NthToggleActivate () {
      if (++this.count > this.count_max) {
         this.bool = !this.bool;
         this.count = 1;
      }
      return this;
   }

   function NthToggle (start_state, max_counter) {
      this.base = Toggle;
      this.base(start_state);
      this.count_max = max_counter;
      this.count = 1;

      this.activate = NthToggleActivate;

   }
   NthToggle.prototype = new Toggle;

   var val = true;
   var toggle = new Toggle(val);
   for (i=0; i<n; i++) {
      val = toggle.activate().value();
   }
   var tmp = (toggle.value() ? "true"
                             : "false");

   val = true;
   var ntoggle = new NthToggle(val, 3);
   for (i=0; i<n; i++) {
      val = ntoggle.activate().value();
   }
   return (tmp + " " +
             (ntoggle.value() ? "true"
                              : "false"));
}

methcall(10000);
```

Nested Loops

```
function nested(n) {
   var x=0;
   var a=n;
   while(a--) {
      var b=n; while(b--) {
         var c=n; while(c--) {
            var d=n; while(d--) {
               var e=n; while(e--) {
                  var f=n; while(f--) {
                     x++;
                  }
               }
            }
         }
      }
   }
   return x;
}

nested(14);
```

Tak

```
function tak(x, y, z) {
   if (!(y < x))
      return(z);
   else {
      return (
         tak (
            tak (x-1, y, z),
            tak (y-1, z, x),
            tak (z-1, x, y)
            ));
   }
}

tak(18, 12, 6);
```

Towers of Hanoi

```
function towers(nb_discs, source, dest, temp) {
   if (nb_discs > 0) {
      return towers(nb_discs - 1,
                    source,
                    temp,
                    dest)
             + 1
             + towers(nb_discs - 1,
                      temp,
                      dest,
                      source);
   }
   return 0;
}

towers(20, 0, 1, 2);
```