Quasi-Static Scoping: Sharing Variable Bindings Across Multiple Lexical Scopes*

Shinn-Der Lee and Daniel P. Friedman
Computer Science Department
Indiana University
Bloomington, Indiana 47405
sdlee@cs.indiana.edu dfried@cs.indiana.edu

Abstract

Static scoping embodies a strong encapsulation mechanism for hiding the details of program units. Yet, it does not allow the sharing of variable bindings (locations) across independent program units. Facilities such as module and object systems that require cross references of variables therefore must be added as special features. In this paper we present an alternative: quasi-static scoping. Quasi-static scoping is more flexible than static scoping, but has the same encapsulation mechanism. The user can control when and in what scope to resolve a quasi-static variable, *i.e.*, to associate it with a variable binding. To demonstrate its versatility, we add quasi-static scoping to Scheme and show how to build the aforementioned facilities at the user-level. We also show that quasi-static scoping can be implemented efficiently.

1 Introduction

Static (Lexical) scoping is ideal for hiding the internal details of program units. In addition, a static variable's reference occurrences can all be identified syntactically at compile time. Thus, there is little, if any, computation needed at run time to *resolve* a variable reference, *i.e.*, to determine the variable binding (location) denoted by the reference. Consequently, static scoping facilitates efficient implementation.

Yet, static scoping does not allow variables to be shared across independent scopes [14]. Facilities such as module and object systems that support cross references of variables among lexical scopes thus must be added as special features. They provide the necessary scoping information to assist the evaluator (compiler or interpreter) in resolving variable references.

*This research was partially supported by the National Science Foundation under grants CCR 89-01919 and CCR 90-00597.

ACM-20th PoPL-1/93-S.C., USA
© 1993 ACM 0-89791-561-5/93/0001/0479...$1.50

In this paper we present an alternative: *quasi-static scoping*. With quasi-static scoping, a free variable in an expression can be designated as quasi-static. A quasi-static variable is initially unresolved, *i.e.*, not associated with any variable binding. To avoid introducing dynamic behavior, it has no default binding in the expression's evaluation-time environment. Rather, the user must resolve it prior to dereferencing; otherwise, it is a dangling reference and raises an error.

We therefore provide a resolution operation to associate a quasi-static variable in an expression with a variable binding in any scope. This resolution is performed independently of the given expression's evaluation, and it need not resolve all the quasi-static variables of the expression at once. Thus when and in what scope a quasi-static variable is resolved is entirely at the *user*'s discretion.

Moreover, once a quasi-static variable is resolved, it remains fixed as if it had been a static variable, hence the term quasi-static. It thus provides the same encapsulation mechanism as static scoping and therefore the same degree of program modularity and security. In addition, since a resolved quasi-static variable is a static variable, run-time dereferencing requires no extra work.

Any renaming (α-conversion) of a static variable is done consistently throughout the variable's syntactically evident finite scope. Hence, a static variable's resolution can safely depend on its name. The name-based resolution of a quasi-static variable, however, does not survive under α-conversion, since the scope of a shared quasi-static variable cannot be determined lexically. We avert this problem by recognizing that every variable has two names: an *internal* name and an *external* name. The former is α-convertible and is used in the finite scope of every static and quasi-static variable. The latter is not subject to renaming and is used by the quasi-static variable resolution operation to associate a reference in one scope with a binding in another scope.

In the next section we extend Scheme with quasi-static scoping. In Section 3 we demonstrate the expressive power of quasi-static scoping by using it as a basic building block for constructing module and object systems. In the follow-

ing two sections we present a denotational semantics and an efficient implementation of quasi-static scoping. In the last section we make comparisons to other work and discuss future research directions.

2 Extending Scheme with Quasi-Static Scoping

Scheme is an elegant statically-scoped functional language with imperative features that has a simple and clean semantics [4]. In this section we add quasi-static scoping to Scheme, showing that quasi-static scoping and static scoping can co-exist and complement each other.

2.1 Quasi-Static Procedures

A quasi-static variable is not as static as a static one; it should only be used in circumstances where a static variable is inadequate. Thus, a variable is static unless stated otherwise. In an applicative-order language like Scheme, a procedure is the only value with possible references to free variables. Hence, we add a new type of procedure, called a quasi-static procedure, that has some of its free variables quasi-statically bound. A quasi-static variable is unresolved, *i.e.*, not associated with a variable binding, when the procedure is defined. It is later resolved to some existing variable binding at the user's discretion.

A quasi-static procedure is the result of evaluating a **qs-lambda0** expression of the following form:

(**qs-lambda0** ((q_1 Q_1) ... (q_k Q_k)) *formals body*)

The names q_1, ..., q_k, where $k \geq 0$, and Q_1, ..., Q_k are the internal and external names of the procedure's quasi-static formal parameters, respectively. They are identified by their positions in a **qs-lambda0** form. To further distinguish them visually, however, we use upper-case for the external ones and lower-case for the internal ones. Like the static *formals*, the quasi-static formals q_1, ..., q_k are also the procedure's *bound* variables with a finite scope consisting of only the body expression. Consequently, like the static formals, they are subject to α-conversion. Their external names, however, are not. In order to avoid ambiguities, the formal parameters, static and quasi-static, must have different internal names, and the external names of the quasi-static formals must be different as well.

When a **qs-lambda0** expression is evaluated, the environment in effect defines the free static variables referenced in the body; the quasi-static formals are unresolved and are known outside through their corresponding external names. Calling a quasi-static procedure binds its static formals but not its quasi-static formals. The latter are associated with variable bindings using **resolve1** described below.

For example, the two expressions in Figure 1 define two independent quasi-static procedures *odd*? and *even*?. The procedure *odd*? has an unresolved quasi-static formal parameter whose internal and external names are *even*? and *F*, respectively. We can rename the internal name just as we might rename a bound variable. Thus, the expression

```
(define odd?
  (qs-lambda0 ((foo F)) (x)
    (if (zero? x) #f (foo (- x 1)))))
```

defines the same *odd*? procedure. On the other hand, the resolution operation relies on the external name *F* to associate *even*? with a variable binding. Hence, it is not subject to α-conversion.

```
(define odd?
  (qs-lambda0 ((even? F)) (x)
    (if (zero? x) #f (even? (- x 1)))))
(define even?
  (qs-lambda0 ((odd? G)) (x)
    (if (zero? x) #t (odd? (- x 1)))))
```

Figure 1: Two independent quasi-static procedures.

The two procedures *odd*? and *even*? have no access to each other; they are not mutually recursive. Applying the procedure *odd*? in Figure 1 binds *x*, but the quasi-static formal parameter *even*? remains unresolved. Hence, the application (*even*? (− *x* 1)) raises an unbound variable error.

One of our design criteria is to make quasi-static scoping as orthogonal to Scheme as possible. Quasi-static procedures are therefore disjoint from ordinary Scheme procedures. Thus, we need a new predicate (*qs-procedure? exp*) that returns true only if the expression *exp* evaluates to a quasi-static procedure. Hence, (*qs-procedure?* (**lambda** (*x*) *x*)) is false as is (*procedure?* (**qs-lambda0** () (*x*) *x*)), even though the latter has no quasi-static formals and behaves exactly like (**lambda** (*x*) *x*).

2.2 Quasi-Static Variable Resolution

The only operation that can resolve a procedure's quasi-static formal parameters is the **resolve1** special form. It maps a quasi-static procedure into another distinct quasi-static procedure with possibly fewer unresolved quasi-static formal parameters. A **resolve1** expression has the form

(**resolve1** *x Q qs-proc*)

(The 1 of **resolve1** indicates that it involves only a single variable.) The variable *x* must be associated with a variable binding and the expression *qs-proc* must evaluate to a quasi-static procedure, say, *f*. The **resolve1** expression then returns a new quasi-static procedure *g*. For each unresolved quasi-static formal parameter *q* of *f*, if its external name is *Q*, the *q* in *g* becomes an *alias* of *x*. That is, the *q* in *g* is associated with the variable binding of *x*. Furthermore, such a resolved *q* becomes a static variable of *g* and therefore does not participate in future resolution operations on *g*. Otherwise, *q* remains unresolved in *g*.

```
(letrec ((my-odd? (resolve1 my-even? F odd?))
         (my-even? (resolve1 my-odd? G even?)))
  (my-odd? 6))
```

Figure 2: Mutual recursion with resolution.

We can use **resolve1** to construct programs from existing compiled programs, rather than their source code, by linking their quasi-static variables. For instance, Figure 2 shows how to build a mutually recursive version of the two procedures *odd?* and *even?* of Figure 1 with **resolve1**. The first **resolve1** expression returns a quasi-static procedure, called *my-odd?*, that is similar to *odd?* except that its quasi-static variable *even?*, having external name F, is aliased to the variable *my-even?*. Similarly, the quasi-static variable *odd?* of the procedure *my-even?* is an alias of the variable *my-odd?*. Hence, *my-odd?* and *my-even?* are mutually recursive and the expression in Figure 2 is equivalent to

```
(letrec ((my-odd?
          (qs-lambda0 () (x)
            (if (zero? x) #f (my-even? (- x 1)))))
         (my-even?
           (qs-lambda0 () (x)
             (if (zero? x) #t (my-odd? (- x 1))))))
  (my-odd? 6))
```

2.3 Currying Quasi-Static Procedures

The addition of **qs-lambda0** takes away an important program transformation technique: currying. For instance, consider currying the static formal parameters of the following quasi-static procedure

```
(qs-lambda0 ((x X)) (a b)
  (x a b))
```

Because x is not used until the inner procedure of the curried result is applied, we would like to allow X to be resolved at either of the two nested quasi-static procedures

```
(qs-lambda0 ?1 (a)
  (qs-lambda0 ?2 (b)
    (x a b)))
```

Putting $((x\ X))$ in either $\underline{?_1}$ or $\underline{?_2}$ and placing $()$ in the other is incorrect, because X can then only be resolved at one of the two quasi-static procedures. Putting $((x\ X))$ in both $\underline{?_1}$ and $\underline{?_2}$ is equivalent to placing it only in $\underline{?_2}$, since the inner x shadows the outer one. So, that also does not work.

What we need is something that can provide two (in general more than one) occasions to resolve X. In addition, they should be connected in such a way that if X is resolved at the outer procedure then the X of the inner procedure is also resolved and has the same variable binding. Otherwise, the inner procedure provides another chance to resolve X.

Hence we define the following **qs-lambda** special form

```
(qs-lambda ((iq1 oq1) ... (iqm oqm))
           ((q1 Q1) ... (qk Qk))
           formals body)
```

A **qs-lambda** expression generalizes a **qs-lambda0** expression with the extra list $((iq_1\ oq_1)\ \ldots\ (iq_m\ oq_m))$, where $m \geq 0$. (The 0 of **qs-lambda0** means its m is 0.) The list indicates that the inner quasi-static formals $iq_1, \ldots, iq_m$ *lexically inherit* the status of the outer quasi-static variables $oq_1, \ldots, oq_m$, respectively. That is, in the resulting quasi-static procedure, iq_j is an alias (having the same variable binding) of oq_j, provided oq_j is resolved. Otherwise, iq_j is an unresolved quasi-static formal parameter with the same external name as oq_j and is therefore resolvable. Furthermore, because lexical inheritance is unidirectional, resolving iq_j does not resolve oq_j as well. So, it is possible for iq_j to be resolved but not oq_j. Again, the inheriting quasi-static formals $iq_1, \ldots, iq_m$ must be different from the other formals. They are the procedure's bound variables and are therefore subject to α-conversion. The outer ones, however, are free within the expression.

Now, the curried version of the above two-argument quasi-static procedure is

```
(qs-lambda0 ((x X)) (a)
  (qs-lambda ((x x)) () (b)
    (x a b)))
```

We can resolve X at either quasi-static procedure. If X is resolved before the outer procedure is called, the inner x lexically inherits the identity of the outer x. Thus, it is a static variable of the inner procedure. On the other hand, if X is unresolved at the outer procedure, the inner x lexically inherits the external name X. It is therefore still a resolvable quasi-static formal.

The ability to lexically inherit quasi-static variables has applications beyond expressing currying. We can use it to implement Common Lisp's optional keyword parameters [18]. Consider the quasi-static procedure f defined in Figure 3. It has a quasi-static parameter a named externally as A that is lexically inherited by the inner quasi-static procedure (**qs-lambda** $((a\ a))$ () () (**set!** a $(+\ n\ a)$) a).

When f is called with 3, A is unresolved. Consequently, the inner quasi-static procedure becomes (**qs-lambda** $((a\ A))$ () () (**set!** a $(+\ n\ a)$) a). But before the inner quasi-static procedure is called, its A is resolved with *private-val*. Therefore, a is aliased to *private-val*, which acts as the default binding of a. Hence, the result of $(f\ 3)$ is $(+\ 3\ 1)$. Also, each call to f generates a distinct *private-val* so that the assignments (**set!** *private-val* $(+\ n$ *private-val*$)$) do not affect one another. If we intend to share *default-val* among all calls to f, however, we should remove the inner **let** and replace (**resolve1** *private-val* ...) with (**resolve1** *default-val* ...).

On the other hand, invoking g on 3 yields $(+\ 3\ 10)$. The quasi-static parameter a is an alias of y. Because A has been resolved, the second **resolve1** operation has no effect

```
(define f
  (let ((default-val 1))
    (qs-lambda0 ((a A)) (n)
      (let ((private-val default-val))
        ((resolve1 private-val A
           (qs-lambda ((a a)) () ()
             (set! a (+ n a))
             a)))))))
(define g
  (let ((y 10))
    (resolve1 y A f)))
```

Figure 3: Optional keyword parameter.

```
(resolve ((x1 Q1) (x2 Q2) ...) qs-proc)
⇒ (resolve1 x1 Q1
    (resolve ((x2 Q2) ...) qs-proc))
(resolve () qs-proc)
⇒ qs-proc
```

Figure 4: Syntactic extension **resolve**.

```
(mk-resolver (x1 Q1) ... (xn Qn))
⇒ (lambda (qs-proc)
    (resolve ((x1 Q1) ... (xn Qn)) qs-proc))
```

Figure 5: Syntactic extension **mk-resolver**.

on the inner quasi-static procedure. The formal parameter a therefore behaves just like an optional keyword parameter of f. Its external name A is the keyword and its default value is *default-val*.

Whereas static formals can be curried, quasi-static formals are not. For example,

```
(qs-lambda0 ((x X) (y Y)) (a b)
  (x (y a b)))
```

is different from

```
(qs-lambda0 ((x X)) (a)
  (qs-lambda ((x x)) ((y Y)) (b)
    (x (y a b))))
```

because in the latter we cannot resolve Y at the outer procedure. Besides, since there is no ordering between X and Y, it is unclear which one should come first.

2.4 Derived Forms

In addition to the two new special forms **qs-lambda** and **resolve1**, we define two syntactic extensions **resolve** and **mk-resolver**.

The **resolve1** operation involves only a single variable. To express the same operation for multiple variables, we use the **resolve** syntactic extension defined in Figure 4. Alternatively, we could take **resolve** as a core form and define **resolve1** as a degenerate case of **resolve**. Indeed, the alternative view is beneficial from the perspective of implementation efficiency. It will not only avoid allocating space for the quasi-static procedures generated by the intermediate steps, but also eliminate the time taken to generate and garbage collect them.

A quasi-static procedure "imports" variable bindings and a resolution operation "exports" them. They complement each other. But, whereas a quasi-static procedure is a first-class value, a resolution operation is not. Fortunately, we can abstract over the operand of a resolution operation to make it into a first-class value. This is captured by the **mk-resolver** syntactic extension defined in Figure 5, where all the external names are different. Such a one-argument procedure is a quasi-static procedure transformer we call a *resolver*. It is essentially a first-class environment of the variables $x_1, \ldots, x_n$ known externally through the names $Q_1, \ldots, Q_n$. For instance, the expression

```
(define pt
  (let ((x 3) (y 4))
    (mk-resolver (x X-COR) (y Y-COR))))
```

returns a resolver *pt* with external names *X-COR* and *Y-COR*.

Besides constructing it from scratch, there are two other ways to build a resolver. First, we can construct a sub-resolver of a resolver by extracting variable bindings from the latter. For instance, consider the expressions

```
(define x-of-pt
  ((pt (qs-lambda0 ((z X)) ()
         (mk-resolver (z X))))))
(define y-of-pt
  ((pt (qs-lambda0 ((z Y)) ()
         (mk-resolver (z Y))))))
```

By matching the external name X, the quasi-static formal z is aliased to the variable x of *pt*. Then the resulting zero-argument procedure is called to return a resolver of z under the external name X. Therefore, *x-of-pt* is a sub-resolver of *pt*. Similarly, *y-of-pt* is a sub-resolver of *pt*.

Second, we can construct a new resolver by superimposing one resolver on another. For example, the expression

```
(define pt0 (superimpose x-of-pt y-of-pt))
```

where *superimpose* is

```
(define superimpose
  (lambda (f g)
    (g (f x))))
```

defines a resolver *pt0* that is the same as *pt* in behavior.

The way *x-of-pt* is constructed above relies on the fact that we know the resolver *pt* has a binding with the external name X. If we have no access to that knowledge, we can use the following **defined?** syntactic extension to determine whether the external name X is defined in the resolver *pt*

```
(defined? Q resolver)
⇒ ((resolver (qs-lambda0 ((q Q)) ()
                (let ((unique "unique"))
                  ((resolve1 unique Q
                     (qs-lambda ((q q)) () ()
                       (not (eq? q unique)))))))))
```

Briefly, Q is an optional keyword parameter with a unique default value *unique*. If *resolver* has a binding for Q, the variable q is aliased to that binding and therefore the test (*not* (*eq*? *q unique*)) is true. Otherwise, q assumes the default binding *unique* and so the test fails.

The same problem, however, is undecidable for a quasi-static procedure. We cannot tell if a quasi-static procedure has an unresolved quasi-static formal parameter with a particular external name. Interestingly, the problem becomes decidable if we allow a quasi-static procedure with no unresolved quasi-static formals to be coercible into an ordinary Scheme procedure (cf. Section 2.1). Briefly, by resolving the given procedure with external names defined in some predefined order and testing whether each result is a Scheme procedure, we can determine the "largest" external name associated with the procedure. Repeating the process until there are no more unresolved quasi-static formal parameters and we have the external names of all the unresolved quasi-static formals in descending order.

From the perspective of information hiding, we consider this asymmetry desirable. What is exported should be made public whereas what is imported should be kept private.

3 Module and Object Systems

To further demonstrate the versatility of quasi-static scoping, we use it to build a module system and an object system. The former illustrates variable binding sharing among modules, the latter depicts code sharing among objects.

3.1 Modules

A module is a protection mechanism that realizes the principle of information hiding. Its operational detail is hidden from other modules. A module communicates with other modules only by importing and exporting variables. Hence, its extensional behavior is totally specified by its import and export variables.

In our module system, there are two kinds of first-class entities: *interfaces* and *clients*. An interface is a resolver that exports a collection of sharable variable bindings, which serve as communication channels between the clients that import them. A client is a quasi-static procedure with its quasi-static formals being the client's import variables. It gains access to sharable variables by *linking* with interfaces and interacts with other clients by writing to and reading from shared variables.

Besides implementing information hiding, our module system also supports incremental development and testing

```
(define user-intf
  (let ((sqrt (lambda (n)
                (error "square root function undefined"))))
    (mk-resolver (sqrt SQUARE-ROOT))))
```

Figure 6: User interface.

```
(define tester-intf
  (let ((tester
         (lambda (impl)
           ((impl (qs-lambda0
                    ((sqrt SQRT) (epsilon EPSILON)) ()
                    ...))))))
    (mk-resolver (tester TEST))))
(define installer-intf
  (let ((installer
         ((user-intf
           (qs-lambda0 ((sqrt SQUARE-ROOT)) ()
             (lambda (impl)
               ((impl (qs-lambda0 ((f SQRT)) ()
                        (set! sqrt f))))))))))
    (mk-resolver (installer INSTALL))))
(define impl-intf (superimpose tester-intf installer-intf))
```

Figure 7: Implementor interface.

of modules. As presented in the previous section, there are various ways of building interfaces (resolvers). Two interfaces can be combined into a single one and a sub-interface can be constructed out of an existing one. In addition, the definition, and therefore the compilation, of a client is independent of its linking with its interfaces. Thus, a client can be created without the existence of its interfaces. Later, it can be linked to its interfaces when they become available. Moreover, changing the definition of a client requires only the recompilation and relinking of the client, no other clients are affected. Our module system is thus highly attractive to the development and testing of programs in an interactive environment.

To illustrate, we show how to apply the module concept to manage the development of a square root function. In Figure 6 we define an interface *user-intf* that is intended for the function's users. It exports a single variable *sqrt*, whose implementation is yet to be defined, under the external name *SQUARE-ROOT*. Next we define in Figure 7 an interface *impl-intf* for the function's implementor. The interface is a combination of two independent interfaces: *tester-intf* and *installer-intf*. The interface *tester-intf* exports the variable *tester* with the external name *TEST*. The procedure *tester* serves as the specification that the implementation must meet. It takes an implementation *impl* and returns

```
(define test&install
  (impl-intf
    (qs-lambda0 ((test TEST) (install INSTALL)) (impl)
      (cond
        ((test impl) (install impl) 'installed)
        (else 'rejected)))))
(define newton-sqrt
  (let ()
    (define tolerance 0)
    (define iterate
      (lambda (x n)
        (if (< (abs (- (* x x) n)) tolerance)
            x
            (iterate (/ (+ x (/ n x)) 2.0) n))))
    (define sqrt
      (lambda (n)
        (iterate 1.0 n)))
    (mk-resolver (tolerance EPSILON) (sqrt SQRT))))
(test&install newton-sqrt)
```

Figure 8: A square root function implementation.

#t only if the implementation satisfies the unspecified tests. An implementation *impl* of the square root function is in the form of an interface. It must provide at least two export variables under the external names *SQRT* and *EPSILON*. The former is associated with the implementation of the square root function. The latter denotes the tolerance of the function's solutions. The tests are expected to tune the tolerance factor of the implementation to meet the needs of the users. The other interface *installer-intf* exports another variable *installer* whose external name is *INSTALL*. The procedure *installer* provides the implementor a means to deposit the final result in the *sqrt* variable that is visible to the users. It gains access to *sqrt* by importing *SQUARE-ROOT* from the user interface *user-intf*.

There are various ways of realizing the square root function. Figure 8 illustrates one possibility. First, testing and installation are merged into a single procedure called *test&install*. It takes an implementation and applies the imported testing procedure to it. It then uses the imported installation procedure to deposit the implementation into the variable visible to the users, provided the testing procedure approves the implementation. Otherwise, it issues a negative response. After the definition of *test&install*, the implementor need only come up with a solution in the form of an interface. For instance, Figure 8 shows a square root function implementation *newton-sqrt*, in the form of a two-variable interface, that is based on Newton's method. To test and install this implementation, the implementor need only invoke the *test&install* procedure on it. The square root function implementation can be altered with minimal recompilation and relinking. The only changes are to the implementation itself; no other clients or interfaces need be recompiled or relinked.

Our module system uses by-reference export and import, as well as run-time linking. Felleisen and Friedman's Scheme module system [7] also employs run-time linking. But it uses by-value import and export. There are thus serious restrictions on the ordering of linking, since import can occur only when the export value is readily available. To ease the restrictions, they switch to by-name import that essentially delays import until the value is needed. Unfortunately, by-name import only works for import values that are procedures.

Curtis and Rauen's Scheme module system [5] uses by-reference import and export. But its goal is to perform compile-time (static) linking since macros are expected to be sharable as well. Therefore their interfaces are compile-time objects, since they must be computable statically.

ML's module system [13] uses by-value import and export. But since ML has first-class references (variable bindings), by-reference import and export are easily expressible as well. ML's functors, which are functions mapping modules to modules, are not first-class values; instead, they are first-order objects. With such a restriction, every functor application can be *inlined* and hence ML's import and export variables can be statically linked as in Curtis and Rauen's system. Without first-class functors, however, procedures like *tester* and *installer* of Figure 7 are not expressible in the two module systems.

In summary, we believe that by-reference import and export and run-time linking are essential to a Scheme module system. But we would also like to have static linking because of macro sharing and efficiency. Thus, a combination of the above systems is preferred.

3.2 Objects

In Scheme object systems [2, 15] where instance variables are lexical variables, inheritance of instance variables is not feasible. One way of getting around this restriction is to define for each instance variable a pair of methods called reader and writer, and simulate instance variable inheritance with method inheritance, as in CLOS [18]. Another constraint posed by lexical instance variables is that an object's method, which is a procedure, must be defined within the lexical scope of the object's instance variables in order for it to gain access to the instance variables. Oaklisp [11] removes the constraint by resorting to the **add-method** special form

(add-method (*op* (*type* . *inst-vars*) . *args*) . *body*)

where *inst-vars* is the list of instance variables of the class *type* that are accessible to the method's body.

The addition of quasi-static scoping, however, provides a simple alternative. By treating a method as a procedure

```
(define mk-obj vector)
(define obj→dispatcher (lambda (x) (vector-ref x 0)))
(define obj→ivars (lambda (x) (vector-ref x 1)))
(define obj→parent (lambda (x) (vector-ref x 2)))
(define obj→mcell (lambda (x) (vector-ref x 3)))
(define obj→template (lambda (x) (vector-ref x 4)))
(define mk-cell vector)
(define cell-ref (lambda (x) (vector-ref x 0)))
(define cell-set! (lambda (x v) (vector-set! x 0 v)))
```

Figure 9: Object representation.

with its quasi-static formal parameters as the instance variables referenced in the method, the definition of an object's method can be separated from the object's instance variables. Consequently, it facilitates incremental method addition to an object *a la* Oaklisp. For example, the following quasi-static procedure

```
(define distance
  (qs-lambda0 ((x X) (y Y)) (self)
    (+ x y)))
```

is a method that references instance variables whose external names are X and Y. Such a method is called an *open method*. It can be added to the classes the user deems appropriate.

Furthermore, by packaging an object's instance variables into a resolver and making it available to the object's children, we have instance variable inheritance as well. To illustrate, let *cp* be an object with at least two instance variables named X and Y, and let their resolver be defined as

```
(define cp-ivars
  (let ((x 3) (y 4) ...)
    (mk-resolver (x X) (y Y) ...)))
```

Moreover, let *mp* be an object that inherits *cp-ivars* from *cp*. Then, to invoke *distance* on *mp*, the open method's quasi-static formals are resolved by *cp-ivars* to produce an *effective method* as follows:

```
(define resolved-distance (cp-ivars distance))
```

It is then invoked on *mp*, which is bound to the static formal parameter *self*,

```
(resolved-distance mp)
```

In the rest of this section we describe a simple object system that is based on quasi-static scoping. The object system employs message passing, has only single inheritance, and achieves self-reference through an explicit parameter *self*. It is based on delegation. There are no specific objects designated as class objects; every object is its own class object. An "instance" object is created from a "class" object by cloning the latter.

In Figure 9 we define our system's object representation. An object is a vector of five elements: a dispatcher, an instance variable resolver, a parent object, a cell of open methods, and an instance variable template. The dispatcher defines how the object responds to messages. The instance variable resolver defines the instance variables to which the object has access, including the ones that the object inherits from its parent. The fourth element is a cell that contains an incrementally growing association list of operation and open method pairs. For ease of presentation, operations are just symbols. The last component of an object is a template from which the instance variable resolver is generated. Essentially, clones of an object share the same open method cell. Code sharing, a single method operating on multiple sets of instance variables, is accomplished through the sharing of open methods among objects.

An object is created by invoking the *new* procedure of Figure 10 with an instance variable template, a parent object, and the initial values of the object's instance variables. Each new object has two built-in methods *add-method* and *clone*. The former is responsible for the incremental addition of methods to the object. It takes an operation and open method pair, and adds it to the object's open method cell. The latter built-in method clones the object itself. It is similar to *new* except that it needs only the initial values of the clone's instance variables, since the dispatcher, the parent object, and the instance variable template are known to be those of the cloned object.

Instance variable inheritance is defined by the *mk-ivars* procedure of Figure 10. It uses the object's instance variable template with initial values to generate the object's *own* instance variable resolver. This resolver is then composed with that of the parent object. Hence, when the composition is applied to a given open method, the method gains access to both the object's own instance variables and the ones of the parent objects. In case instance variable inheritance is undesirable, we can replace the procedure *mk-ivars* of Figure 10 by

```
(define mk-ivars
  (lambda (template ivals parent)
    (apply template ivals)))
```

This time the parent object's instance variable resolver is ignored and only the object's resolver is used to resolve an open method. Hence, the parent's instance variables are not visible to the method.

The procedure *obj-maker* of Figure 10 is used by both *new* and *clone* to create an object. Its most noticeable task is to create the object's dispatcher. Upon receiving a message *op*, which is an operation, an object's dispatcher searches the open methods in its cell. If one is found, it is passed along with the object itself to the success continuation *sk*, which handles the rest of a method call. Once an open method is located for a message, we apply the instance variable resolver of the handler object, the object in which the open method is found, to yield the final effective method. The effective method is then called with the actual arguments. The procedure *send* of Figure 10 is an abstraction of such a process.

```
(define new
  (lambda (template parent . ivals)
    (let ((mcell (mk-cell built-in-methods))
          (ivars (mk-ivars template ivals parent)))
      (obj-maker ivars parent mcell template))))
(define add-method
  (qs-lambda0 (self op method)
    (let ((mcell (obj→mcell self)))
      (cell-set! mcell
        (cons (cons op method) (cell-ref mcell))))))
(define clone
  (qs-lambda0 (self . ivals)
    (let ((parent (obj→parent self))
          (template (obj→template self))
          (mcell (obj→mcell self)))
      (let ((ivars (mk-ivars template ivals parent)))
        (obj-maker ivars parent mcell template)))))
(define built-in-methods
  (list (cons 'add-method add-method)
        (cons 'clone clone)))
(define mk-ivars
  (lambda (template ivals parent)
    (let ((ivars (apply template ivals)))
      (lambda (method)
        ((obj→ivars parent) (ivars method))))))
(define obj-maker
  (lambda (ivars parent mcell template)
    (let ((parent-dispatcher (obj→dispatcher parent)))
      (letrec
        ((obj (mk-obj
                (lambda (op sk fk)
                  (cond
                    ((assq op (cell-ref mcell))
                     => (lambda (p) (sk (cdr p) obj)))
                    (else (parent-dispatcher op sk fk))))
                ivars parent mcell template)))
        obj))))
(define send
  (lambda (receiver op . args)
    ((obj→dispatcher receiver) op
      (lambda (method handler)
        (apply (resolve-method receiver handler method)
          receiver args))
      (lambda () (method-call-error op)))))
(define resolve-method
  (lambda (receiver handler method)
    ((obj→ivars handler) method)))
```

Figure 10: A simple object system.

```
(define base-object
  (mk-obj
    (lambda (op sk fk) (fk))
    (lambda (x) x)
    'no-parent
    (mk-cell '())
    (lambda args (mk-resolver))))
```

Figure 11: The base object.

```
(define cartesian-distance
  (qs-lambda0 ((x X) (y Y)) (self)
    (sqrt (+ (* x x) (* y y)))))
(define manhattan-distance
  (qs-lambda0 ((x X) (y Y)) (self)
    (+ x y)))
(define cp1
  (new (lambda (x y) (mk-resolver (x X) (y Y)))
       base-object 3 4))
(define cp2 (send cp1 'clone 1 2))
(send cp2 'add-method 'distance cartesian-distance)
(define mp (new (lambda args (mk-resolver)) cp1))
(send mp 'add-method 'distance manhattan-distance)
```

Figure 12: Cartesian and Manhattan points.

On the other hand, if an object cannot handle a given message, it is delegated to the object's parent. Eventually, if no object in the inheritance chain is able to respond to the message, the dispatcher of the ultimate parent object, *base-object* of Figure 11, is used to activate the failure continuation *fk*.

The procedure *resolve-method* used in *send* implements *early binding* for the instance variables. That is, the effective method assumes its instance variables from the handler object. The following alternative implements *late binding* in which the effective method gets its instance variables from the receiver object, the object initially receiving the message of a method call.

```
(define resolve-method
  (lambda (receiver handler method)
    ((obj→ivars receiver) method)))
```

Finally, we use the example in Figure 12 to demonstrate incremental method addition and instance variable inheritance at work. First we define two open methods *cartesian-distance* and *manhattan-distance* that both reference instance variables X and Y. The former computes the Cartesian distance of a two-dimensional point from the origin, whereas the latter computes the Manhattan distance. Next, we use *new* to define a two-dimensional point object *cp1*

whose coordinates are (3,4). The object has no parent object. It has two instance variables called X and Y externally, as defined by the instance variable template (**lambda** (x y) (**mk-resolver** (x X) (y Y))). When this object is created it has no other methods besides the two built-ins *add-method* and *clone*. We invoke the latter to create a clone *cp2* of *cp1*. Next, we add the *cartesian-distance* open method to *cp2*. Since both *cp1* and *cp2* share the same open method cell, *cartesian-distance* is available to *cp1* as well. Then, we decide to view *cp1* as a Manhattan point object. Hence, we define an object *mp* that inherits from *cp1*. This new object inherits all the methods and instance variables of *cp1*. But the *cartesian-distance* method is inappropriate for *mp*. Thus, we add a new method *manhattan-distance* to *mp*, shadowing the version associated with *cp1*. So, invoking the distance method on *mp* uses *manhattan-distance* to operate on the instance variables of *cp1*, showing the effect of instance variable inheritance.

4 Formal Semantics

We present in Figure 13 a denotational description of a simplified Scheme with quasi-static scoping. Besides **qs-lambda** and **resolve1**, we also include variable reference, variable assignment, and procedure invocation. In order to simplify the presentation, we only describe single-argument procedures. Furthermore, we treat an ordinary Scheme procedure as a quasi-static procedure with no quasi-static formal parameters. That is, in this section (**lambda** (x) e) is considered equivalent to (**qs-lambda0** () (x) e), which in turn is equivalent to (**qs-lambda** () () (x) e).

4.1 Finite Functions

We summarize the finite function definitions employed in the semantics. Let $f : A \multimap B$ denote a finite function from A to B. We write $b/a \in f$ if $f(a) = b$ and represent f by the finite set $\{b/a \mid b/a \in f\}$. The *arid* finite function is denoted by $\emptyset$.

Let $f : A \multimap B$, then

- $Dom(f) = \{a \in A \mid \exists b \in B, b/a \in f\}$ is the *domain* of f,
- $Rng(f) = \{b \in B \mid \exists a \in A, b/a \in f\}$ is the *range* of f.

Let $f, f', f'' : A \multimap B$, $Dom(f) \cap Dom(f') = \emptyset$, $A' \subseteq A$, $B' \subseteq B$, $a \in A$, $b \in B$, and $g : B \multimap C$, then

- $f \oplus f' = \{b/a \mid b/a \in f \text{ or } b/a \in f'\}$ is the *sum* of f and f',
- $f \backslash A' = \{b/a \in f \mid a \notin A'\}$ is the *domain omission* of f over A',
- $f \backslash\backslash B' = \{b/a \in f \mid b \notin B'\}$ is the *range omission* of f over B',
- $f[b/a] = (f \backslash \{a\}) \oplus \{b/a\}$ is the *extension* of f with a mapped to b,
- $f[f''] = (f \backslash Dom(f'')) \oplus f''$ is the *extension* of f with f'',
- $g \circ f = \{c/a \mid b/a \in f, c/b \in g\}$ is the *composition* of f and g.

The operation $\oplus$ is commutative and associative; the operation $\circ$ is associative; the operations [ ], \, and \\ are left associative. The descending precedence order is [ ], \, \\, $\circ$, and $\oplus$.

4.2 Evaluation Function

In addition to an expression e, the evaluation function $[\![\,]\!]$ takes three more arguments ω, ρ, and σ, each of which is a finite function. The *link* ω maps variables (internal names) to external names. It records the quasi-static variables declared in the enclosing context of e. The *environment* ρ maps variables to locations. It defines the resolved variables, both static and quasi-static, that are accessible to e. Not shown in the semantic clauses for variable reference, assignment, and quasi-static variable resolution is that the variable in question must be defined in the environment: $x \in Dom(\rho)$. The domains of ω and ρ are not necessarily disjoint. A variable that is defined in both ω and ρ is a quasi-static variable that has been resolved prior to the evaluation of e. The *store* σ maps locations to values. It is included here to model variable assignments.

A quasi-static procedure is denoted by a triple $\langle c, \omega_u, \rho_r \rangle$ of a piece of code, a link, and an environment. The link ω_u specifies the procedure's unresolved quasi-static formals. The environment ρ_r defines the procedure's resolved quasi-static formals. The domains of ρ_r and ω_u are always disjoint, since a quasi-static formal parameter is either resolved or unresolved. The code c is the denotation of the procedure's body parameterized over the resolved quasi-static formals, as well as the value of the procedure's static formal parameter.

When a procedure $\langle c, \omega_u, \rho_r \rangle$ is invoked, the ρ_r component, which defines the resolved quasi-static formals, is passed to c. The link ω_u, however, is discarded, since any unresolved quasi-static formals are no longer resolvable in the body. See the procedure invocation clause in Figure 13.

4.2.1 Resolving an External Name

Let the expression e in the evaluation of a **resolve1** expression

$$[\![(\textbf{resolve1}\ x\ Q\ e)]\!]\,\omega\,\rho\,\sigma$$

denote the quasi-static procedure $\langle c, \omega_u, \rho_r \rangle$. Then, in the resulting quasi-static procedure, the quasi-static formals in ω_u that are associated with the external name Q are resolved to the location $(\rho\, x)$. The link ω_u is composed with the function $\{(\rho\, x)/Q\}$, yielding $\{(\rho\, x)/Q\} \circ \omega_u = \{(\rho\, x)/y \mid Q/y \in \omega_u\}$. The result is a finite function from variables to locations, *i.e.*, an environment. This environment defines the quasi-static formals in ω_u that are resolved by $\{(\rho\, x)/Q\}$. It is combined with the environment ρ_r to form the new procedure's environment component, $\{(\rho\, x)/Q\} \circ \omega_u \oplus \rho_r$. Since the resolved quasi-static formals are now associated with locations, they should be removed from the link. Hence, the

Abstract Syntax:

$$\begin{array}{rcll} x, y, z & : & \text{Var} & \text{(Internal Names or Variables)} \\ Q & : & \text{Nam} & \text{(External Names)} \\ e & : & \text{Exp} & \text{(Expressions)} \end{array}$$

$$\begin{aligned} e \quad ::= \quad & x \mid (\textbf{set!}\ x\ e) \mid (e\ e) \mid (\textbf{resolve1}\ x\ Q\ e) \\ & \mid (\textbf{qs-lambda}\ ((y_1\ z_1)\cdots(y_m\ z_m))\ ((x_1\ Q_1)\cdots(x_k\ Q_k))\ (x)\ e) \end{aligned}$$

Semantic Domains:

$$\begin{array}{rcll} l & : & \text{Loc} & \text{(Locations)} \\ \rho & : & \text{Env} = \text{Var} \multimap \text{Loc} & \text{(Environments)} \\ \pi & : & \text{Inh} = \text{Var} \multimap \text{Var} & \text{(Inheritors)} \\ \omega & : & \text{Lnk} = \text{Var} \multimap \text{Nam} & \text{(Links)} \\ \sigma & : & \text{Sto} = \text{Loc} \multimap \text{Val} & \text{(Stores)} \\ c & : & \text{Cod} = \text{Val} \longrightarrow \text{Env} \longrightarrow \text{Sto} \longrightarrow (\text{Val} \times \text{Sto}) & \text{(Codes)} \\ & & \text{Proc} = \text{Cod} \times \text{Lnk} \times \text{Env} & \text{(Procedures)} \\ v & : & \text{Val} = \text{Proc} + \cdots & \text{(Values)} \end{array}$$

Evaluation Function $[\![\ ]\!] : \text{Exp} \longrightarrow \text{Lnk} \longrightarrow \text{Env} \longrightarrow \text{Sto} \longrightarrow (\text{Val} \times \text{Sto})$

$$[\![x]\!]\,\omega\,\rho\,\sigma = \langle(\sigma\,(\rho\,x)), \sigma\rangle$$

$$[\![(\textbf{set!}\ x\ e)]\!]\,\omega\,\rho\,\sigma = \textbf{let}\ \langle v, \sigma'\rangle = [\![e]\!]\,\omega\,\rho\,\sigma\ \textbf{in}\ \langle v, (\sigma'[v/(\rho\,x)])\rangle$$

$$[\![(e_p\ e_a)]\!]\,\omega\,\rho\,\sigma = \textbf{let}\ \langle\langle c, \omega_u, \rho_r\rangle, \sigma'\rangle = [\![e_p]\!]\,\omega\,\rho\,\sigma\ \textbf{in}\ \textbf{let}\ \langle v, \sigma''\rangle = [\![e_a]\!]\,\omega\,\rho\,\sigma'\ \textbf{in}\ c\,v\,\rho_r\,\sigma''$$

$$[\![(\textbf{resolve1}\ x\ Q\ e)]\!]\,\omega\,\rho\,\sigma = \textbf{let}\ \langle\langle c, \omega_u, \rho_r\rangle, \sigma'\rangle = [\![e]\!]\,\omega\,\rho\,\sigma\ \textbf{in}\ \langle\langle c, \omega_u \backslash\backslash \{Q\}, \{(\rho\,x)/Q\} \circ \omega_u \oplus \rho_r\rangle, \sigma'\rangle$$

$$\begin{aligned} & [\![(\textbf{lambda}\ (x)\ e)]\!]\,\omega\,\rho\,\sigma = \\ & \quad \textbf{let}\ \omega_d = \omega \backslash \{x\}, \\ & \quad\quad \rho_f = \rho \backslash \{x\}\ \textbf{in} \\ & \quad \langle\langle(\lambda v \rho_r' \sigma' .\, [\![e]\!]\,\omega_d\,((\rho_r' \oplus \rho_f)[l/x])\,(\sigma'[v/l])), \emptyset, \emptyset\rangle, \sigma\rangle \\ & \quad\quad \textbf{where}\ l \notin \mathit{Dom}(\sigma') \end{aligned}$$

$$\begin{aligned} & [\![(\textbf{qs-lambda0}\ ((x_1\ Q_1)\cdots(x_k\ Q_k))\ (x)\ e)]\!]\,\omega\,\rho\,\sigma = \\ & \quad \textbf{let}\ \omega_u = \{Q_1/x_1, \ldots, Q_k/x_k\}\ \textbf{in} \\ & \quad\quad \textbf{let}\ \omega_d = \omega \backslash \{x\}[\omega_u], \\ & \quad\quad\quad \rho_f = \rho \backslash \mathit{Dom}(\omega_u) \backslash \{x\}\ \textbf{in} \\ & \quad\quad \langle\langle(\lambda v \rho_r' \sigma' .\, [\![e]\!]\,\omega_d\,((\rho_r' \oplus \rho_f)[l/x])\,(\sigma'[v/l])), \omega_u, \emptyset\rangle, \sigma\rangle \\ & \quad\quad\quad \textbf{where}\ l \notin \mathit{Dom}(\sigma') \end{aligned}$$

$$\begin{aligned} & [\![(\textbf{qs-lambda}\ ((y_1\ z_1)\cdots(y_m\ z_m))\ ((x_1\ Q_1)\cdots(x_k\ Q_k))\ (x)\ e)]\!]\,\omega\,\rho\,\sigma = \\ & \quad \textbf{let}\ \omega_o = \{Q_1/x_1, \ldots, Q_k/x_k\}, \\ & \quad\quad \pi = \{z_1/y_1, \ldots, z_m/y_m\}\ \textbf{in} \\ & \quad\quad \textbf{let}\ \omega_\pi = \omega \circ \pi, \\ & \quad\quad\quad \rho_r = \rho \circ \pi\ \textbf{in} \\ & \quad\quad\quad \textbf{let}\ \omega_u = \omega_\pi \backslash \mathit{Dom}(\rho_r) \oplus \omega_o, \\ & \quad\quad\quad\quad \omega_d = \omega \backslash \{x\}[\omega_\pi \oplus \omega_o], \\ & \quad\quad\quad\quad \rho_f = \rho \backslash \mathit{Dom}(\pi) \backslash \mathit{Dom}(\omega_o) \backslash \{x\}\ \textbf{in} \\ & \quad\quad\quad \langle\langle(\lambda v \rho_r' \sigma' .\, [\![e]\!]\,\omega_d\,((\rho_r' \oplus \rho_f)[l/x])\,(\sigma'[v/l])), \omega_u, \rho_r\rangle, \sigma\rangle \\ & \quad\quad\quad\quad \textbf{where}\ l \notin \mathit{Dom}(\sigma') \end{aligned}$$

Figure 13: Denotational semantics of Scheme with quasi-static scoping.

new procedure's link component is $\omega_u \backslash\backslash \{Q\} = \{Q'/x \in \omega_u \mid Q' \not\equiv Q\}$. Thus, in the resulting triple, the domains of the link and the environment remain disjoint. The code c, however, is carried over to the new procedure unchanged. Hence, the two procedures *share* the same piece of code but operate on different sets of bindings for the quasi-static formal parameters.

4.2.2 Deriving Qs-lambda

We develop the meaning of a **qs-lambda** expression in three stages. First we describe **lambda**, which is **qs-lambda** without quasi-static formal parameters. In the second stage we generalize **lambda** to **qs-lambda0**, which has quasi-static formal parameters but no lexical inheritance of quasi-static variables. Finally we include quasi-static variable inheritance to get **qs-lambda**. See Figure 13 in which, for explication purposes, we also include the clauses for **lambda** and **qs-lambda0**.

The meaning of a **lambda** expression

$$[\![(\textbf{lambda}\ (x)\ e)]\!]\,\omega\,\rho\,\sigma$$

is defined as follows:

$$\begin{array}{l}\textbf{let}\ \omega_d = \omega\backslash\{x\},\\ \quad \rho_f = \rho\backslash\{x\}\ \textbf{in}\\ \langle\langle(\lambda v \rho'_r \sigma' \,.\, [\![e]\!]\,\omega_d((\rho'_r \oplus \rho_f)[l/x])(\sigma'[v/l])), \emptyset, \emptyset\rangle, \sigma\rangle\\ \quad \text{where } l \notin Dom(\sigma')\end{array}$$

Since there are no quasi-static formals, the resulting triple's link and environment components are both arid. The quasi-static variables declared in the enclosing context of the body expression e are $\omega_d = \omega\backslash\{x\}$. They are the ones declared in the enclosing context of the **lambda** expression that are not shadowed by the static formal parameter x. The resolved free variables, both static and quasi-static, visible to the body expression e prior to the evaluation of the **lambda** expression are defined by the environment $\rho_f = \rho\backslash\{x\}$. It is the procedure's evaluation time environment ρ but with the static formal parameter x omitted. The resolved free variables visible to e during its evaluation are the variables defined in ρ_f, the resolved quasi-static formal parameters of ρ'_r, and a fresh binding of the static formal parameter x: $(\rho'_r \oplus \rho_f)[l/x]$.

But when the code is invoked, the parameter ρ'_r is certain to be associated with $\emptyset$, since a **lambda** expression has no quasi-static formals. Also, by the definition of finite function extension, $(\rho\backslash\{x\})[l/x])$ is equivalent to $\rho[l/x]$. Therefore, the triple's code is equivalent to

$$(\lambda v \rho'_r \sigma' \,.\, [\![e]\!]\,(\omega\backslash\{x\})\,(\rho[l/x])\,(\sigma'[v/l]))$$

which, except for the extra link $\omega\backslash\{x\}$ and the ignored parameter ρ'_r, is what we have for a procedure in Scheme.

Next we define the meaning of a **qs-lambda0** expression

$$[\![(\textbf{qs-lambda0}\ ((x_1\ Q_1)\cdots(x_k\ Q_k))\ (x)\ e)]\!]\,\omega\,\rho\,\sigma$$

as follows:

$$\begin{array}{l}\textbf{let}\ \omega_u = \{Q_1/x_1, \ldots, Q_k/x_k\}\ \textbf{in}\\ \quad \textbf{let}\ \omega_d = \omega\backslash\{x\}[\omega_u],\\ \qquad \rho_f = \rho\backslash Dom(\omega_u)\backslash\{x\}\ \textbf{in}\\ \qquad \langle\langle(\lambda v \rho'_r \sigma' \,.\, [\![e]\!]\,\omega_d((\rho'_r \oplus \rho_f)[l/x])(\sigma'[v/l])), \omega_u, \emptyset\rangle, \sigma\rangle\\ \qquad\quad \text{where } l \notin Dom(\sigma')\end{array}$$

The resulting triple's link component ω_u specifies the unresolved quasi-static formals $x_1, \ldots, x_k$. Since there is no lexical inheritance, the triple's environment component is arid. The code component is the same as that of a **lambda** expression, except that ω_d and ρ_f take on different meanings. The link $\omega_d = \omega\backslash\{x\}[\omega_u]$ captures the quasi-static variables declared with respect to the body expression e. It consists of the most recently specified ones and the ones declared in the enclosing context of the **qs-lambda0** expression that are not shadowed by the static formal parameter. The resolved free variables available to the body expression when the **qs-lambda0** expression is evaluated are defined by the environment $\rho_f = \rho\backslash Dom(\omega_u)\backslash\{x\}$. They are the ones resolved before the evaluation of the **qs-lambda0** expression that are not shadowed by the newly specified quasi-static variables or the static formal parameter.

In order to verify that **lambda** is a degenerate case of **qs-lambda0**, we need only replace ω_u by $\emptyset$.

Then, let ψ be $((y_1\ z_1)\cdots(y_m\ z_m))$ and ϕ be $((x_1\ Q_1)\cdots(x_k\ Q_k))$, we define the meaning of a **qs-lambda** expression

$$[\![(\textbf{qs-lambda}\ \psi\ \phi\ (x)\ e)]\!]\,\omega\,\rho\,\sigma$$

as follows:

$$\begin{array}{l}\textbf{let}\ \omega_o = \{Q_1/x_1, \ldots, Q_k/x_k\},\\ \quad \pi = \{z_1/y_1, \ldots, z_m/y_m\}\ \textbf{in}\\ \quad \textbf{let}\ \omega_\pi = \omega \circ \pi,\\ \qquad \rho_r = \rho \circ \pi\ \textbf{in}\\ \qquad \textbf{let}\ \omega_u = \omega_\pi\backslash Dom(\rho_r) \oplus \omega_o,\\ \qquad\quad \omega_d = \omega\backslash\{x\}[\omega_\pi \oplus \omega_o],\\ \qquad\quad \rho_f = \rho\backslash Dom(\pi)\backslash Dom(\omega_o)\backslash\{x\}\ \textbf{in}\\ \qquad\quad \langle\langle(\lambda v \rho'_r \sigma' \,.\, [\![e]\!]\,\omega_d((\rho'_r \oplus \rho_f)[l/x])(\sigma'[v/l])), \omega_u, \rho_r\rangle, \sigma\rangle\\ \qquad\qquad \text{where } l \notin Dom(\sigma')\end{array}$$

This yields a quasi-static procedure $\langle c, \omega_u, \rho_r\rangle$ whose components are defined in detail below.

The syntax ψ denotes the *inheritor* finite function $\pi = \{z_1/y_1, \ldots, z_m/y_m\}$ and the syntax ϕ denotes the link $\omega_o = \{Q_1/x_1, \ldots, Q_k/x_k\}$. Not shown in the semantic clause is that the inherited quasi-static variables $z_1, \ldots, z_m$ must be declared in the enclosing context. That is, it is a syntax error unless $Rng(\pi) \subseteq Dom(\omega)$. The inheriting quasi-static formals $y_1, \ldots, y_m$ have the same external names as $z_1, \ldots, z_m$, respectively. Hence, their link is $\omega_\pi = \omega \circ \pi = \{Q/y \mid z/y \in \pi, Q/z \in \omega\}$. The inherited quasi-static variables that have been resolved prior to the **qs-lambda** expression's evaluation are defined in ρ. So, the environment of the resolved inheriting quasi-static formals is $\rho_r = \rho \circ \pi = \{l/y \mid z/y \in$

$\pi, l/z \in \rho\}$, which is the third component of the resulting triple. Consequently, the link of the unresolved inheriting quasi-static formals is $\omega_\pi \backslash Dom(\rho_r)$. The link component ω_u of the procedure's unresolved quasi-static formal parameters is therefore $\omega_o \oplus \omega_\pi \backslash Dom(\rho_r)$, the sum of ω_o and $\omega_\pi \backslash Dom(\rho_\pi)$.

The procedure's code component is the same as that of **qs-lambda0**, except for the different interpretations of ω_d and ρ_f. The link ω_d of the declared quasi-static variables that are visible to e is $\omega \backslash \{x\}[\omega_\pi \oplus \omega_o]$. They are the variables specified by the procedure and the variables declared in the enclosing context that are not shadowed by the static formal parameter. The parameter ρ_r' is the procedure's resolved quasi-static formals ρ_r when the code is invoked. The environment ρ_f of the resolved free variables known to the body expression prior to the **qs-lambda** expression's evaluation is $\rho \backslash Dom(\pi) \backslash Dom(\omega_o) \backslash \{x\}$. It is the variables defined in ρ that are not shadowed by the inheriting quasi-static formals $y_1, \ldots, y_m$, the quasi-static formals $x_1, \ldots, x_k$, or the static formal parameter x.

Again, by substituting $\emptyset$ for π, it is straightforward to verify that **qs-lambda0** is a degenerate case of **qs-lambda**.

5 Implementation

We describe an implementation of quasi-static scoping. It includes the representation of quasi-static procedures, the run-time support for procedure invocations and quasi-static variable references, and the quasi-static variable resolution operation. Since we have designed quasi-static scoping to be as orthogonal to Scheme as possible, we have also made its implementation as independent of Scheme implementations as possible.

We represent a quasi-static closure (procedure)

$$\begin{array}{l}(\textbf{qs-lambda}\ ((iq_1\ oq_1) \ldots (iq_m\ oq_m)) \\ \qquad\qquad ((q_1\ Q_1) \ldots (q_k\ Q_k)) \\ \qquad\qquad \textit{formals body})\end{array} \quad (1)$$

by a *frame* of $1+m+k+h$ consecutive memory slots arranged as follows:

$$\langle c, s_1, \ldots, s_m, s_{m+1}, \ldots, s_{m+k}, f_1, \ldots, f_h \rangle \quad (2)$$

The first slot c is the location of a *static closure* representing the ordinary Scheme procedure (**lambda** *formals body*). Each of the next $m+k$ slots contains either a location or an external name. The first m slots are for the inheriting quasi-static formals $iq_1, \ldots, iq_m$. The next k slots are for the local quasi-static formals $q_1, \ldots, q_k$. The last h slots $f_1, \ldots, f_h$ are locations of the enclosing frames. They are the *display* [6] of the procedure's free quasi-static variable references. We discuss how the slots are filled later.

The values of m and k are readily available from the syntax of the **qs-lambda** expression itself. The value of h depends on the syntactic context of the **qs-lambda** expression. It is the number of enclosing **qs-lambda** expressions. Hence, each frame's shape can be determined statically.

Every static closure contains a pointer to a frame through which the procedure's body gains access to the quasi-static variables visible to it. For a static closure created by a **lambda** expression, this frame is the most recently activated one when the closure is built. For a static closure constructed by a **qs-lambda** expression, it is the newly created frame. That is, a frame's static closure points back circularly to the frame itself. This circularity is justifiable since, when a frame is invoked, the accessible quasi-static variables are defined in the frame.

The additional run-time support then consists of a single register called the Frame Pointer register (FP) that points to the most recently activated frame. Context switching is simple. The activation record of each procedure invocation maintains the caller's FP value. Returning from a procedure invocation simply reinstalls FP to that value. Invoking a procedure, static or quasi-static, requires only saving the current value of FP in the newly created activation record before adjusting it to point to the callee's frame. In case the call is in tail position, however, the callee's activation record assumes the same FP value as that of the caller's.

We next explain how a quasi-static variable reference is achieved. We assign to each quasi-static variable reference occurrence a pair of statically determined numbers (d, i) called the *lexical address*. The number $d \geq 0$ is the *depth* between a reference occurrence and its binding occurrence, *i.e.*, the number of **qs-lambda** expressions between the two occurrences. The other number i is the binding occurrence's slot index in its frame. Using the notation $[l]$ to denote the contents of the location l, the value of a zero-depth reference occurrence, $(0, i)$, is denoted by $[[[\mathrm{FP}] + i]]$. The current frame's location is denoted by [FP]. Adding the index i to it, $[\mathrm{FP}] + i$, gives us the location of the quasi-static variable's slot. Dereferencing it, $[[\mathrm{FP}] + i]$, yields the contents of the quasi-static variable's slot. If it contains a location, dereferencing it, $[[[\mathrm{FP}] + i]]$, returns the quasi-static variable's value. Otherwise, since an external name is *not* a location, an attempt to dereference it raises an unbound variable error.

The value of a reference occurrence with a lexical address of (d, i), where $d \geq 1$, requires one more level of indirection. It is denoted by $[[[[\mathrm{FP}] + t] + i]]$ where the statically determined number t is $d+m+k$, with $m+k$ being the number of quasi-static formals in the current frame. Hence $[[\mathrm{FP}] + t]$ denotes the frame in which the binding occurrence's slot resides. Besides this extra indirection, the rest is the same as a zero-depth reference.

The two kinds of quasi-static variable references described above both refer to the variable's *r-value*. The *l-values*, which are needed later, can be obtained as in the r-value cases except for the last dereferencing. That is, $[[\mathrm{FP}] + i]$ if

the reference occurrence's depth is zero and $[[[\mathrm{FP}]+t]+i]$ otherwise.

There are two ways to create a new quasi-static frame. First is the evaluation of a **qs-lambda** expression of (1) that yields a frame of (2). The location c of the static closure fills the first slot. The next m slots are filled with the l-values of the inherited quasi-static variables $oq_1, \ldots, oq_m$, because iq_i inherits the location or external name (not value) of oq_i. Since $q_1, \ldots, q_k$ are unresolved, their slots are filled with the external names $Q_1, \ldots, Q_k$, respectively. That is, $s_{m+i} = Q_i$ for $i = 1, \ldots, k$. The display's frame pointers $f_1, \ldots, f_h$ get their values from the current frame pointed to by FP. The frame pointer f_1 points to the closest frame when the new frame is invoked (*i.e.*, [FP]). The other frame pointers $f_2, \ldots, f_h$ are the display of the current frame. Since the current frame's shape is known at compilation, its slot indices are known. Therefore it is straightforward to generate the code needed to copy the display.

We can apply two compile-time optimizations at this stage. First, the frame need not allocate a slot for any of the quasi-static formals $iq_1, \ldots, iq_m$ or $q_1, \ldots, q_k$ that is not referenced in the body. Second, the display need only include the frames that are actually needed in the body. That is, if none of the quasi-static variables of an outer frame is referenced in the body, it can be excluded from the display.

The other way to create a new frame is by evaluating a **resolve** expression

$$(\textbf{resolve}\ ((x_1\ Q_1)\ \ldots\ (x_n\ Q_n))\ \textit{qs-proc})$$

First we check that the variables $x_1, \ldots, x_n$ are resolved, *i.e.*, associated with locations. For each x_i that is a quasi-static variable, we must make sure that its l-value is a location, not an external name. Furthermore, the expression *qs-proc* must evaluate to a frame

$$\langle c, s_1, \ldots, s_{m+k}, f_1, \ldots, f_h \rangle$$

Let l_i be the location associated with the variable x_i. Then the **resolve** expression returns a new frame

$$\langle c, s'_1, \ldots, s'_{m+k}, f_1, \ldots, f_h \rangle$$

where for $i = 1, \ldots, m+k$,

$$s'_i = \begin{cases} l_j & \text{if } s_i \equiv Q_j \text{ for some } n \geq j \geq 1 \\ s_i & \text{otherwise.} \end{cases}$$

That is, each slot s_i is compared against the external names $Q_1, \ldots, Q_n$. If there is a match with Q_j, the corresponding slot s'_i is the location l_j. Otherwise, s_i is either a location or a different external name. Hence, s'_i is the same as s_i.

We should point out that without sufficient dataflow information, each lexical variable x_i must be assumed to be assignable. As a result, optimizations performed by many Scheme compilers on a lexical variable that is free of side-effects are not applicable to x_i.

Finally we sketch a simple variation of the implementation presented above. The purpose of a frame's display is to gain access to the slots of the procedure's free quasi-static variables. Hence, instead of copying a sequence of frame pointers, we could copy down the slots of the free quasi-static variables. The result is reminiscent of the way Cardelli's Functional Abstract Machine [3] handles a procedure's free lexical variable references. It would require potentially more time and space to build a frame. On the other hand, it reduces every quasi-static variable reference's lexical address to a depth of zero. Hence, it speeds up free quasi-static variable referencings.

6 Comparisons and Future Work

We accomplish variable sharing across lexical scopes with resolvers that selectively "export" variable bindings and quasi-static procedures that selectively "import" variable bindings. Resolvers are run-time linking operators and quasi-static procedures are compiled but only partially linked programs.

MIT Scheme's first-class environments [1, 12] resemble our resolvers. They differ from our quasi-static scoping in four aspects. First, there is no mechanism like **resolve** that can export variables selectively. Consequently, in the presence of first-class environments, optimization techniques such as constant folding that involve eliminating variables are no longer meaning preserving transformations. Second, first-class environments rely on the internal names of variables. Hence, α-conversion is invalid on some variables. This imposes a serious problem on macro systems since generated variable names could cause inadvertent capturing. Third, *eval*, the user-accessible interpreter, compiles source code at run time and the same piece of source code is compiled as many times as it is used. Thus, the performance is inferior to our approach in which each piece of source is compiled only once. Fourth, environment is a notion that is not employed by every computational model. Thus, adding first-class environments to a language means that the language cannot be implemented easily on some environment-less architectures such as the G-machine [16]. Jagannathan's environment-based reflection language Rascal [9] is another language with first-class environments. Rascal provides a way to identify reifiable (exportable) variables; however, it still suffers from the first two disadvantages mentioned above, because some non-reifiable variables are dynamically bound.

Lamping's unified system of parameterization [10] emphasizes the other part of our approach. It uses a special form **data** to identify non-lexical variables. But instead of providing an explicit resolution operation like **resolve**, it always uses the run-time dynamic environment to resolve the non-lexical variables. They are therefore truly dynamic variables. Consequently, programs are harder to analyse and the language is difficult to implement efficiently.

Quasi-static scoping can be characterized as a mechanism for run-time linking as opposed to the compile-time linking provided by other systems. It is just one point in the spectrum of linking disciplines ranging from static (compile-time) to dynamic (run-time). We are convinced that in a programming environment various linking disciplines are necessary for different purposes. Our goal is therefore to investigate if there exists a small set of linking disciplines that will suffice for all needs. If not, we would like to establish a metric for measuring the degree of "staticity," or equivalently "dynamicity," in order to classify all the possible disciplines.

We are currently exploring various ways of speeding up the resolution process. For instance, we can assign types to both resolvers and quasi-static procedures and initiate the resolution process only if their types match. Then, run-time external name matching would be eliminated.

Finally, for reasoning about quasi-static scoping, we have developed a calculus that is consistent and has a standardization procedure. The calculus is an extension of Felleisen and Hieb's calculus of sequential state [8], which in turn is an extension of Plotkin's call-by-value λ-calculus [17]. We will continue its development into a more complete logical system and report it elsewhere.

Acknowledgements. We are grateful for the insightful comments by Kent Dybvig, Chris Haynes, and the late Bob Hieb during the early stages of this research. We would also like to thank Mike Ashley, Matthias Felleisen, Julia Lawall, Jon Rossie, and John Simmons for their comments on earlier drafts of this paper.

References

[1] H. Abelson and G. J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 277–288, 1988.

[3] L. Cardelli. Compiling a functional language. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 208–217, 1984.

[4] W. Clinger and J. Rees (editors). Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, 1991.

[5] P. Curtis and J. Rauen. A module system for Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 13–19, 1990.

[6] E. W. Dijkstra. Algol 60 translation. *Supplement ALGOL Bulletin 10*, 1960.

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

[8] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[9] S. Jagannathan. Reflective building blocks for modular systems. To appear in the IMSA '92 International Workshop on Reflection and Meta-Level Architecture.

[10] J. O. Lamping. A unified system of parameterization for programming languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 316–326, 1988.

[11] K. J. Lang and B. A. Pearlmutter. Oaklisp: An object-oriented dialect of Scheme. *Lisp and Symbolic Computation*, 1(1):39–51, 1988.

[12] J. S. Miller and G. J. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation*, 4(2):107–141, 1991.

[13] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[14] J. H. Morris Jr. Protection in programming languages. *CACM*, 16(8):15–21, 1973.

[15] K. Nørmark. Simulation of object-oriented concepts and mechanisms in Scheme. Technical Report R 90-01, Institute of Electronic Systems, Aalborg University, January 1990.

[16] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[17] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[18] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition, 1990.