Language Extensibility via First-class Interpreters
and Constructive Modules

David Espinosa

Ph.D. Thesis Proposal

Columbia University
Department of Computer Science
New York, NY 10027
espinosa@cs.columbia.edu
April 11, 1993

1 Abstract

This document proposes these theses:

- First-class interpreters offer a flexible means of language extensibility.
- A simple module system can provide:
  - incremental specialization
  - program construction rather than program organization
  - static rather than dynamic composition of programs
  - explicit rather than implicit control over composition

The former two claims duplicate properties presently associated only with object-oriented programming, while we regard the latter two as improvements.

2 Introduction

Software design *is* language design *for particular problems*. The functional programming community has stressed the power and flexibility of this view on many occasions [Hen82, AS85, Hug90, Wad87, HS88], and one of the goals of this thesis is to emphasize it once more.

According to this point of view, a programming language is a vehicle for constructing problem-specific languages. Existing languages support this view to varying degrees, but few allow significant semantic extensions to the language itself. This thesis aims to show that such a mechanism is desirable and possible. It suggests first-class interpreters as an organization, and argues that they are easy to use and flexible.

The point of first-classness for interpreters is not quite the same as for other objects. There is not necessarily a need to run many interpreters at once or to use them as data objects. The thesis will not define combinators or predicates on interpreters. Instead, first-classness means that interpreters can be composed from parts in different ways. These parts will have a well-defined and hopefully simple semantics and will provide a language for language building.

Object-oriented methods suggest an organization in the form of a "generic interpreter". However, the standard techniques of object-oriented programming are undesirable on several accounts. These problems lead to the second half of the thesis, which is to analyze, duplicate, and extend object-oriented techniques using first-class modules, which we find semantically and practically more attractive.

This proposal has the standard organization. Section 3 presents new ideas and approaches. Section 4 discusses a prototype extensible interpreter which can demonstrate some applications. Section 5 reviews previous work in this area. Section 6 presents a plan for solving the problem. Section 7 details specific goals and timetables. Section 8 outlines the eventual thesis.

2.1 Why extensibility?

Language extensibility leads to simpler, better organized, more efficient programs. These improvements permit more reliable and less costly software. As with any idea, some uses for extensibility are already apparent, while others will only be found once languages supporting it become more widely available. In fact, there is a movement toward including embedded languages such as TCL [Ous90] in other programs to provide extensibility.

Programs will be simpler and better organized because linguistic abstraction allows the language of the *program* better to mirror the language of the *problem*. Performance can be enhanced by building extensions that are tailored to particular features of a problem.

Studies of software complexity have shown that language is a powerful and robust method for organizing programs. By making these benefits more available, extensibility has the potential to improve the quality of software.

2.2 Examples of extensibility

This section gives, without much explanation, a list of potentially useful linguistic extensions. A later section will develop two examples, non-determinism and normal-order procedures.

- Poor man's partial evaluation
  Creating specialized procedures using code constructors.
- Extensible generic arithmetic
  For organizing symbolic algebra systems.
- Identity-changing objects
  Objects that can *become* other objects.
- Incremental programming
  Maintaining dependencies for backtracking in the presence of changes.
- Expression-level manipulation of procedures
  For scientific computation and compiler analyses.
- Constraint programming
  Constraints allow computation to proceed in multiple directions.
- Partial closures
  Procedures whose arguments may be supplied incrementally.
- First-class expressions
  Expressions whose names are carefully controlled to avoid conflict.
- Pattern match argument passing
  More general procedure call interfaces.
- Normal order evaluation
  By-name parameter passing.

- Non-deterministic evaluation
  McCarthy's `amb` operator.

- Debugging
  Stepping, variable monitoring, and tracing.

- Lamping's unified system of parametrization [Lam88]
  Dynamic binding with expression-like composition.

- Abstract interpretation
  Program analysis and compilation using the language itself.

- Automatic implementation selection
  A construct for globally controlled dynamic choice.

- Composable continuations
  Capturing the continuation as a composable function.

- Various types of parallel procedure calls
  Wrapping `touch` and `future` around expressions in a call.

- Parallel simulation
  Simulators for consistency, scheduling, and placement models.

- Parallel OR
  Short-circuiting non-termination through parallelism.

- Object system
  Experimenting with object-oriented extensions.

- Explicit-demand interpreter
  Explicit representation of the result demanded from a computation.

- Non-deterministic merge
  Streams for building functional operating systems.

3 Ideas

The idea of a generic interpreter is clear to anyone who works with generic functions and interpreters. But it is not clear that object-oriented techniques such as inheritance and dynamic dispatch are necessary to realize the concept. The functional programming analog to a generic interpreter is a *first-class* interpreter.

Hence, the contributions of the work will be, first, to develop a new module system which can construct programs from components in a new and significant way, and, second, to show that first-class interpreters are a convenient and flexible means of language extension.

3.1 First-class interpreters

An object is called "first-class" if it can be

- passed as an argument to a procedure,
- returned as a result from a procedure,
- held as the value of a variable, and
- stored in a data structure.

First-classness is a powerful tool for achieving simplicity, clarity, and expressive power. This approach to language design may be summarized as follows. To extend a language with a vocabulary for expressing a computational domain, find the essence of that domain and incorporate it as a first-class object. For example, Scheme includes procedures for abstraction and continuations for control. It most likely *should* have included cells for assignment.

For language extension, we choose as fundamental the interpreter. There are other possibilities; we could choose instead compilers or semantic specifications. The possibility of extending this approach from interpretation to compilation is discussed in section 3.4.

The only operation on first-class interpreters is `eval`, which accepts an interpreter, an expression in the language the interpreter supports, and a context in which to evaluate the expression. The result is a new context, part of which is the value of the expression.

Although uses for multiple interpreters will be investigated, first-class interpreters are best employed to implement appropriate languages for particular problems. Hence, a program will tend to use only a few at a time.

Perhaps the central point of the first-class approach is that interpreters can be constructed feature-by-feature, instead of monolithically. The importance of modularizing language specification and implementation is described in Stephen Adams' thesis [Ada91].

Abstractly, any language has three parts:

- a set of primitives
- a means of combination
- a means of abstraction.

The primitives for our language-building language will be parts of interpreters in the form of modules.

3.2 Modules

This section describes why current module systems are incapable of incremental specialization and suggests how the problem can be solved.

The dynamic semantics of many object-oriented languages (Smalltalk [Gol85] and Self [US87] are examples) is usually described by passing an implicit "self" argument to each method. This semantics aids code sharing by allowing methods inside an object to refer to it from outside and thereby "see" bindings which have been added on afterward. Modules have no provision for this type of reference.

For example, suppose that a class `A` has methods `x`, `y`, and `z` and that a class `B` inherits from `A`. Suppose further that B redefines `x` but that `y` calls `x`. We desire that `y` should find `B`'s version of `x`. To realize this situation in terms of modules is a standard problem. We represent each class as a module:

```
(define A
  (module
   'x (lambda (a) (* a a))
   'y (lambda (b) (+ b (x 5)))))
```

```
(define B
  (module-layer
   (module 'x (lambda (a) (* a a a)))
   A))
```

Then we desire for `B` to be equivalent to the module constructed by

```
(module
 'x (lambda (a) (* a a a))
 'y (lambda (b) (+ b (x 5))))
```

The problem is that if modules map names to values, then the procedure `(lambda (b) (+ b (x 5)))` is closed over `x` when module `A` is created. Subsequent operations cannot change this binding, and module `B` will not be formed as desired.

An approach to the solution of this problem is to delay the resolution of bindings until a specific "fixing" operation is applied. Once "fixed", module bindings cannot be altered. In effect, we introduce a type of "pre-module" which cannot be executed but which can be manipulated through a set of module operations.

Module combinators are similar to higher-order functions, except that they manipulate whole interfaces rather than single procedures. The abstract data type of module interfaces has much more structure than just procedure application. Category theory (see [Pie91]) may provide some formal assistance; for instance, it may be possible to define a co-product of two modules which implement the same interface. Also, I have recently investigated (with Franklyn Turbak of MIT) binding constructs and first-class names. The problems faced here are similar.

Another point worth noting is that the general confusion between types and implementations, which was once thought to be well-understood, has been muddied in the presence of inheritance. The more difficult question of coercion between implementations versus inclusion among types has probably never been understood. Issues of efficiency and multiple implementations complicate matters further, leading Abelson and Sussman to remark in [AS85], "We do not yet understand data types." These issues all surface in the domain of generic arithmetic systems, and we find it quite naive to suppose that generic functions can adequately solve the generic arithmetic problem (see [Zip93]).

The present work may have applications to these issues since I believe that a successful module system needs to be able to distinguish types and implementations in the presence of multiple implementations and coercions.

3.3 Extensible denotational semantics

This section outlines an approach to some questions about the context in which an expression is evaluated. We consider an interpreter as a map from an expression and a context into a new context. The context is *annotated* as it is threaded through the execution of a program. Typical annotations are value, environment, store, and continuation.

The importance of annotation is that it enhances modularity. A construct need not disassemble the context and reassemble it to change a single field. This seemingly minor detail allows knowledge to be localized in the program text to the places in which it used. Annotatable objects serve the same purpose as the store in an imperative language: to transmit information without disturbing intermediate computations. They are also similar to association lists in Lisp. The concept of annotation is taken directly from Haase's work on the FRAMER knowledge representation system [Haa92].

A simple example in an interpreter is a possible rule for constants:

```
(define (eval-constant exp context)
  (annotate context 'value (constant-value exp)))
```

The reader may object that annotation is too general and therefore inefficient. Besides, in an interpreter, the environment, store, and continuation are used in different ways. Various restrictions hold; for example, the continuation cannot be mutated, and the store cannot be captured.

These restrictions can be captured by a *protocol*, that is, a contract, not statically enforceable, about what operations are valid on what objects. The assertions about behavior made by a protocol can be used by a compiler to enhance efficiency.

Second, if an association list must be accessed to find the environment, the resulting interpreter will be slow. However, the interpreter can be transformed into the more usual form by a general "unpackaging transformation". This transformation would change the constant evaluator shown above into

```
(define (eval-constant exp value env rest-of-context)
  (return (constant-value exp) env rest-of-context))
```

The application of this transformation could be controlled by the user. This idea is by no means a complete solution to the efficiency problem, but it suggests two important points:

- transformations which cannot be easily applied in general can still be applied in particular under user direction
- an idea should not be discarded too quickly on grounds of inefficiency; often some trick may be applied.

The annotation approach suggested here may be well-known in the field of attribute grammars; it is not expected to be new.

3.4 Efficiency

Interpreters are inherently less efficient than compilers, and this thesis *could* be about extensible compilers. Compilers are more complex than interpreters for two reasons:

- Instead of values, they generate programs which generate values
- Compilers perform considerable specialization.

Since the aim of the thesis is to study language extensibility and program composition, the complexity of compilation is unnecessary. To build a competitive extensible compiler would require much unrelated effort. A non-production compiler would be less difficult and is a possible additional topic.

Extensible systems are not inherently efficient; in fact, they are usually less efficient on account of their generality. However, they allow efficient constructs to be incorporated naturally. For example, existing language features can be tailored to specific problems. In [Kic92], Kiczales describes the creation of a new metaclass for objects with many instance variables. A Scheme interpreter could be extended with specialized procedure calls which expect specific numbers or types of arguments. Of course, added constructs do not need to shadow existing features; they can be entirely new.

Semantic analyses and optimizations can be applied to generate compilers from interpreters. These transformations are reasonable because, once interpreters are constructed, their program text and semantics do not change.

Even superficial partial evaluation will remove dispatches on expression type, and the resulting program can potentially be compiled since it is in the implementation language of the interpreter. Semantics-directed compiler generation techniques are also applicable since interpreters closely resemble denotational specifications.

4 A preliminary implementation

This section describes a preliminary generic interpreter for Scheme written in SOS, the Scheme object system. SOS is a variant of CLOS implemented by Chris Hanson of MIT and is based on TINY, a CLOS in Scheme written by Gregor Kiczales at Xerox PARC. Also presented are two applications of the interpreter: an extension for call-by-value procedures and an extension for McCarthy's `amb` operator. The latter is interesting because `amb` is seldom implemented correctly.

4.1 The interpreter

The example system is built from several parts:

- a parser, which translates from concrete to abstract syntax
- an evaluator, which maps expressions to values
- a user interface, which allows the user to interact with the evaluator.

The parser and user interface are not part of the interpreter but are needed for interactive use. A parser can be viewed as a mechanism for constructing abstract syntax. Programs which write programs would use some form of parser, but it would not necessarily be as complex as needed by a human.

The concrete syntax of programs is built from Lisp list structure. The abstract syntax objects, or *expressions*, are instances of classes such as `<variable>`, `<lambda>`, etcetera.

Each of these interpreter parts is extensible by adding methods to generic procedures. A final version of the interpreter would include rigorous documentation of the protocol for its modification. It could also include other extensible parts, such as an error system.

The evaluator accepts an expression, an environment, and a continuation. For a discussion of how this interface could be extended, see section 3.3.

```
(define-generic-procedure eval (exp env k))
```

Its default method (specialized to the most general class, `<object>`) is that it can't do anything:

```
(define-method eval ((exp <object>) env k)
  (error "Can't eval: " exp))
```

To that, we add methods for each of the abstract syntax classes. For example:

```
(define-method eval ((exp <constant>) env k)
  (k (constant-value exp)))
```

This standard interpreter is augmented with a carefully chosen set of hooks which form its meta-object protocol. For instance, the method for variable lookup calls a generic procedure `lookup-transform`:

```
(define-method eval ((exp <variable>) env k)
  (env-lookup
   (variable-name exp) env
   (lambda (v) (lookup-transform v env k))
   (lambda () (error "Unbound variable: " (variable-name exp)))))

(define-generic-procedure lookup-transform (value env k))

(define-method lookup-transform ((value <object>) env k)
  (k value))
```

The default method for transforming values is just to return them. We shall see other possibilities soon. The evaluation method for procedure calls is also overrideable:

```
(define-method eval ((exp <call>) env k)
  (eval (call-procedure exp) env
        (lambda (procedure)
          (eval-arguments
           procedure (call-arguments exp) env
           (lambda (arguments)
             (apply procedure arguments k))))))
```

```
(define-generic-procedure eval-arguments (type arguments env k))

(define-method eval-arguments ((type <object>) arguments env k)
  (let loop ((arguments arguments) (k k))
    (if (null? arguments)
        (k '())
        (eval (car arguments) env
              (lambda (value)
                (loop (cdr arguments)
                      (lambda (values) (k (cons value values))))))))))
```

The parser, usually called a syntaxer in Scheme, first maps keywords to concrete syntax classes, then maps these to abstract syntax classes. The purpose of the concrete syntax classes is to facilitate code reuse.

```
(define-class <syntax> ())

(define-class <syntax-let> (<syntax>))  ; this isn't the only one!

(define (syntax exp)
  (generic-syntax (make-instance (syntax-class exp)) exp))

(define (syntax-class exp)
  (cond ((symbol? exp) <syntax-variable>)
        ((pair? exp)
         (let ((entry (assq (first exp) *special-forms*)))
           (if entry (cdr entry) <syntax-call>)))
        (else <syntax-constant>)))

(define-generic-procedure generic-syntax (class exp))

(define-method generic-syntax ((type <syntax-lambda>) exp)
  (make-instance
   (concrete->abstract type)
   (second exp)
   (syntax-implicit-begin <syntax-begin> (cddr exp))))
```

To state that a form is syntaxed “just like let”, we declare its syntax class to inherit from `<syntax-let>`. We also declare the keyword associated with the new form. The generic procedure `concrete->abstract` is a part of the protocol that allows an expression which is "just like let" to be assigned an abstract class other than `<let>`.

4.2 First example: Normal-order procedures

In this section, we show how call-by-name (also called normal-order) procedures can be added to the interpreter. To build a new special form, we declare an abstract syntax class and then specify how to parse expressions into that class:

```
(define-class <normal-lambda> (<lambda>))

(define-class <syntax-normal-lambda> (<syntax-lambda>))

(add-special-form! 'normal-lambda <syntax-normal-lambda>)

(define-method concrete->abstract ((type <syntax-normal-lambda>))
  <normal-lambda>)
```

We also specify classes for the forms `normal-let` and `normal-letrec`. The special form `letrec` in Scheme is for binding sets of values which depend on each other or on themselves.

```
(define-class <normal-letrec> (<letrec>))

(define-class <syntax-normal-let>    (<syntax-let>))
(define-class <syntax-normal-letrec> (<syntax-letrec>))

(add-special-form! 'normal-let    <syntax-normal-let>)
(add-special-form! 'normal-letrec <syntax-normal-letrec>)

(define-method concrete->abstract ((type <syntax-normal-let>))
  <normal-lambda>)
(define-method concrete->abstract ((type <syntax-normal-letrec>))
  <normal-letrec>)
```

Since `normal-let` is implemented as a macro, its concrete syntax class is `<normal-lambda>` rather than `<normal-let>`, which doesn't actually exist. This anomaly is unfortunate and will hopefully disappear.

Since <normal-lambda> and <normal-letrec> are declared as subclasses of <lambda> and <letrec>, it is necessary only to describe the differences between them. These are the evaluation methods which give the constructs their semantics. First, normal lambdas create normal procedures:

```
(define-method eval ((exp <normal-lambda>) env k)
  (k (make-instance <normal-procedure> exp env)))
```

Second, the arguments to a normal procedure (or the values bound in a normal-letrec) are evaluated by wrapping them as instances of a new class <thunk>. These methods override the standard argument evaluation methods, but only for procedures of class <normal-procedure> or expressions of the class <normal-letrec>.

```
(define (normal-eval-arguments call-next-method type arguments env k)
  (k (map (lambda (exp) (make-instance <thunk> exp env))
          arguments)))

(add-method eval-arguments
  (make-method (list <normal-procedure>) normal-eval-arguments))

(add-method eval-arguments
  (make-method (list <normal-letrec>) normal-eval-arguments))
```

The procedure add-method is used to add a previously defined procedure to a generic function. The only other semantic change is to specify that thunks should be forced when they are found as values of variables. This is in fact sufficient to implement call-by-name. To a semanticist, that is obvious. To an ordinary programmer, it is a miracle. The usual method of repeatedly forcing the result at the end of the computation is less insightful.

```
(define-method lookup-transform ((thunk <thunk>) env k)
  (eval (thunk-exp thunk) (thunk-env thunk) k))
```

Finally, we define the classes of thunks and normal procedures:

```
(define-structure-class <thunk> (<value>) exp env)

(define-class <normal-procedure> (<procedure>))
```

That is all the code necessary to implement normal-lambda, normal-let, and normal-letrec. Since normal procedures inherit from ordinary procedures, they are applied by the same methods. This code is part of the evaluator, not the extension:

```
(define-generic-procedure apply (procedure arguments k))

(define-method apply ((procedure <object>) arguments k)
   (error "Can't apply: " procedure))

(define-method apply ((procedure <procedure>) arguments k)
  (eval (lambda-body (procedure-lambda procedure))
        (env-extend
         (map make-binding
              (lambda-names (procedure-lambda procedure))
              arguments)
         (procedure-env procedure))
        k))
```

Here are some examples of the interpreter running:

```
(go)
==> ((lambda (x) 3) (display "foo"))foo      ; <== foo
3

==> ((normal-lambda (x) 3) (display "foo"))  ; <== no foo
3

==> (define loop
      (letrec ((loop (lambda () (display "foo") (loop))))
        loop))
loop

==> (let ((a (loop)))
      7)foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
;Up!

(go)
==> (normal-let ((a (loop)))
      7)
7

==> (normal-letrec
```

```
     ((a (+ c 1))
      (b (- a 3))
      (c 9))
     (list a b c))
(10 7 9)
```

Although the usual `cons` still evaluates its arguments, we can make one that doesn't. In all cases here, we could use normal lambda in place of ordinary lambda, but we don't have to.

```
==> (define church-cons
      (normal-lambda (a b)
        (lambda (m) (m a b))))
church-cons

==> (define church-car
      (lambda (x)
        (x (normal-lambda (a b) a))))
church-car

==> (define church-cdr
      (lambda (x)
        (x (normal-lambda (a b) b))))
church-cdr

==> (church-car (church-cons 1 2))
1

==> (church-cdr (church-cons 1 2))
2

==> (church-car (church-cons 1 (loop)))
1

==> (church-cdr (church-cons 1 (loop)))foofoofoofoofoofoofoofoofoo
foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
;Up!
```

Also, we could introduce a `normal-call` in addition to `normal-lambda` which would allow finer per-call distinctions of call-by-name versus call-by-value.

4.3 Second example: Amb

Amb is a non-deterministic choice construct introduced by McCarthy in about half a page of text [McC63]. The semantics of amb can be interpreted in different ways, but the general idea is that (amb 1 2) can return 1 or 2, non-deterministically. We say that the meaning of an expression is the set of all possible values which it can return, and we set as our goal to implement an interpreter which returns a (potentially infinite) stream of possible values. There is also an associated form (fail) which indicates that no values should be returned. Most of the necessary additions to the interpreter are changes to the read-eval-print loop. The semantics of amb are extremely interesting, but their interaction with the rest of the interpreter is in fact minimal.

A typical use of amb is in a generate-and-test paradigm. This example is from [Zab88]:

```
(define (any)
  (amb 0 (1+ (any))))

(define (any-prime)
  (let ((n (any)))
    (if (prime? n)
        n
        (fail))))
```

To add amb to our interpreter, we introduce the necessary syntax classes as before:

```
(define-structure-class <amb>  (<expression>) first second)
(define-structure-class <fail> (<expression>))

(define-class <syntax-amb>  (<syntax-uniform>))
(define-class <syntax-fail> (<syntax-uniform>))

(add-special-form! 'amb  <syntax-amb>)
(add-special-form! 'fail <syntax-fail>)

(define-method concrete->abstract ((type <syntax-amb>))  <amb>)
(define-method concrete->abstract ((type <syntax-fail>)) <fail>)
```

Uniform syntax means that all arguments are syntaxed uniformly. For example, if has uniform syntax. Now we add two methods to the evaluator:

```
(define-method eval ((exp <amb>) env k)
  (run! (lambda () (eval (amb-first exp)  env k)))
  (run! (lambda () (eval (amb-second exp) env k)))
  (next))

(define-method eval ((exp <fail>) env k)
  (next))
```

The procedures `run!` and `next` are part of a simple task abstraction that is necessary for breadth-first traversal of the tree of choice points. For brevity, we omit the queue abstraction.

```
(define *queue*)

(define (init!)
  (set! *queue* (make-queue)))

(define (run! thunk)
  (enqueue! thunk *queue*))

(define (next)
  (if (queue-empty? *queue*)
      'FAIL
      ((dequeue! *queue*))))

(define (run-loop)
  (let ((v (next)))
    (if (eq? v 'FAIL)
        the-empty-stream
        (cons-stream v (run-loop)))))
```

The procedure `next` returns either a value or the token `FAIL`. The procedure `run-loop` assembles the non-failure values into a stream of results. The only remaining changes are to the read-eval-print loop. The meta-object protocol of the loop includes the procedures `make-eval`, `make-define-eval`, `make-env`, `make-cont`, and `make-printer`, whose default definitions are:

```
(define (make-eval)        *standard-eval*)

(define (make-define-eval) (make-eval))
```

```
(define (make-env)         *standard-env*)

(define (make-cont)        *standard-cont*)

(define (make-printer)     *standard-printer*)

(define *standard-cont*    (lambda (x) x))

(define *standard-printer* write-line)

(define *standard-eval*
  (lambda (exp env)
    (eval (syntax exp) env (make-cont))))
```

A read-eval-print loop is a good example of a facility which would vary per interpreter, but which does not seem natural to implement using generic functions. Perhaps this can be repaired. The changes for `amb` are:

```
(define (make-eval)
  (lambda (exp env)
    (init!)
    (run! (lambda () (*standard-eval* exp env)))
    (run-loop)))

(define (make-printer)
  (lambda (x) (print-stream 20 x)))

(define (make-define-eval)
  (lambda (exp env)
    (let ((s ((make-eval) exp env)))
      (if (empty-stream? (tail s))
          (head s)
          (error "Can't use define with ambiguous values!")))))
```

Among the papers on `amb` which I have seen [BA81, Bro83, Kos79, Zab87, Zab88, Cli81, Cli82, PS88, McC63], there are few discussions about implementation. The interpreters given by Zabih [Zab88] and Wadler [Wad92] are deficient in that they traverse the tree of choice points in a depth-first ordering. No depth-first ordering can produce all results for all programs. Indeed, for these interpreters, reversing the order of arguments to amb in the any function seen earlier results in non-termination. The importance of traversal order for programs with infinite branches may be perfectly well-known in the Prolog community, however. Here are examples of amb in action:

```
(go)
==> (amb 1 2)
{1 2}

==> (fail)
{}

==> (amb 1 (amb (fail) 3))
{1 3}

==> (define any
      (letrec ((any (lambda ()
                      (amb 0 (1+ (any))))))
        any))
any

==> (any)
{0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...

==> (letrec ((any (lambda ()
                    (amb (1+ (any)) 0))))   ; reverse arguments
      (any))
{0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...

==> (let ((n (any)))
      (if (even? n)
          n
          (fail)))
{0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 ...
```

Finally, we can give an example using both normal-order procedures and amb. This example does not demonstrate true composability since these extensions do not really interact, but I will investigate more substantial examples later. The mixing properties of amb, applicative order, and normal-order are well-known [Cli82, Wad92].

```
==> ((lambda (x) (+ x x)) (amb 1 2))
{2 4}

==> ((normal-lambda (x) (+ x x)) (amb 1 2))
{2 3 3 4}
```

5 Previous work

This section first discusses previous work on extensibility, then on modularity and program composition. There is undoubtedly much work that I have not yet examined, but I plan to overview published approaches to these problems and discuss interesting cases in detail.

5.1 Extensibility

This section covers various traditional approaches, such as procedures, macros, and interpreters, and also some relatively new ones, such as meta-object protocols and monads.

5.1.1 Procedures

Procedures are at present, and in most imaginable futures, *the* way of constructing new languages for computation. For example,

```
(define (square x)
  (* x x))
```

extends Scheme to include expressions such as `(square 5)`. Procedures are limited both syntactically and semantically, although less so in normal-order languages, in which forms such as `if` can be implemented as procedures.

5.1.2 Macros

In Lisp, macros are typically used in three ways (thanks to Gerald Sussman for this articulation):

- For postponing argument evaluation in procedures
- For inlining procedure calls

- For defining new syntactic forms

It may be argued that evaluation postponement and inlining are best carried out by other means, but syntactic extension is important for several reasons. It points out a new pattern of usage for previously existing language, and it provides a convenient shorthand for accessing this pattern. The availability of shorthand is essential to the programmer, who has limited time and patience. This is arguably the real reason for the existence of macros.

In some systems, macros can also be used to powerful effect for redefining existing syntactic forms. If the macro system provides a handle on variable reference and procedure call (even though these have no associated keywords in most Lisps), then it appears that virtually any language extension can be constructed locally by syntax-direction compilation. For example, section A.1 gives a set of macros that adds assignment to "pure Scheme". In the absence of higher-order procedures in the base language, the transformations would be more complicated but similar.

If the translation is not required to be performed "locally", then, given a sufficiently powerful macro defining language, we can write

```
(compile <program>)
```

and allow the `compile` macro to do whatever is necessary.

Even if macros are sufficiently powerful for most extensions, there are other factors that discourage their use, such as their complexity and syntactic orientation.

On the positive side, because syntax composes extremely well, so do macros, and, because macros operate on a static program text, they are compilable and therefore efficient.

5.1.3 Interpreters and Compilers

Interpreters allow new languages to be built over old ones. They are by definition the most flexible means of extension but are inconvenient because:

- They do not interface easily to their environments.

- They must be constructed from scratch, even if only a single new feature not present in the language is desired.

- They are inefficient because they do not take advantage of a static program text.

Compilers suffer from the first two disadvantages but not the third. With efficiency as a goal, they sacrifice flexibility and ease of implementation.

5.1.4 Meta-object protocols

The term *meta-object protocol* currently represents two ideas. The more special concept is language extension though incremental specialization using object-oriented design methods (see [KdRB91]). The more general concept is an interface to the implementation of an abstraction to allow increased functionality and efficiency (see [Kic92]). Both are exceedingly good ideas (although [KdRB91] would have been better titled *A Meta-object Protocol for CLOS*).

The essence of the CLOS MOP is to show how extensibility can be provided using generic functions. This method allows new extensions to be built, first, without altering existing semantics and, second, by specifying only the *differences* between the desired semantics and the existing semantics. These features are extremely valuable.

The work proposed here will attempt to show that the functionality provided by this type of meta-object protocol can be obtained with more simplicity and generality by using modules instead of generic functions and inheritance. Hence the proposed ideas could be called a meta-object protocol for Scheme. In any case, the extended language, types of extensions allowed, and goals of extension are different from those in CLOS.

5.1.5 Language-based Environments

Any language implementation that includes a runtime system written in the language must provide some access to the implementation. This can be called (if necessary) a meta-object protocol. For example, MIT Scheme allows varying levels of access across its domains and supports modification of the implementation through incremental redefinition (see [MR91]) and several types of hooks. It was not built in general to be extensible.

The Scheme language itself is remarkable for its control over the environment, store, and continuation domains, yet still only provides a fixed protocol for manipulating them. In order to define objects which deal differently with these structures, the programmer must interpose machinery between the language and his program. This machinery, however well disguised, is never entirely transparent, and the programmer is always aware that he is working in a Lisp with a new construct stuck on, rather than a dialect of Lisp with that construct. Furthermore, since Scheme was not designed to be extended, new constructs are added by what are essentially tricks rather than natural means.

Since it was designed as a programming environment, Smalltalk (see [Gol85]) includes much reflective power. However, the use of inheritance in its runtime system is fairly unprincipled, and the implementors' approach to modification is to provide the source code to the entire system. Although the library is well-written, it was not originally intended to be modified by the user.

5.1.6 Towers of interpreters

Beginning with Brian Smith's original invention of 3-Lisp [Smi84], there has been a series of papers on reflective towers of interpreters, in which each one runs the code of the one below it [WF88, Baw88, JF92]. These are nice conceptually but do not seem suited to practical application for efficiency reasons, especially the infinite (or even transfinite (!) [LK]) models.

5.1.7 Monads

Monads are a categorical method of interpreter extension which allows additional information to be passed through a purely functional interpreter. This use of the categorical concept was recognized by Moggi [Mog89], but see [Wad92] for an excellent presentation. In general, monads do not compose because their structure is unconstrained. As a formal device, they inherit the usual benefits (precision, analyzability) and drawbacks (limited scope). Most importantly, the category theory behind them does not seem to provide fundamentally new insight. It is easy to become bound to a theory simply because it is the only *formal* alternative available. Although the present work will go beyond the capabilities of monads, some of the flavor of the monadic life will be retained. Also, monadic interpreters are perhaps the first parametrized interpreters in the literature.

5.1.8 Semantics-directed compiler generation

This field (see [Wan84, Lee89, Tof90, App85]) attempts to generate compilers from declarative specifications. Apparently, the results have been good, but perhaps the work has not yet found its audience. The tools are not easily usable and are not well integrated into current programming environments. The resulting compilers are not yet efficient enough for general use. Still, this is an active area of research and, if the questions of efficiency can be overcome, may render hand-built compilers obsolete.

5.1.9 Other first-class interpreters

John Simmons [SJ92] presents a one-page outline for a system of first-class interpreters. He proposes a system of *dispatches* and *prelims*, where dispatches handle syntactic forms (as above) and prelims are performed before dispatches. His discussion also indicates that his work will be biased toward the reflective tower approach, whereas the work proposed here will emphasize forming interpreters through module composition. Mike Ashley at Indiana is also reportedly working on a "meta-object protocol for Scheme", but I have not inquired about his work.

5.1.10 Partial evaluation

Richard Schooler's thesis [Sch84] describes an attempt to regain efficiency in user-defined constructs by means of partial evaluation. Essentially, his thesis is not about language extensibility, but about using partial evaluation (applied through explicit annotations) to improve performance.

5.1.11 Extensible compilers

There is a history of work on extensible compilers, beginning with the program transformation approach [GB86, Kan86]. Presently, several groups at Xerox PARC [LKRR92, ACS92] are working on protocols for enhancing object code efficiency, rather than source language expressivity, as stressed here. There has also been some work on automatically generated global optimizers [WS91, TH92, Wei].

5.2 Previous module systems

Although I have not completed an adequate review, I believe that module systems to date have focused on program structuring rather than program composition. Those such as Oberon [RW92] which address object-oriented concerns will be particularly interesting to review.

5.2.1 Modules for higher order languages

Well-known module systems for higher-order languages are the ML module system [Mac84], Curtis and Rauen's Scheme module system [CR90], Rees's Scheme48 modules [RK], and Rees's recent proposal [Ree92]. These systems support more complex features, such as types and macros, but provide little control over namespaces, which I view as a more central objective. Polymorphic systems such as ML's do not parametrize bindings over objects but only over types. They are not designed to construct programs, only to organize them.

5.2.2 Inheritance

Again, I have not thoroughly reviewed the object-oriented community's literature, but it is my understanding that the tension between modules and inheritance has not been resolved. This tension goes back at least to Cardelli and Wegner's now classic survey paper [CW85]. There are several reasons why I believe that modules are preferable to inheritance as a means of reusing software.

First, modules specify binding explicitly and statically. By employing a flexible interconnection language, the form of the resulting program can be highly unconstrained, yet once a program's structure is fixed, the potential for compilation and increased efficiency is open. Techniques for improving dynamic performance, such as method caching, are unnecessary if no dynamic activity is actually present.

Second, methods are added to generic functions by mutating tables associated with the functions. The functional programming community has shown that mutation is often less flexible than construction.

Third, inheritance is extremely useful but does not yet rest on a firm semantic foundation. The problem when multiple methods are applicable to a given call, there is no natural ordering on them and no clear way of combining their action. The default ordering and combination rules in CLOS [KdRB91] are user modifiable through a meta-object protocol, and, since there is no natural choice, allowing different options is generous, but it is not clear that such complexity is necessary. Because modules provide an explicit and straightforward means of combination, they potentially afford the programmer finer control over a program's organization.

Finally, tagging all objects in the system seems unnecessary when many types are known. Universal tagging is equivalent to forming a disjoint union over all types. Disjoint unions are more often needed only locally. However, it may be possible to leave modules "open" by binding a "generic interface" that allows later modification by mutation.

5.2.3 Generic Functions

Generic functions provide a powerful facility for structuring a system of multiple operations on multiple classes of data. As has been noted by Abelson and Sussman [AS85] and by Filinski [Fil89], message passing and procedure call organize an operation table by rows and by columns, respectively. Each one is preferred for certain classes of operations. Neither method is able to generalize in a nice way to dispatch on multiple arguments. The insight of generic functions is to specify the table not by row or by column, but by entry. The multi-dimensional tables necessary for more than one argument are easily specified in this way.

The purpose of inheritance is to extend a partial specification of the dispatch table to a total specification, but as discussed before, it is less than ideal and better means are needed. Indeed, the semantic complexity of CLOS and other object-oriented languages arises from inheritance, not from generic dispatch.

Generic functions also provide a form of argument pattern matching more general than ordinary procedure call or message passing. In the proposed work, dispatch and extensibility are needed, but neither multiple dispatch nor inheritance. However, generic functions have provided much of the inspiration.

6 What to do next

First, I plan to develop a module system for Scheme and a useful set of module combinators. Using them, I will attempt to duplicate the functionality of CLOS (including the MOP) with respect to code reuse. Then I will implement an interpreter construction set using the module system. The results can be compared with the generic procedure implementation already in progress. To test the kit, I will implement many of the examples described in section 2.2. This should allow me to refine the interface to the kit (and the module system), which I will document.

I will aim for maximal composability of language extensions and will produce a detailed analysis of the results (whether positive or negative). Next I will investigate the issue of efficiency and try to determine to what extent compilation is possible. Perhaps a reduced or restricted interface will enable some compilation. It may also be possible to improve interpreters by partial evaluation, once they have been produced by the toolkit. Finally I may be able to apply the module system to the design of extensible generic arithmetic systems.

7 Practicalities

This section discusses evaluation, deadlines, stopping conditions, and deliverables. The software includes:

- the module system
- the interpreter construction toolkit
- complete documentation

The thesis will discuss two ideas:

- language extensibility via first-class interpreters
- program composition using constructive modules

These ideas will be analyzed independently and tested somewhat independently by building interpreters using inheritance in place of modules.

- a description of ideas and work completed
- examples of how the module system works and is used
- at least five non-trivial examples of the toolkit
- a comparison between modules and inheritance with respect to flexibility, code reuse, and efficiency
- an analysis of extensibility
- an analysis of composability of language features
- an analysis of efficiency
- a comparison of the toolkit to other methods of extensibility

All comparisons and analyses will be based on concrete examples. Work will be undertaken sufficient to decide whether the approach succeeds or fails with respect to extensibility, composability, and efficiency.

When the deliverables have been produced, the thesis will be considered complete. Other projects or investigations such as partial evaluation, compilation, or generic arithmetic shall be considered optional (at the *author's* discretion). The expected date of completion is May 1994.

8 Thesis outline

Introduction

- Problems and Ideas

Synthesis

- Modules
- Toolkit implementation
- Toolkit interface
- Toolkit examples
- Generic interpreters

Analysis

- Inheritance versus modules
- Extensibility?
- Composability?
- Efficiency?
- Related work

Conclusion

- What has been accomplished?
- What can these ideas do for you?

A Appendices

A.1 Macros for assignment

This section gives a set of macros for adding assignment to pure Scheme. Its purpose is to demonstrate how difficult to write and hard to read such an extension is.

```
;;;; Adding a store to pure Scheme

;;;    One argument procedures only, WLOG (by currying).
;;;    Ignores capturing problems.

;; Sto  = Loc              --> Val
;; Exp  = Sto x Cont       --> Val x Sto
;; Cont = Val x Sto        --> Val x Sto
;; Proc = Val x Sto x Cont --> Val x Sto

(define-macro (%constant v)
  `(lambda (s k) (k ,v s)))

(define-macro (%variable v)
  `(lambda (s k)
     (k (store-contents s ,v) s)))

(define-macro (%if p c a)
  `(lambda (s k)
     (,p s (lambda (p s)
             (if p
                 (,c s k)
                 (,a s k))))))

(define-macro (%lambda var body)
  `(lambda (s k)
     (k (lambda (v s k)
          (store-bind
           s v
           (lambda (s loc)
```

```
              ((lambda (,var) (,body s k)) loc))))
        s)))

(define-macro (%begin e1 e2)
  `(lambda (s k)
     (,e1 s
          (lambda (v s)
            (,e2 s k)))))

(define-macro (%set! var e)
  `(lambda (s k)
     (,e s
         (lambda (v s)
           (k 'undefined
              (store-set-contents s ,var v))))))

(define-macro (%primitive p)
  `(lambda (s k)
     (k (lambda (v s k) (k (,p v) s)) s)))

(define-macro (%call p a)
  `(lambda (s k)
     (,p s
         (lambda (p s)
           (,a s
               (lambda (a s)
                 (p a s k)))))))

;;; Stores

(define (make-initial-store)
  '())

(define (store-contents s loc)
  (cdr (assq loc s)))

(define (store-set-contents s loc val)
  (cons (cons loc val) s))
```

```
(define (store-bind s val k)
  (let ((loc (cons #f #f)))           ; make a name -- could use integers
    (k (cons (cons loc val) s) loc)))

;;; For testing

(define (exp-value exp)
  (exp (make-initial-store) (lambda (v s) v)))

;;; Example:

(define (curried-* v1)
  (lambda (v2 s k)
    (k (* v1 v2) s)))

(exp-value
 (%call
  (%lambda fact
    (%call
     (%lambda n
       (%call
        (%lambda r
          (%begin
           (%set! fact
                  (%lambda ignored
                     (%if (%call (%primitive zero?) (%variable n))
                          (%variable r)
                          (%begin
                           (%set! r
                                  (%call (%call (%primitive curried-*)
                                                (%variable n))
                                         (%variable r)))
                           (%begin
                            (%set! n (%call (%primitive -1+) (%variable n)))
                            (%call (%variable fact)
                                   (%constant 'ignored)))))))
           (%begin
            (%set! n (%constant 5))
            (%begin
```

```
            (%set! r (%constant 1))
            (%call (%variable fact) (%constant 'ignored))))))
        (%constant 'r)))
      (%constant 'n)))
  (%constant 'fact)))
;Value: 120
```

A.2 Generic Interpreter Extensions

There are some minor syntactic differences between this code and what was given in the text, mostly to keep away from the Scheme evaluator's syntactic keywords. That problem was historical and has gone away, so the percent sign and g- prefixes will disappear. But this is only a prototype implementation and will doubtlessly be thrown away.

A.2.1 Normal-order Procedures

```
;;;; Normal order procedures

;;; Abstract syntax

(define-class <%normal-lambda> (<%lambda>))
(define-class <%normal-letrec> (<%letrec>))

;;; Values

(define-structure-class %thunk (<%value>) exp env)

(define-class <%normal-procedure> (<%procedure>))

;;; Evaluation methods

(define-method g-eval ((exp <%normal-lambda>) env k)
  (k (make-instance <%normal-procedure> exp env)))

(define (normal-eval-arguments call-next-method type arguments env k)
  (k (map (lambda (exp) (make-instance <%thunk> exp env))
          arguments)))

(add-method g-eval-arguments
  (make-method (list <%normal-procedure>) normal-eval-arguments))

(add-method g-eval-arguments
  (make-method (list <%normal-letrec>) normal-eval-arguments))
```

```
(define-method g-lookup-transform ((thunk <%thunk>) env k)
  (g-eval (%thunk-exp thunk) (%thunk-env thunk) k))

;;; Concrete syntax

(define-class <s-normal-lambda> (<s-lambda>))
(define-class <s-normal-let>    (<s-let>))
(define-class <s-normal-letrec> (<s-letrec>))

(add-special-form! 'normal-lambda <s-normal-lambda>)
(add-special-form! 'normal-let    <s-normal-let>)
(add-special-form! 'normal-letrec <s-normal-letrec>)

(define-method concrete->abstract ((type <s-normal-lambda>)) <%normal-lambda>)
(define-method concrete->abstract ((type <s-normal-let>))    <%normal-lambda>)
(define-method concrete->abstract ((type <s-normal-letrec>)) <%normal-letrec>)
```

A.2.2 Amb

```
;;;; Amb and Fail

;;; Abstract syntax

(define-structure-class %amb  (<%expression>) first second)
(define-structure-class %fail (<%expression>))

;;; Evaluation methods

(define-method g-eval ((exp <%amb>) env k)
  (run! (lambda () (g-eval (%amb-first exp)  env k)))
  (run! (lambda () (g-eval (%amb-second exp) env k)))
  (next))

(define-method g-eval ((exp <%fail>) env k)
  (next))

;;; Concrete syntax

(define-class <s-amb>  (<s-uniform>))
(define-class <s-fail> (<s-uniform>))

(add-special-form! 'amb  <s-amb>)
(add-special-form! 'fail <s-fail>)

(define-method concrete->abstract ((type <s-amb>))  <%amb>)
(define-method concrete->abstract ((type <s-fail>)) <%fail>)

;;; REP loop changes

(define (make-eval)
  (lambda (exp env)
    (init!)
    (run! (lambda () (*standard-eval* exp env)))
    (run-loop)))
```

```
(define (make-printer)
  (lambda (x) (print-stream 20 x)))

(define (make-define-eval)
  (lambda (exp env)
    (let ((s ((make-eval) exp env)))
      (if (empty-stream? (tail s))
          (head s)
          (error "Can't use define with ambiguous values!")))))

(define (make-cont) *standard-cont*)

;;; Tasks for breadth-first ordering

(load "queue.scm")

(define *queue*)

(define (init!)
  (set! *queue* (make-queue)))

(define (run! thunk)
  (enqueue! thunk *queue*))

(define (next)
  (if (queue-empty? *queue*)
      'FAIL
      ((dequeue! *queue*))))

(define (run-loop)
  (let ((v (next)))
    (if (eq? v 'FAIL)
        the-empty-stream
        (cons-stream v (run-loop)))))

;;; Stream procedures

(define (print-stream n s)
  (define (print-aux n s)
```

```
  (cond ((empty-stream? s)
         (display "}"))
        ((zero? n)
         (display " ..."))
        (else (display " ")
              (display (head s))
              (print-aux (-1+ n) (tail s)))))
(newline)
(display "{")
(if (empty-stream? s)
    (display "}")
    (begin (display (head s))
           (print-aux (-1+ n) (tail s)))))
```

References

[ACS92] Norman Adams, Pavel Curtis, and Mike Spreitzer. First-class data-type representations in SCHEMEXEROX. To be published, November 1992.

[Ada91] Stephen R. Adams. *Modular Grammars for Programming Language Prototyping.* PhD thesis, University of Southampton, March 1991.

[App85] Andrew Appel. *Compile-time evaluation and code generation for semantics-directed compilers.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1985.

[AS85] Hal Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press / McGraw Hill, 1985.

[BA81] J. Dean Brock and William B. Ackermann. *Scenarios: A Model of Non-Determinate Computation*, pages 252–259. Lecture Notes in Computer Science, number 107. Springer-Verlag, New York, 1981.

[Baw88] Alan Bawden. Reification without evaluation. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 342–351, Snowbird, Utah, July 1988.

[Bro83] Jarvis Dean Brock. *A Formal Model of Non-determinate Dataflow Computation.* PhD thesis, MIT Lab for Computer Science, August 1983. LCS-TR-309.

[Cli81] William D. Clinger. *Foundations of Actor Semantics.* PhD thesis, MIT Artificial Intelligence Lab, May 1981. AI-TR-633.

[Cli82] William D. Clinger. Non-deterministic call by need is neither lazy nor by name. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 226–234, 1982.

[CR90] Pavel Curtis and Jim Rauen. A module system for Scheme. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, pages 13–19, Nice, France, 1990.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Fil89] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, Computer Science Department, University of Copenhagen, August 1989. DIKU report 89/11.

[GB86] Cordell C. Greene and David R. Barstow. On program synthesis knowledge. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, chapter 26, pages 455–474. Morgan Kaufmann, Los Altos, CA, 1986.

[Gol85] Adele Goldberg. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, Reading, MA, 1985.

[Haa92] Kenneth Haase. A short description of FRAMER. Available by ftp from cecelia.media.mit.edu, January 1992.

[Hen82] Peter Henderson. *Functional Programming and its Applications.* Cambridge University Press, New York, 1982.

[HS88] Matthew Halfant and Gerald Jay Sussman. Abstraction in numerical methods. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 1–7, Snowbird, Utah, July 1988.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17–42. Addison-Wesley, Reading, MA, 1990.

[JF92] Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 48–58, Tama-City, Tokyo, November 1992.

[Kan86] Elaine Kant. On the efficient synthesis of efficent programs. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, chapter 8, pages 157–184. Morgan Kaufmann, Los Altos, CA, 1986.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 1–11, Tama-City, Tokyo, November 1992.

[Kos79] Paul Roman Kosinski. *Denotational Semantics of Determinate and Non-Determinate Dataflow Programs.* PhD thesis, MIT Lab for Computer Science, May 1979. LCS-TR-220.

[Lam88] John Lamping. A unified system of parametrization for programming languages. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 316–326, Snowbird, Utah, July 1988.

[Lee89] Peter Lee. *Realistic Compiler Generation.* MIT Press, Cambridge, MA, 1989.

[LK] John Lamping and Gregor Kiczales. A transfinite reflective tower. In a hack volume dedicated to Brian Smith on his 40th birthday.

[LKRR92] John Lamping, Gregor Kiczales, Luis Rodriguez, and Eric Ruf. An architecture for an open compiler. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 95–106, Tama-City, Tokyo, November 1992.

[Mac84] David MacQueen. Modules for standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, Austin, Texas, August 1984.

[McC63] John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems.* North-Holland, Amsterdam, 1963.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, CA, June 1989.

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

[Ous90] John K. Ousterhout. TCL: an embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, 1991.

[PS88] Prakash Panangaden and Vasant Shanbhogue. McCarthy's AMB cannot implement fair merge. Technical report, Computer Science Department, Cornell University, May 1988.

[Ree92] Jonathan Rees. Another module system for Scheme. To be published, January 1992.

[RK] Jonathan Rees and Richard Kelsey. Scheme48. Available by ftp from altdorf.ai.mit.edu.

[RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon*. Addison-Wesley, New York, 1992.

[Sch84] Richard Schooler. Partial evaluation as a means of language extensibility. Master's thesis, MIT Laboratory for Computer Science, August 1984. LCS-TR-324.

[SJ92] John Simmons and Stanley Jefferson. Language extension via first-class interpreters. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 59–59, Tama-City, Tokyo, November 1992.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 82–93, San Francisco, CA, June 1992.

[Tof90] Mads Tofte. *Compiler Generators*. EATCS monographs on theoretical computer science, volume 19. Springer Verlag, New York, 1990.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–242, October 1987.

[Wad87] Philip Wadler. A critique of Abelson and Sussman, or why calculating is better than scheming. *SIGPLAN Notices*, 22(3):83–94, March 1987.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, January 1992.

[Wan84] Mitchell Wand. A semantic prototyping system. In *Proceedings of the SIGPLAN 1984 Symposium on Compiler Construction*, pages 213–221, June 1984.

[Wei] Daniel Weise. Notes for a course on programming languages. Unpublished manuscript.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, June 1988.

[WS91] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 120–129, Toronto, Ontario, Canada, June 1991.

[Zab87] Ramin Zabih. Non-deterministic Lisp with dependency-directed backtracking. In *Proceedings of the 1987 AAAI Conference*, pages 59–64, Seattle, WA, October 1987.

[Zab88] Ramin Zabih. Dependency-directed backtracking in non-deterministic Scheme. Master's thesis, MIT Artificial Intelligence Lab, August 1988. AI-TR-956.

[Zip93] Richard Zippel. The Weyl computer algebra substrate. Available by ftp from cs.cornell.edu, January 1993.