MIT/LCS/TR-324

PARTIAL EVALUATION AS A MEANS OF LANGUAGE EXTENSIBILITY

Richard Schooler

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

This blank page was inserted to preserve pagination.

Partial Evaluation as a means of Language Extensibility

by Richard Schooler

Submitted to the
Department of Electrical Engineering and Computer Science
on 17 August 1984 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

An optimization technique known as partial evaluation is explored. A partial evaluator optimizes code by making use of static information about program values. Our partial evaluator is designed to optimize mainly applicative code. Un-checked assertions are used to identify applicative constructs in the input code and guide the partial evaluator. Side-effects in the input code are retained but are not optimized.

This thesis is part of a larger project devoted to language extensibility. Source language constructs will be transformed into kernel language constructs by syntactic transforms. These transforms will add applicative code for type-checking, etc. To assess the effectiveness of the partial evaluator, some examples of kernel language programs are partially evaluated, and the results discussed.

Thesis Supervisor: David K. Gifford, Asst. Prof.

Keywords: Programming Language Optimization, Implementation, and Extensibility; Partial Evaluation.

This work was funded by DARPA Contract No. N00014-83-k-0125.

Table of Contents

List of Figures

1. Introduction

1.1 Extended Abstract

In this thesis, an optimization technique known as *partial evaluation* is explored. Though the technique itself is not new, further research is worthwhile for a number of reasons:

1. While partial evaluation has been mentioned many times in the literature, it has not yet been fully explored. In particular, the application of partial evaluation to a lexically scoped language with full function values is new.

2. Partial evaluation has applications to automatic compiler generation. Theory indicates that partially evaluating an interpreter acting on a program results in compiled code by specializing the interpreter to the source program.

The primary optimization paradigm of partial evaluation is *specialization*: given a general function and some knowledge about the input, a specialized version can be automatically produced. Automatic specialization could have a large impact on software engineering (the art of production programming): how much easier it would be simply to maintain libraries of very general functions, such as generalized mapping and iteration constructs, which would then be automatically specialized to the particular input parameters by a partial evaluator!

Partial evaluation provides a clean conceptual framework for many traditional code optimizations [1]: in-line expansion, function specialization, constant propagation, loop unrolling, recursion elimination, etc. By expanding function bodies in-line, substituting the partially-specified actual parameters for the formal parameters, and then eliminating constant computations (or more generally, optimizing partially-determined computations), one achieves a very fine grain of optimization.

This thesis is part of a larger project devoted to language extensibility. One

problem with present-day production (non-research) languages is that they exhibit a serious performance difference between built-in and user-defined constructs. Programmers are forced to chose between well-structured programs that are inefficient because of the use of user-defined constructs, and efficient programs that are ill-structured because of the avoidance of user-defined constructs. The goal then, is to provide a truly *extensible* language, one that can be extended to apply easily and naturally to any problem domain, and one that is competitive in performance with a conventional, more specialized language.

1.2 Language Extensibility

Generally speaking, programming languages are equipped with a certain set of built-in constructs (built-in functions, types, etc.) and are extensible in that there exist facilities for the user to define additional constructs (user-defined functions, types, etc.). However, the built-in constructs usually have certain privileges, which include special syntax, more general semantics, and greater efficiency. The less distinction between built-in and user-defined constructs, the more extensible the language.

Achieving syntactic and semantic extensibility at the cost of pragmatic factors such as performance is uninteresting for a practical language. Pragmatic extensibility is achieved when user-defined constructs are roughly as efficient as built-in constructs. In a fully extensible language, while the constructs usually built-in to a conventional language might be somewhat less efficient, user-defined constructs would be considerably more efficient. The aggregate efficiency of a fully extensible language could thus surpass that of the conventional language.

1.2.1 Examples from Pascal

Pascal [40] exhibits the usual forms of extensibility, as well as the classic limitations. For example, there is a set of built-in functions, and a way of defining additional functions. Yet built-in functions are more powerful. Not only do they have

more general syntax (infix notation, variable number of arguments), but they have more general semantics, in that they are overloaded, i.e. performing different computations based on the argument types (e.g. the numeric functions).

Another example is Pascal's type system. A facility exists for introducing new types, but these are just type names, as they lack some fundamental features. Built-in types (such as **RECORD** and **ARRAY**) have special syntax for creation and selection, and more significantly, they have semantic advantages such as hidden representation and parameterization.

1.2.2 Improvements in CLU

CLU [24] represents a considerable improvement over Pascal. The generic and infix-syntax issues are addressed by *de-sugaring* (syntactically transforming) all the built-in operator symbols to standard functions. (E.g. **a + b** de-sugars to **T$plus(a,b)** where **T** is the type of **a**.) One cannot, however, define new infix operator symbols. The type system is also more extensible. The **CLUSTER** facility provides user-defined abstract types, which provide hidden representations, parameterization, etc. Yet some problems remain. Some built-in types have special constructor syntax or specialized constructs, which can hamper program modification, and prevent the user from defining similar types. Of course, in neither CLU nor Pascal can one significantly extend the type system (e.g., to add inheritance).

1.2.3 The Russell Type System

Russell [5, 11] was designed to illustrate the philosophy that data types should be full-fledged values, just as integers or arrays are. As a result, its type system is exceptionally flexible and extensible. Operations on types are provided that enable one to modify existing types in many ways to produce new types. Among other things, one can produce abstract types by controlling the export of operator names, and one can produce parameterized types by writing a function that accepts the parameter type and returns the desired type.

Russell's contribution to extensibility consists of elevating type implementations to first-class values. This enables one to use all the facilities of the language in order to define new types. Unfortunately, the facilities of the language aren't quite general enough for the user to be able to define constructs like the built-in **record** constructor, since this construct takes a variable number of arguments. Nor are they general enough to add significantly to Russell's type structure, adding new type constructors, for example.

1.2.4 Extensibility in EL1

In all the above languages, however consistent the syntax and semantics of various constructs are, the built-in constructs are more efficient. Given a compiler of average talent, an attempt to define records (for example) by the user can only result in a serious performance degradation. EL1 [38] attempts to deal with *all* the issues of extensibility (syntax, semantics, and pragmatics) by not only providing very general extension facilities, but also using a different execution paradigm: partial evaluation.

By making "type" a type, as does Russell, EL1 allows one the full power of the language to create new types. (It also shares Russell's limitations.) Operations on types also permit user-defined type-checking, type-coercion, and type-dispatching. At first sight all this user-defined machinery seems to imply a great deal of run-time overhead, as run-time type tags, type-checking, etc. appear to be necessary. EL1 provides two ways of eliminating this difficulty. First, routines can be compiled with a list of known values for certain of the free variables of the routine (including type values). The compiler will incorporate these values directly in the code, as well as executing at compile-time those functions which depend only on known values. Second, the compiler treats certain constructs specially. For example, the user-defined type-dispatching construct is recognized by the compiler, and the dispatch will be pre-computed, if possible, eliminating the type check and the necessity for type tags.

This approach to execution is generally known as *partial evaluation*: the execution of as much code as possible before run-time. In particular, any computation that depends on only known values can be eliminated and replaced by its known value (though side effects must be retained). Unfortunately, EL1 utilizes partial evaluation in a rather ad hoc fashion. Only certain constructs are automatically partially evaluated, and the manual specialization of functions by binding free variables is a rather clumsy solution to various problems such as parameterization.

1.3 Language Construction

Our proposed methodology is a generalization of the Russell and EL1 techniques: all extensions are implemented in the language, allowing the full power of the language to be used, and allowing full user access to the extension mechanism. In addition, partial evaluation will be used to optimize the code to the point where using the user-defined extension mechanisms is essentially free in terms of run-time performance. The base language suggested here incorporates just one basic semantic extension mechanism: the *closure*[1], or lexically-scoped function value.

An example of this technique is the implementation of a Smalltalk-style [16] message-passing type system. A Smalltalk object would be implemented as a closure whose code consisted of a dispatcher, and whose environment contained a method-table (table of operations), as well as a link to the superclass (parent type). Each method invocation would normally require the dispatcher to search (at run-time) for the method in the method-table, and then in the super-class's method-table, etc. Hopefully, the partial evaluator would frequently decide at partial evaluation time which method is to be invoked, and even expand the method's code in place of the invocation to achieve further optimization.

---

[1] A closure can be viewed as a pair consisting of a function's code and its definition-time environment, which results in free variables being lexically scoped, as in a block-structured language.

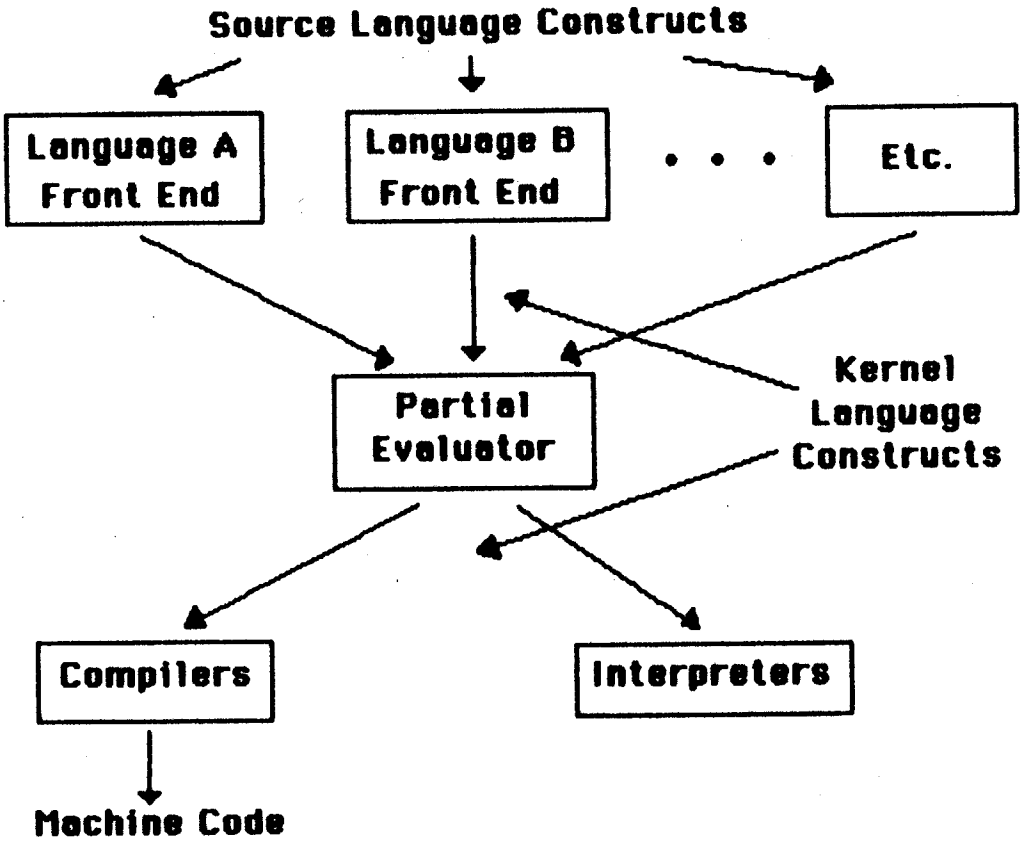

Figure 1-1: The Proposed Extensible Language System

At a higher level of abstraction, the system can be viewed as a compiler (see figure 1-1). The front end consists of the various transformations (macro-expansion, parsing, etc.) from the source language to the kernel language. The back end is a traditional implementation of the kernel language (an interpreter or a compiler). The partial evaluator acts on the kernel language and serves as a "middle end": a global machine- and source-independent optimizer [8].[2]

An advantage of the above approach to language construction over traditional compiler implementations is modularity. To re-source, one writes a new front end, which should not be time-consuming. To re-target, one changes the machine-dependent portions of the kernel language implementation, which should not be a

[2] [8] presents a machine-independent optimizer for U-Code - a low-level, stack-oriented intermediate language. Mainly conventional optimization techniques are used, though contributions are made in the area of code motion and machine independent register allocation.

great deal of work, since the kernel language is quite small and simple. The partial evaluator, which would probably be the most complex component, would remain constant.

Code optimization has been split into two major phases: the partial evaluator performs global control-flow optimization in a machine-independent fashion. Optionally, further local, machine-dependent optimizations can be performed by the kernel language implementation. This partitioning of the optimization process is shown to be useful in [8].

The application of partial evaluation to compilers is not entirely new: similar proposals have been reported in [19] and [7] which present IBM's General-Purpose Optimizing Compiler project. Their compilation strategy is to transform source code into machine code through a series of intermediate languages. At each stage, each language construct has one defining procedure in the next lower-level language. Instead of special-casing, a partial-evaluation-like approach is used to optimize at each step by taking advantage of constraints on program values.

1.4 The Remainder of the Thesis

This thesis is devoted to the middle component of the proposed extensible language system: the partial evaluator. The code which the partial evaluator acts on will be generated by syntactic transforms from surface language constructs. The generated code will preserve all user-specified side-effects but will also include applicative constructs for type-checking, etc. We anticipate that applicative interpreters will be used to implement certain source language constructs. The partial evaluator is designed to optimize this added applicative code while retaining all side-effects.

First the Imagine Base Language (which serves as the kernel language) will be presented. This very small language has been designed to provide an adequate

implementation base for the proposed extension mechanisms. After various design issues are considered, a language definition is given.

Partial evaluation is then defined in more detail, and some related research is explored. To meet our particular needs, our own partial evaluator differs from previous designs. The detailed design of the partial evaluator is presented by first discussing the structure of the program, and then considering the partial evaluation of each of the various constructs of the kernel language.

As an illustration of the proposed technique of language extension, and as an extended example of partial evaluation at work, the implementation of a simple message-passing system in IBL will be described. Programs using this user-defined package will be partially evaluated and the results discussed. Another example of language extension, and a further demonstration of the power of partial evaluation is presented next. An interpreter for a simple declarative language is implemented, and the partial evaluator is used as a compiler to eliminate all unnecessary run-time interpretation.

Finally, the limitations and further prospects of the above techniques are considered.

2. The Imagine Base Language

2.1 Overview

In this chapter, the Imagine Base Language (IBL) is presented. This small kernel language is designed to serve as an implementation language for user-defined extension mechanisms. It is a lexically-scoped dialect of Lisp [39]. After discussing various design considerations, the various code forms and built-in types are presented. IBL closely resembles Scheme [33], with the addition of a user-extensible parsing mechanism and self-evaluating keywords.

2.2 Design Considerations

Imagine Base Language (IBL) is a small "kernel" language which serves as an implementation language for the various source languages. In other words, it is not intended that users program directly in IBL, but rather that source programs are transformed into IBL programs through syntactic transformations. Thus certain standard design criteria, such as readability and type-safety, are less important in IBL than in a typical language. IBL is based on Lisp [39] mainly because of the program-data equivalence offered by that language (which eases program analysis and manipulation), but also to take advantage of the development environment offered by ZetaLisp [26] running on the Symbolics 3600 Lisp Machine [36].

IBL is a small language (for ease of implementation and analysis) but complete in the sense that it provides all the constructs necessary to efficiently support source language constructs. Such implementation types as words and word-vectors are supported, as are other convenient types such as pairs (Lisp cons-cells). The control structure must also be complete in this sense, yet it must also be extremely simple for the purposes of analysis and optimization.

The main primitive control structure is the function call. IBL functions are actually lexically-scoped closures and are first-class values, so the function call

mechanism subsumes such traditional constructs as "own" variables, co-routines, etc. Reynolds [27] shows that a lexically-scoped language with function values is sufficiently general to implement such constructs as program jumps and mutable state (in an applicative languagel).

The astute reader may notice that no iteration construct is provided. Steele and Sussman have shown that tail-recursion can be implemented as efficiently as iteration, both in space and in time, and both in an interpreter [30, 31, 32, 35] and a compiler [34].

While a purely applicative language would have been much easier to analyze, it was decided that this would have been an over-simplification for practical purposes, and thus imperative features such as assignment and structure mutation have been included in IBL.

Several features of traditional Lisp dialects have been omitted. For example, users are not allowed to define new special forms, since each special form must be handled individually by the partial evaluator. "All-powerful" functions such as **eval** and **set** limit the usefulness of analysis, since they can invalidate almost any assumption about variable values, mutability, and other side effects. For this reason, they have not been included in IBL. Since the program jump (goto) greatly complicates analysis, it too has been eliminated. IBL has no exception-handling mechanism, which while an over-simplification in a real language, simplifies the analysis for the purpose of this thesis.

Since the source language will implement all type-checking, generic operators and other complicated constructs, IBL is a bare-bones language with minimal run-time overhead. It should be possible to compile IBL to run very efficiently on conventional architectures.

IBL is clearly very similar to Scheme [33]. This is not accidental, since Scheme

provides much of the desired qualities: a small, lexically-scoped language with program-data equivalence.

2.3 Description and Semantics

IBL (Imagine Base Language) is designed to serve as the kernel language of the extensible language system. In surface syntax and in semantics, IBL greatly resembles Scheme. IBL is a lexically-scoped dialect of Lisp with self-evaluating keywords. A user-extensible syntax mechanism has been included so that syntactic sugar can be introduced.

IBL is an applicative-order language. In other words, all arguments to a function are always evaluated. However, the order of argument evaluation is undefined. Arguments may be evaluated left-to-right, right-to-left, or in any other order. Arguments are passed by *sharing*, as in Lisp. Call-by-sharing resembles call-by-reference for mutable values, since an argument may be mutated by the called function, but resembles call-by-value for immutable values.

IBL is implemented by parsing the surface syntax, which consists of S-expressions (the Lisp surface syntax of nested lists of atoms), into an internal form known as I-code (internal code). This parsing is done by a set of built-in parsing functions, augmented by user-defined functions which can transform arbitrary user syntax into syntax understood by the built-in parser. These user-defined parsing functions are "reader macros," since they act when a program is "read in", or parsed.

The IBL interpreter acts only on I-code structures. Values returned by the evaluation of an expression are unparsed before printing. The main loop of the interpreter is thus READ-PARSE-EVAL-UNPARSE-PRINT rather than the usual READ-EVAL-PRINT (see figure 2-1).

IBL is a weakly-typed language in that data carry their type with them, but no

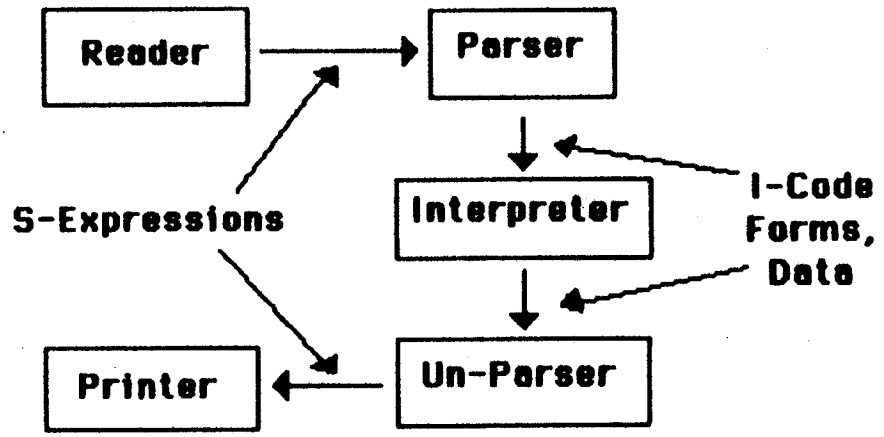

Figure 2-1: The Interpreter Loop

type-checking is done. The usual complement of built-in types is provided: fixnums (small integers), booleans, pairs (cons-cells), vectors, lambdas (user-defined functions), etc. No type extension facilities are provided, since it is assumed that IBL types serve mainly as implementation types for the source languages, whose type systems are implemented in IBL.

2.4 I-code and Syntax

2.4.1 Code Forms

Internally, an IBL program is a tree with the following types of nodes:

- **definition** - consists of a *name*, a *value*, and a *read-only* boolean flag. When executed, evaluates *value* and binds *name* to the result in the current environment. The *read-only* flag is used by the partial evaluator and is described later. The built-in syntax (**def** *name value read-only*) creates a definition.

- **lambda-form** - consists of a list of *required* parameters, a *rest*

parameter, a list of *assertions*, and a *body*. When executed, creates a *lambda* (see section 2.5.1). The built-in syntax **(lambda** (*required* . *rest*) **(assert** *assertions*) *body*), where the **assert** clause is optional, creates a lambda-form. (See section 3.8 for the form of an **assert** clause.)

- **conditional** - consists of a *predicate*, a *consequent*, and an *alternative*. When executed, first evaluates the *predicate*. If the *predicate* evaluates to true, the *consequent* is evaluated, otherwise the *alternative* is evaluated. The built-in syntax **(if** *predicate consequent alternative*) creates a conditional.

- **sequence** - consists of a list of *expressions*. When executed, the *expressions* are evaluated sequentially, and the value of the last expression is returned. The built-in syntax **(sequence** *expressions*) creates a sequence.

- **combination** - consists of a *function* and a list of *arguments*. When executed, the *function* and the *arguments* are all evaluated, then the *function* is applied to the *arguments*. The built-in syntax (*function arguments*) creates a combination. To avoid ambiguity, *function* cannot be a symbol which denotes a built-in syntax rule (**if**, **sequence**, etc.).

- **variable reference** - consists of three components: a *name*, a *closure*, and a *frame-count*. When executed, the *name* is looked up in the environment of the *closure*, skipping *frame-count* environment frames[3]. If *closure* is nil, the *name* is looked up in the current lexical environment (the usual case). The built-in syntax **(variable** *name closure frame-count*) creates a variable reference. For convenience, a symbol is parsed as a variable reference with *closure* **nil** and a *frame-count* of zero.[4]

- **assignment** - consists of four components: a *name*, a *closure*, a *frame-count*, and a *value*. When executed, the *name* is looked up in the

---

[3]The *closure* component is a lambda in whose environment the variable is scoped. An environment frame corresponds to a lexical scoping level, i.e. a function definition nesting level. Exposing lexical levels unfortunately constrains and exposes the implementation of environments to some extent.

[4]Note that the user should always use the latter form of specifying a variable reference, so that *closure* is nil. Only the partial evaluator should create variable references with non-nil *closure*. *Frame-count* is used both to optimize variable lookup, and by the partial evaluator to dis-ambiguate different variables with the same name. The user need never specify a non-zero *frame-count*.

environment of the *closure*, skipping *frame-count* frames. As above, when the *closure* is nil, the *name* is looked up in the current lexical environment. *Name* is then rebound to *value*, which is the result of the assignment. The built-in syntax **(assign** *name contour frame-count value*) creates an assignment.

The leaves of the tree are values of the various built-in types, which are covered in section 2.5.1.

2.4.2 Other Syntax

Another useful built-in syntax rule is **(quote** *literal*). While most literals are self-parsing, such as strings and fixnums, others are not. For example, a list (generally) parses to a combination and a symbol (again, generally) parses to a variable. To denote list or symbol literals, one uses the **quote** form, which when parsed, simply returns its argument unchanged. Note that no quote-form is needed in I-code, since a list-literal or a symbol can not be mistaken for executable code at the I-code level.

The last built-in syntax rule is **(user-syntax** *variable-list body*). When executed, this expression creates a user-defined syntax rule. As an example of the use of such a rule, consider the implementation of the usual Scheme syntax for defining a function:

```
(define (cadr x)
        (car (cdr x)))
```

The above syntax will be transformed into the following primitive syntax by the parser:

```
(def cadr                ;; name
     (lambda (x)         ;; value
        (car (cdr x)))
     true)               ;; read-only?
```

The appropriate syntax-rule definition which implements the above transformation is:

```
(def define (user-syntax (form)
                         (list 'def (car (car form))
                               (cons 'lambda
                                     (cons (cdr (car form))
```

```
                                    (cdr form)))
                            'true))
    true)
```

The single-quote character is the usual abbreviation for the **quote** form. The user-syntax function is passed exactly one argument, which is the **cdr**, or tail, of the input S-expression.

The syntax mechanism works as follows: if the input S-expression is a list, the first element of the list is examined. If this is a symbol and the symbol denotes a built-in syntax rule (**def**, **lambda**, etc.), that rule is applied. If the symbol denotes a user-defined syntax rule, the rule is applied to the tail of the list, and the result is recursively parsed. If the head of the list does not denote a syntax rule, each element is recursively parsed and a combination is created. Atomic expressions are parsed specially, as explained in the next section. (For a more sophisticated example of a user-defined syntax rule, see appendix II.)

2.5 Built-In Types

The following sections describe the built-in data types of IBL. Since the user cannot define new types, all IBL objects are of one of the following types. As mentioned above, source language type systems will be built on top of these basic implementation types.

2.5.1 Lambdas

The most important type is the *lambda*, or user-defined function. There is only one way to create a lambda, and that is by executing a lambda-form. A lambda is a first-class IBL value and can thus be an argument to a function and the result of a function. Since a lambda contains its definition-time environment, free variables are lexically scoped, so a lambda is actually a full closure.

The Scheme "dotted-tail" notation is used for formal parameter lists to indicate *required* and *rest* parameters. An example of its use is: **((lambda (a b . d) (print d)) 1 2 3 4 5 6)**. As usual, **a** is bound to **1**, and **b** is bound to **2**, but **d** is bound to the list of remaining arguments, i.e. **(3 4 5 6)**.

The function **list** which creates and returns a list of its inputs can thus be very simply defined by:

```
(define  (list  . x)
  x)
```

2.5.2 Built-In Functions

A *built-in function* is very much like a lambda, except that it is part of the implementation of IBL. When IBL starts, certain variables are bound to the built-in functions in the global environment. Most of the built-in functions are described below together with the type of data with which they are associated. The remainder of the functions are described here.

The function (**type** *value*) returns a keyword (see section 2.5.9) describing the type of *value*, either **lambda:**, **built-in-func:**, **pair:**, **nil:**, **fixnum:**, **boolean:**, **vector:**, **string:**, **symbol:**, **keyword:**, **built-in-syntax:**, or **user-syntax:**. This function can be used to implement type-checking and -dispatching.

Various I/O functions are provided:

- (**print** *value*) - Unparses a value and prints it as an S-expression; returns the value.
- (**read**) - Reads a value as an S-expression.
- (**load** *file-name*) - Evaluates the contents of the file denoted by the string *file-name*; returns nil.
- (**save** *file-name*) - Saves the global environment to the file denoted by the string *file-name* in a format compatible with **load**; returns nil.

A basic pointer-equality predicate is provided: (**eq?** *a b*), which returns a boolean (true if *a* and *b* evaluate to the same object, otherwise false).

The following two functions aid in the implementation of user-defined syntax rules:

- **(apply** *function arg-list*) - The function is applied to the list of arguments, e.g. **(apply iplus '(1 2))** (which returns **3**).

- **(symeval** *symbol*) - The argument is evaluated and then looked up as a variable in the current environment, e.g. **((symeval 'iplus) 1 2)** (which also returns **3**).

The following functions provide an interface to the partial evaluator:

- **(pe-exp** *form*) - Partially evaluates the *form* and returns the result in readable format.

- **(pe-func** *func*) - Partially evaluates the body of *func* and as a side-effect, replaces the body of *func* with the optimized version. Returns the optimized body.

- **(pe-all)** - Performs **pe-func** on every **lambda** in the global environment; returns nil.

2.5.3 Fixnums

A *fixnum* is a machine word, which can be used to model small integers, among other things. The usual built-in operations are provided: **iplus, iminus, itimes, idiv, irem, iless?, igreater?,** and **iequal?.**

2.5.4 Pairs and Nil

A *pair* is simply a pair of slots. The following operations are provided on pairs:

- **(car** *pair*) - Returns the first element of the pair.

- **(cdr** *pair*) - Returns the second element of the pair.

- **(cons** *a b*) - Creates and returns a new pair with components *a* and *b*.

- **(set!-car** *pair new-car*) - Updates the first component of the pair; returns the updated pair.

- **(set!-cdr** *pair new-cdr*) - Updates the second component of the pair; returns the updated pair.

Nil is a special value which indicates an empty list. The symbol `nil` is parsed as nil. Note that nil is not a pair, one cannot take the **car** or **cdr** of nil.

2.5.5 Booleans

Traditional Lisps use nil for "false" and anything else for "true". IBL instead has a true *boolean* type. The symbol `true` parses to the boolean true value, and the symbol `false` parses to the boolean false value. All predicates (generally speaking, those functions ending with a "?") return a boolean value and conditionals expect a boolean result from their predicates. The following functions operate on boolean values (note that all arguments are always evaluated):

- **(and** *a b***)** - Boolean AND.
- **(or** *a b***)** - Boolean OR.
- **(not** *a***)** - Boolean NOT.

2.5.6 Vectors

A *vector* is a heterogeneous linear array of values. The following operations are provided:

- **(vector-cons** *size initial-value***)** - Creates and returns a new vector with *size* elements, all initialized to *initial-value*. The vector is zero-based.
- **(vector-ref** *vector slot***)** - Returns the component of *vector* denoted by the fixnum *slot*.
- **(vector-set!** *vector slot new-value***)** - Updates the component of *vector* denoted by *slot* with *new-value*; returns the updated vector.
- **(vector-size** *vector***)** - Returns the number of slots in the *vector*.

2.5.7 Strings

A *string* is a linear array of characters. No operations are currently defined on strings (except the generic functions such as **print**, **type**, etc.)

2.5.8 Symbols

A *symbol* is simply a name. A symbol is parsed as a variable or other value unless the **quote** form is used. For example, **(quote true)** is parsed as the symbol **true**, whereas **true** is parsed as the boolean true value.

2.5.9 Keywords

A *keyword* is a symbol whose last character is either ":" or ".", e.g. **keyword:**. Keywords evaluate to themselves, and thus do not need to be quoted. Because they can never represent variables, they cannot be assigned. Keywords, which can be used in implementing message-passing programs, are also useful for describing a small set of options.

2.6 Utilities

The extensible syntax mechanism has been used to implement a subset of Scheme in IBL. Since the two languages are so similar, only some simple syntactic transformations are necessary. The resulting extended language lends itself to programming much more than "bare" IBL.

Among the constructs provided are useful abbreviations such as **cadr**, **cddr**, **first** and **second**; type-checking functions such as **fixnum?** and **lambda?**; and more complex syntax such as **cond** for multi-branch conditionals, **let** for local variable binding, **setq** for assignment, and back-quote syntax for easy macro-creation. (Appendix I contains a complete listing.)

2.7 Summary

IBL is a lexically-scoped dialect of Lisp resembling Scheme. The list-oriented surface syntax is parsed to internal code forms. The parsing mechanism is user-extensible to allow syntactic sugar to be introduced. Self-evaluating keywords are introduced to aid in the implementation of message-passing objects. A utility package has been written which implements a useful subset of Scheme.

3. The Partial Evaluator

3.1 Overview

The partial evaluator is presented in this chapter. Partial evaluation is described, and some related work is discussed. The design of the partial evaluator is first outlined, and then presented in detail by considering the partial evaluation of each of the various constructs of IBL. The difficulties caused by full closures and side effects such as mutation and assignment are explored.

3.2 Introduction

As mentioned previously, a partial evaluator is an automatic specializer. An example is helpful to clarify the concept. Consider the **printf** function of the C language [21]. This function takes a control string and a number of arguments. The control string specifies how the remaining arguments are to be printed, e.g. **printf ("%d %f", a, b)** will print **a** as a decimal integer and **b** as a floating point number. In normal execution, the control string is interpreted at run-time by the **printf** routine.

It is likely that a partial evaluator could specialize the above invocation of **printf** to:

```
printdecimal    (a);
printspace      ();
printfloat      (b);
```

The partial evaluator has in a sense compiled the control string interpreter to produce optimized code.

Beckman et al. [4] give a more formal description of partial evaluation which I paraphrase here. Imagine that we have a program P which takes two parameters X and Y. In general we will want to run P with many different values of X and Y. It is likely that in some cases one of the arguments, say X, is known. A possibility then exists of replacing the call to the general procedure P by a version which is

optimized by replacing references to X by its known value, performing computations which depend only on constants, and eliminating code which will not be executed. This optimization could be done by hand, a tedious and error-prone procedure in general, or we could supply another program R that would accept P and a value for X and output an optimized program $R(P,X) = P_X$ such that $P_X(Y) = P(X,Y)$. Presumably P_X would be more efficient than P. The program R is then a partial evaluator.

3.3 Related Work

Partial evaluation is a technique that has appeared in the literature many times. Other names have been used: "symbolic evaluation", "symbolic execution", "symbolic interpretation", "partial execution", etc. The earliest reference known to the author is in Lombardi and Raphael's "Lisp as the Language for an Incremental Compiler" [25], from 1964. The literature on automatic programming contains some examples of partial evaluation techniques, such as Darlington and Burstall's work [10]. Partial evaluation ideas have been used in other automatic specializers. For example, Scherlis [29] generates *expression procedures* (syntactic transformation rules) from existing definitions using a small set of basic operations. These operations correspond to beta-expansion, closed-specialization, and variable elimination. Code is specialized by choosing (in an un-specified manner) from the set of syntactic transformations that have been generated by the basic operations. Deutsch [12] used partial evaluation to simplify theorems in a program verifier. By propagating partially-specified values and folding expressions, many theorems were reduced to the boolean constants and were thus "proved" (or disproved) without the use of the theorem prover.

The fundamental tension in any optimization strategy, including partial evaluation, is between increased performance (wanting to improve the code as much as possible) and correctness (preserving program semantics). Such features as assignment of variables and mutation of values complicate the partial evaluator's

task, but seem highly desirable in a production environment. By formally defining an abstract machine for Prolog, and then extending the abstract machine to do partial evaluation while proving that the changes preserve the original semantics, Komorowski [22, 23] has described a partial evaluator for Prolog that is provably correct. However, Prolog is such a simple language (no assignment, nested definitions, etc.) that this approach doesn't seem practical for more powerful languages with more intractable semantics. About the best one can do at the moment is to argue carefully (and hopefully convincingly), albeit informally, that program semantics are indeed preserved for each transformation.

Ershov [13, 14, 15] calls partial evaluation "mixed computation". His model is that a partial evaluator works by interpreting the code, doing what computation can be done as soon as possible, and emitting code to perform the portion of the computation that must be done at run-time. The partially-determined environment, or partial-evaluation-time environment, is constructed in precisely the same way as the interpreter builds the fully-specified run-time environment. Since an interpreter is simple to write, and the partial evaluator can share the structure of the interpreter, and indeed much of the code, this implementation approach seems simpler and more powerful than the general program-transformation model.

Ershov's papers are largely theoretical, in that the emphasis is more on the definition and application of partial evaluation rather than on practical algorithms to implement partial evaluation. Ershov mentions many possible applications of partial evaluation, including automatic specialization of generic code to produce efficient, tailored code, and the partial evaluation of an interpreter (or operational semantics) acting on a source program to produce compiled code.

Haraldsson's work [17, 18] is the most relevant to the project at hand. He describes a simple partial evaluator known as REDFUN (first described in [4]), and discusses its various features and failings. Drawing on the REDFUN experience, he

then presents the REDFUN-2 system. Haraldsson explores at length many of the same issues that this thesis must confront, such as partially-specified values, handling of side-effects, etc. His emphasis is of course somewhat different, concentrating more on specialization of functions and general optimization than language extensibility.

Haraldsson gives a very complete description of a working partial evaluator for InterLisp [37]. His work has a number of shortcomings (at least from our point of view). The design is essentially that of a program manipulator, not that of an augmented interpreter. Many of his optimizations are in the form of syntactic transformations. This complicates his algorithms considerably, since it is harder to deal with semantic difficulties such as arguments with side-effects. Although InterLisp is dynamically-scoped by default, it does contain *funargs*, which are lexically-scoped closures. Haraldsson, though, does not deal with the "disappearing environment" problem of beta-expanding closures (see section 3.5.2.2). Nor does he deal with the more common problem of variable shadowing in beta-expansion. He over-simplifies the issue of the correct code for an evaluated result, simply using the **quote** special form, which is incorrect in the presence of mutation (see 3.5.3).

On the other hand, he is more ambitious in some areas. His partially-specified values contain more information, such as lists of possible values, lists of impossible values, datatypes, etc. He attempts to deal with all the (sometimes baroque) features of InterLisp, such as jumps, **eval**, **setq**, etc. He performs such optimizations as deleting unnecessary variables and closed specialization (although he doesn't go into detail on the latter). He examines conditionals more carefully, extracting information from the predicate so that while optimizing the true-branch he can use whatever information can be deduced from the predicate evaluating to true and similarly for the false-branch. Of course, his program is almost ten times as big as our implementation.

Good optimization of side-effects requires a rather different approach. For example, Cocke [9] keeps track of the program state (memory contents, etc.) in order to detect and eliminate redundant side-effects. By noting where the program state returns to a previous state, Cocke detects where a simple jump can replace more expensive constructs.

3.4 Design

In our application, the partial evaluator is intended to work on code that has been generated by transformations from some surface syntax. The generated code will preserve all side-effects specified by the source constructs, but will also include applicative constructs for type-checking, etc. The partial evaluator is designed to optimize the added applicative code while retaining all side-effects. The partial evaluator is *not* intended to optimize general user-written code.

We envision that one of the main applications of the partial evaluator will be performing type computations and specializing polymorphic functions. By including type-checking code and eliminating it wherever possible, we hope to capture static and dynamic type-checking within a single framework.

A partial evaluator could perform an arbitrary amount of work, including sophisticated dataflow analysis, constraint propagation, and theorem proving, in order to accumulate information to aid in code optimization. Another option is to have the user supply declarations or assertions. These can either be checked for consistency or not. For simplicity's sake, we assume that the code generated by the above-mentioned front ends will contain automatically generated assertions that are guaranteed to be consistent. These unchecked assertions will be used to guide the partial evaluation process.

Among the assertions that have proven useful are:

- **evaluate** - to indicate that a function invocation is applicative and should be evaluated before run-time if possible;

- **beta-expand** - to indicate that a function invocation should be beta-expanded (expanded inline) if possible;

- **read-only** - to indicate that a variable will not be re-bound (e.g. by assignment) and that the bound value will not be mutated;

For a complete list of assertions, see section 3.8. Each assertion will be fully described in the following sections.

Following Ershov's model, the partial evaluator is designed as an augmented interpreter. An interpreter is often defined as an expression evaluator with the following signature:

eval-exp: exp × env → value

(ignoring side effects). An expression (**exp**) is evaluated with respect to an environment (**env**) and returns a result (**value**). Similarly, an expression-partial-evaluator has the following signature:

pe-exp: exp × before-env × after-env → value × code

An expression (**exp**) is partially evaluated with respect to the partially-determined environments (**before-env**, **after-env**) and returns a partially-specified result (**value**) and the optimized expression which will produce the fully-specified result at run-time (**code**). These last two components are collectively called a *partial*. As will be seen later, a partial also contains other components.

Two environments are necessary because partial evaluation may change the environment in which an expression is evaluated. For example, in normal evaluation a function body is evaluated in the function's definition-time environment augmented by the formal parameter bindings. However, if the function is *beta-expanded* (expanded inline), the body must be evaluated in the invocation-time environment. **Before-env** corresponds to the original or pre-partial-evaluation environment and is used to look up variable values. **After-env** corresponds to the post-partial-evaluation environment and is used to calculate the code for a variable reference. (See section 3.6 for more details.)

In order to denote partially-specified values, the value space of the interpreter must be augmented with a distinguished "unknown" value, which will be printed here as ***unknown***.[5] More complex partially-specified values are constructed using the standard combining forms such as pairs and vectors. For example, **(1 . *unknown*)** denotes a pair whose **car** is 1 and whose **cdr** is unknown.

3.5 Partially Evaluating Invocations

Function invocation represents a fertile area for optimization. Constant propagation can replace an invocation by the partial-evaluation-time computed result. Beta-expansion can replace an invocation by the function's body expanded in-line with the arguments substituted for the formal parameters. Specialization can replace a call to a generic function by a call to a specialized version of the function. In the following two sections, the partial evaluation of built-in and user-defined function invocations is discussed.

3.5.1 Built-In Functions

The only way to optimize a built-in function invocation is to replace the invocation by its value. Beta-expansion is inapplicable since there is no function body, and specialization is impossible since there are no "generic" built-in functions in IBL (this might, of course, not hold for other languages). The easiest way to replace an invocation by its value is to evaluate it at partial-evaluation-time. For example, **(iplus 1 2)** can be replaced by **3**. Evaluation is possible only if the function is applicative and all the arguments are both known and applicative. Since these are very restrictive conditions, an optimizer cannot stop there.

If the function has side effects, the invocation cannot simply be evaluated. Any side-effects must be retained, but it may still be possible to return a value. For example, the imperative function **print** is defined to return its argument. While a call

---

[5] Actually, unknown values are typed, e.g. one can have ***unknown fixnum*** or ***unknown keyword***. An unknown value with unknown type is represented as just ***unknown***.

to **(print 1)** cannot be replaced by the result 1, the partial evaluator can note that the value of this expression is 1.

If the function is applicative, but the arguments are not, all side-effects must be retained, though it may still be possible to do some optimization and return a value. As an example, the expression **(iplus (print 3) 1)** is replaced by **(sequence (print 3) 4)**, which has the value 4. The partial evaluation of an expression (in this implementation) also returns a list of side-effect forms, which aids in the cascading of such expressions. For example, **(iplus (print 3) (itimes 4 (print 5)))** is replaced by **(sequence (print 3) (print 5) 23)**. Note that all applicative arguments have been eliminated. Note also that IBL does not guarantee the order of argument evaluation, so **(iplus (print 1) (print 2))** might well be replaced by **(sequence (print 2) (print 1) 3)**.

If not all the arguments are known, it may still be possible to return a value. For example, if **a** is bound to a partial with value **(1 . *unknown fixnum*)**, **(car a)** could be replaced by **1**. Though nothing could be done with **(cdr a)**, **(type (cdr a))** could be replaced by **fixnum:**. Each built-in function must have a distinct partial evaluation routine in order to take advantage of any special characteristics of the arguments.

The interaction of *constructor* functions (such as **cons**) and *accessor* functions (such as **car** and **cdr**) deserves special attention. Consider **(car (cons a b))**, where neither **a** nor **b** is known. It would be desirable to replace this form with just **a**, but it appears that the partial evaluation of **(cons a b)** (which will return a partial with value (***unknown . unknown***) and code **(cons a b)**) does not return enough information. While some optimizers would use syntactic transformations in this situation, a more general solution is preferred, since more difficult situations arise. For instance, **b** could be a form with side-effects.

Our current solution is for the partial evaluation of **(cons a b)** to return not

only the value and code, but also a *partial-code* component, which is essentially a pair of two partials: one each for the **car** and **cdr** of the form.[6] The partial-code can be (and usually is) nil, indicating that no partial-code information exists. The partial evaluation of rest parameters and list literals also uses the partial-code mechanism. The partial-code mechanism could be extended very simply to include vectors, by allowing a vector of partials to appear as the partial-code component.

The partial-code component is created during the partial evaluation of an invocation of **cons** (for example), and is used in the partial evaluation of invocations of **car** and **cdr**. If the argument has a non-nil partial-code component, the appropriate part of the partial-code component can be returned as the result of the partial evaluation.

3.5.2 User-Defined Functions

Optimizing a built-in function invocation is simply a matter of replacing an invocation by the value that it will return (while preserving side-effects). The optimization of user-defined function invocations is more interesting, as there are usually a number of different optimization techniques available. Following Haraldsson's lead, we have the following choices:

1. evaluation - The invocation is evaluated and replaced by the result.

2. beta-expansion - The invocation is replaced by the function body, with the actual arguments substituted for the formal parameters. The body is specialized to the actual arguments.

3. open specialization - A lambda-form is used to create an anonymous, specialized function which replaces the original function in the invocation.

---

[6] Actually, instead of a pair of partials, the current implementation uses a pair of nullary closures (essentially delays), which when invoked create the appropriate partial. The motivation is to reduce construction of unneeded structure, as the partial corresponding to a component is only created if that component in needed.

4. closed specialization - A new function is defined that is a specialized version of the original. The original invocation is replaced by an invocation of the specialized version.

The final choice is of course to do nothing and simply return the invocation as is. Even if no optimization is done, it may still be possible to return some information by including a (**returns** *type-keyword*) form in the **assert** clause to indicate that the function returns a certain type of value.

We can illustrate these options more fully by considering an example:

```
(define   (foo  a b)
  (iplus   a b))
```

Consider the partial evaluation of **(foo 1 (print x))** where **x** is unknown. One of the following could occur:

- Using beta-expansion would result in **(iplus 1 (print x))**.
- Using open specialization would result in **((lambda (b) (iplus 1 b)) (print x))**.
- Closed specialization would result in **(foo1 (print x))** where **foo1** is defined as:

```
(define   (foo1   b)
  (iplus   1 b))
```

The following sections consider the above optimizations in more detail. The conditions under which they are applicable are discussed, and examples are given of their use.

3.5.2.1 Evaluation

Evaluation is only performed if the arguments are known (fully-specified) and the function is applicative. (Side-effects of the arguments can be taken care of by using a sequence form, as in section 3.5.1.) Even if evaluation is possible, it may not always be desirable. For example, the function may be extremely expensive, computationally or otherwise. Evaluation is controlled by including the symbol **evaluate** in the **assert** clause of the function definition.

3.5.2.2 Beta-Expansion

Beta-expansion is a very useful transformation, since it eliminates a function call. Even if function calls aren't particularly expensive, beta-expansion improves the quality of optimization, since removing the function call interface allows more specialization. Beta-expansion is not universally applicable, however. One problem is that code size can increase, since beta-expansion implies giving up the code-sharing advantages of function calls. Recursive functions demonstrate this problem at its extreme. Code size can also increase in the process of substituting actual arguments for formal parameters. If a single formal parameter is referenced many times, the code of the actual argument will be repeated. If this argument code has side-effects, the side-effects would also be repeated, thus changing the program semantics. For this reason, beta-expansion is usually restricted to invocations with only applicative arguments.[7] Scheifler [28] discusses inline expansion in some detail.

Including the symbol **beta-expand** in the **assert** clause of a function definition will cause all invocations of this function with all-applicative arguments to be beta-expanded. Conditional beta-expansion can be achieved by using instead (**beta-expand** *expression*), in which case an invocation will only be expanded when the the value of the *expression* is known at partial evaluation time (and all the arguments are applicative), e.g. in a recursive definition, to prevent unbounded beta-expansion. This mechanism is more powerful than it may appear at first glance. Consider `(beta-expand (type b))` where `b` is a parameter of the function, which indicates beta-expansion only where the type of `b` is known.

Including the symbol **macro-like** in the **assert** clause will cause unconditional beta-expansion, which is useful for "ideal" functions that mention each formal parameter exactly once, e.g.

[7] Another option is to introduce local variables to hold the arguments, eliminating the above problems. Unfortunately, these local definitions would tend to accumulate in recursive beta-expansion, and the partial evaluator does not eliminate dead (un-used) variables at the moment.

```
(define (caddr x)
        (assert macro-like (read-only x))
  (car (cdr (cdr x))))
```

Beta-expansion is more than just replacing an invocation by the definition with the arguments appropriately substituted for the formal parameters: we wish to specialize the code to take advantage of any partial knowledge concerning the arguments or the environment. The above formulation of the partial evaluator makes this quite simple: the definition is partially evaluated as an expression in the after-env with the formal parameters bound to constructs indicating substitution.

Substitution has to deal with the problem of name conflict. Consider for example **(foo a)** where **foo** is defined by:

```
(define (foo b)
  (let ((a (baz b)))
    (iplus a b)))
```

If **b** is replaced by the code **a** in **(iplus a b)**, then **a** is *caught* by the **a** introduced in the **let** form. In the lambda calculus (which is lexically scoped) this problem is handled by re-naming inner variables so as not to conflict with outer-level names. Another solution is to extend variable names with *traversal paths*, which are constructs which indicate where in the environment a value is to be found, i.e. counts of how many environment frames (lexical levels) to traverse before looking up the name. Traversal paths contain enough information to dis-ambiguate distinct variable references with the same name. In the above example, **(foo a)** will partially evaluate to:

```
(let ((a (baz (variable a nil 0))))
  (iplus (variable a nil 0) (variable a nil 1)))
```

Actually **(variable a nil 0)** would be unparsed as **a** since the variable form indicates a name to be looked up in the current environment (closure component is nil), skipping no frames (frame-count component 0). Traversal paths are thus both an optimization of variable lookup and an extended naming mechanism.

IBL functions can take a *rest* parameter, which is the list of remaining arguments after the required arguments. Substituting a rest parameter during beta-expansion requires a bit more work. Consider the following function:

```
(define (foo . lst)
        (assert beta-expand (read-only lst))
  (sequence (print lst)
            (print (* (first lst) (second lst)))))
```

If the expression **(foo 1 2)** is beta-expanded, **(print lst)** should be replaced by **(print (cons 1 (cons 2 nil)))**, but **(first lst)** should be replaced by **1**. The partial-code mechanism mentioned earlier handles this by setting the partial-code of a rest parameter to return the appropriate constructs when the **car** or **cdr** of the replacement are needed.

Consider beta-expanding an invocation of a function with free variable references, e.g. **(let ((foo (make-counter))) (foo))** where **make-counter** is defined as:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (setq count (+ count 1)))))
```

If **(foo)** is beta-expanded, the reference to **count** within the body of **foo** is unresolvable since **foo**'s definition-time environment is inaccessible. The solution is to introduce a construct (the *closure* parameter in a variable reference) which looks up a name in the indicated closure. The expression **(foo)** would thus be expanded as **(setq (variable count foo 0) (+ (variable count foo 0) 1))**.

Notice that the above variable form violates lexical scoping by referring to a variable in an arbitrary closure's environment. The existence of such a form implies that variables can be mutated outside of their local scope and thus complicates certain types of program analysis.

3.5.2.3 Open Specialization

Open specialization is useful where beta-expansion is impossible or undesirable, for example where actual arguments have side-effects or are very large. Since open specialization preserves the function call mechanism, imperative and computation-intensive arguments present no problem. An invocation is open-specialized by partially evaluating the function definition in the after-env augmented

by binding the formal parameters to the partially evaluated actual arguments, and creating a lambda-form with the returned code. As is the case with beta-expansion, the body will be optimized to take advantage of any partial information. Open specialization is signalled by including the symbol **open-specialize** in a function's **assert** clause.

A possible optimization of open-specialized functions (and lambda forms in general) is the elimination of lambda variables that are not needed because the optimized body contains no references to these variables (dead-variable elimination). This simple optimization has not been implemented.

3.5.2.4 Closed Specialization

Closed specialization is useful where the same open specialization would be performed many times and one would prefer simply to define a new function. The partial evaluator does not presently implement closed specialization. For closed specialization to be useful, very fine control would have to be provided. For example, a separate specialization might be desired for each value of a certain argument or each data-type of a certain argument, or even the number of arguments.

3.5.3 Evaluated Results

A point that has been glossed over in the preceding discussions is the code that is reported for an evaluated invocation, in which the value is fully-specified. Ideally one would wish to replace the form by an appropriate literal. Unfortunately, several problems arise.

Some values have no literal forms, such as vectors and closures. Even if a literal form were to exist, we might not want to use it, for example in the case of a very large list structure. For example, in partially evaluating `(make-empty-list 10000)`, though the value would be an empty list of ten thousand elements, the code should be the generating expression.

Another difficulty arises when the value of the form is subject to mutation, e.g. the form returns a component of a shared data structure. In this case, using a literal would be incorrect, since it would create a new value where the original (shared) value is needed to preserve program semantics. Yet another issue is preserving pointer identity: if two variables (say `a` and `b`) contain the same value (say `'(1 2)`), the expression `(eq? a b)` should return true, but the expression `(eq? a '(1 2))` will return false, since the literal will create a different list, albeit with the same structure.

The present solution to the above problems is to only use literals for small, immutable values, such as fixnums and booleans. Otherwise the returned code is simply the original expression (though the returned value is the evaluated result).

Another solution is to introduce new variables to hold evaluated values and thus preserve sharing and pointer identity. Large numbers of new variables may need to be introduced, however.

Perhaps the best solution would be to define an external representation for values of each type. Herlihy [20] shows that it is possible to define such a representation that preserves sharing and identity. To preserve the readability of the output code, external representations of values could be "commented" with the code that would generate the value. This generating code can then be edited instead of the external representation of the value.

3.6 Partially Evaluating Variable References

The interpreter, when presented with a variable reference, simply looks up the name in the environment as described in the last chapter. The partial evaluator follows a more complicated strategy: it looks up the variable, and then decides what value and code to return.

In the simpler case, the name is to be looked up in the environment of a

particular closure. First the code denoting the closure (in the variable form) is partially evaluated. If the closure is known, the lookup is performed. The lookup returns a value, a frame-count, and the value's read-only flag. A case analysis is done in order to return the appropriate code and value:

- If the read-only flag is true, and the value is a simple, immutable value, then the variable can be replaced by the appropriate literal. The literal serves as both code and value.

- If the read-only flag is true, but the value is a compound value, then the value is returned, but the code returned is (**variable** *name closure frame-count*), where *closure* is the optimized closure-code, and *frame-count* is the count reported by the lookup.

- If the read-only flag is false, then the value returned is ***unknown***, and the code returned is as above. Returning an ***unknown*** value guarantees that the code will incorporate no assumptions that could be invalidated by non-local code.

In the other case, the closure-code is nil and the name is to be looked up in the "current" environment. The name must be looked up in the before-env, but the code must be calculated relative to the after-env, as mentioned previously. The name lookup returns not only a value, a frame-count, and a read-only flag as above, but also a *handle* (which is the code of the variable's enclosing closure, or nil), and an *argument* flag (which indicates that the value is a partial representing an actual argument to be substituted for a formal parameter reference). To calculate the code, the after-env is searched for a binding containing the value. As above, a case analysis is necessary:

- If the read-only flag is true, and the value is a simple, immutable value, then the variable can be replaced by a literal, as discussed above.

- If the argument flag is true, then the value is returned, after compensating for any change in lexical position, as discussed in the section on beta-expansion.

- If a binding was found for the value in the after-env, then the code returned is (**variable** *name* **nil** *frame-count*) where *frame-count* is

relative to the after-env. The value returned is the value found in the before-env if the read-only flag is true, otherwise ***unknown***.

- Otherwise, the code returned is **(variable** *name handle frame-count*), and the value returned is either the value found in the before-env if the read-only flag is true, otherwise ***unknown***.

Note that the after-env is only used to calculate the code for a variable reference: all values are looked up in the before-env.

As an example, consider the beta-expansion of **(bar b)** where **bar** is defined by:

```
(define (bar a)
        (assert beta-expand)
   (- a x))
```

A in **(- a x)** is replaced by **b**[8] since **b** is an actual argument to be substituted and so argument flag will be true. Assuming **x** is not read-only, it will be replaced by **(variable x bar 1)** (say).

Including a **(read-only** *vars*) form in the **assert** clause of a function definition introduces a list of read-only variables. The programmer (or the generator of the code) bears the responsibility for ensuring that values asserted to be read-only do not get mutated. For example, in

```
(let ((a '(1 2 3)))
    (assert (read-only a))
    (sequence (mung a)
              (* (first a) (second a))))
```

the programmer must ensure that **mung** does not perform a **set!-car** or other mutation on **a**.

---

[8] which is actually **(variable b nil 0)**

3.7 Partially Evaluating Other I-code Forms

3.7.1 Definitions

The interpreter, upon encountering a definition, evaluates the value and inserts the result in the current frame under the name. The partial evaluator analogously partially evaluates the value and inserts the returned partial into the environments (both the before- and after-env). The code returned is a definition with the same name, but the value component is the code component of the returned partial (i.e. the optimized code that will produced the desired result).

3.7.2 Lambda-Forms

When the interpreter encounters a lambda-form, it creates a lambda closure. The partial evaluator does the same, first partially evaluating the body of the form with the formal parameters bound to ***unknowns*** in order to (partially) optimize the body of the lambda.

3.7.3 Sequences

The sequence form is provided only for side-effects: all but the last expression in the form are executed only for the side-effects they produce. Thus only the side-effects of each expression (except the last) need be retained. A common use of the sequence form is to introduce local definitions. If, in the partial evaluation of the rest of the expressions in the form, the name of the local function is no longer used, then the definition could potentially be eliminated (though this is not currently implemented). Finally, if only one expression remains, the entire form can be replaced by the final expression.

3.7.4 Assignments

The value is partially evaluated, and the result returned. The remaining parameters (the name, closure, and frame-count) are partially evaluated in a fashion similar to variables.

3.8 List of Assertions

As described earlier, assertions are used to guide the partial evaluator. Here is the complete list of possible assertions and their meanings:

- **evaluate** - Invocations of this function should be evaluated if all the arguments are known.
- **applicative** - Invocations of this function are applicative and can be eliminated if only side-effects are needed.
- **macro-like** - Invocations of this function should be beta-expanded.
- **beta-expand** - Invocations of this function should be beta-expanded if all of the arguments are applicative.
- **(beta-expand** *expression*) - Invocations of this function should be beta-expanded if all of the arguments are applicative and the value of the expression is known at partial evaluation time.
- **open-specialize** - Invocations of this function should be open-specialized.
- **(read-only** *names*) - The variables with the given names are read-only, i.e. they contain immutable values and will not be re-bound.
- **(returns** *type-keyword*) - The function returns values of the indicated type.

3.9 Summary

A partial evaluator is defined as a program R that takes a binary function P and its first argument X such that R(P,X)(Y) = P_X(Y) = P(X,Y). P_X is an optimized version of P that has been specialized to the particular value of X. The partial evaluation of an expression returns both a partially-specified value and the optimized code that will produce the fully-specified value at run-time. Optimizations presented include constant folding and propagation, beta-expansion and open specialization. The partial evaluation of variable references is complicated by the possible change of an expression's enclosing environment before and after partial evaluation. User-supplied assertions are used to guide the partial evaluation process.

4. Examples of Partial Evaluation

4.1 Overview

We intend to use partial evaluation as a tool in implementing language extension mechanisms. The two examples of partial evaluation presented in this chapter correspond to two methods of implementing language extensions. First, a simple message-passing system is implemented by means of syntax macros. In the second example, a declarative programming construct is implemented by an interpreter written in IBL.

4.2 Implementing Extension Mechanisms

As mentioned previously, this thesis forms part of a larger project. A component of the proposed system is a "front-end" which is a collection of transformations from various surface language constructs to IBL constrcuts. Since this front-end does not yet exist, some sample transformations were implemented to test the capabilities of the partial evaluator. The extension mechanisms described here are unrealistically simple, but were helpful in understanding the functionality necessary of a useful partial evaluator.

The first example is an implementation of a message-passing construct using syntax macros. A module definition is transformed into the definition of an instance-creating function. An instance is a message-handling function whose code dispatches to the various operations that the module supports. The partial evaluator can optimize an instance invocation by repeated beta-expansion.

In the second example, a simple declarative programming construct is implemented by writing an interpreter in IBL. The rules-interpreter (as this example is known) takes a list of formulae (rules), lists of input and output variables and a list of input values, and displays the output values. The partial evaluator is used to specialize invocations of the rules-interpreter with some of the above parameters un-

specified. The experiment was designed to explore to what extent a partial evaluator can act as a compiler.

4.3 The Optimization of Message-Passing

As an example of how to implement type systems in IBL, the implementation and partial evaluation of a simple message-passing type mechanism will be presented in this section. First a simple no-inheritance version will be shown, and then a multiple-inheritance [6] version will be discussed.

4.3.1 Simple Types

Though many different semantic models of type definition are available, a simple static definition model is used below for ease of exposition. In this model, a type is declared with all its operations at once, e.g.

```
(defmodule ex1 (var1 var2)      ; module "ex1" has instance variables
                                ; "var1" and "var2" and operations
                                ; "init:", "var1:", and "var2:"

  (init: (a b)  ; initialize the instance variables
    (sequence (setq var1 a)
              (setq var2 b)))
  (var1: ()     ; return the value of "var1"
    var1)
  (var2: ()     ; return the value of "var2"
    var2))
```

It is not possible to change the operation set except by re-defining the type. Note that other semantics are quite feasible, e.g. a dynamic definition model in which the set of operations is mutable after definition-time.

The essential idea is to use a syntactic transformation (see appendix II) to convert the above module definition into:

```
(define (ex1 type-keyword . args)     ; module creator
        (assert evaluate (read-only type-keyword args))
  (cond ((eq? type-keyword create-instance:)
         (let ((var1 nil)
               (var2 nil))
           (sequence
             (define (self instance-keyword . args)   ; newly created module
                     (assert (beta-expand instance-keyword)
                             (read-only instance-keyword))
               (cond ((eq? type-keyword init:)
                      (apply (lambda (self a b)
                               (assert beta-expand (read-only self))
```

```
                    (sequence (setq var1 a)
                              (setq var2 b)))
                  (cons self args)))
            ((eq? type-keyword var1:)
             (apply (lambda (self)
                      (assert beta-expand (read-only self))
                      var1)
                    (cons self args)))
            ((eq? type-keyword var2:)
             (apply (lambda (self)
                      (assert beta-expand (read-only self))
                      var2)
                    (cons self args)))
            ((eq? type-keyword type:)
             'ex1)
            (true "unknown instance-keyword")))
      self)))
 ((eq? type-keyword type:)
  'module)
 (true "unknown module keyword")))
```

Self is passed as an argument to the methods so that the method bodies can send messages to their own instances. Note that each instance will share code but replicate the environment frame that holds the instance variables, so that each instance has separate variables.

To create a type instance, a form such as

(defconst x (ex1 create-instance:))

is evaluated. The partial evaluation of

(x init: 1 2)

proceeds as follows:

1. **x** is a read-only variable (since **(defconst a b)** is just syntax for **(def a b true)**), so **x** partially evaluates to a partial with value **(lambda (instance-keyword . args) ...)**. The arguments are all constant, so they partially evaluate to (essentially) themselves.[9]

2. Since **x** is a lambda, its **assert** clause is checked. The **instance-keyword** is indeed known, so the invocation is beta-expanded, and reduces to the **cond** form.

3. **Cond** is actually syntax for the equivalent nested **if** forms, so the expression to be partially evaluated is actually

```
(if (eq? instance-keyword init:) (apply ...)
    (if (eq? instance-keyword var1:) (apply ...)
```

---

[9]Actually **1** partially evaluates to a partial with value 1 and code **1**.

```
... ))
```

Partially evaluating an **if** form involves first partially evaluating the predicate, in this case **(eq? instance-keyword init:)**.

4. **Eq?** partially evaluates to a partial with the applicative built-in function **eq?** as value. Since all the arguments are known (**instance-keyword** is replaced by **init:** through beta-expansion, and **init:** is a constant), the invocation is evaluated and the result is **true**.

5. The **if** form thus reduces to its consequent: **(apply (lambda (self a b) ...) (cons self args))**. **Apply** partially evaluates to a partial with the built-in function **apply** as value. The first argument partially evaluates with a closure as value.

6. The second argument, **(cons self args)**, is an invocation of the built-in function **cons**. **Self** partially evaluates with the same value as **x** (the original instance), and code **(variable self x 0)**. **Args** is replaced through beta-expansion by a partial with value **(1 2)** and code **'(1 2)** (since **args** was a rest parameter).

7. Since all the arguments are known and applicative, the partial evaluation of the **apply** invocation reduces to the partial evaluation of (approximately)

```
((lambda (self a b)
   (beta-expand (read-only self))
   (sequence (setq var1 a)
             (setq var2 b)))
 (variable self x 0) 1 2)
```

The arguments are "unspread" by partially evaluating the **car**, **cadr**, etc. of the second argument to **apply**. The partial-code mechanism ensures that **(car '(1 2))** is replaced by **1**.

8. The above form is now partially evaluated, and beta-expansion is again performed, with **(sequence (setq var1 a) (setq var2 b))** being the residual expression. First the sub-forms of the sequence form are partially evaluated.

9. **(Setq var1 a)** is syntax for **(assign var1 nil 0 a)**. **A** partially evaluates with value 1, and the remaining parameters are partially evaluated as would be **(variable var1 nil 0)**. **Var1** is first looked up in the pe-env. The variable is not read-only, so the value reported is simply ***unknown***. The value does not exist in the run-env, so the code reported is **(variable var1 x 0)**. The **assign** form thus partially evaluates with value 1 and code **(assign var1 x 0 1)**.

10. The code of the partially evaluated sequence form is the sequence of the side-effects of its sub-forms, so the resulting code is `(sequence (assign var1 x 0 1) (assign var2 x 0 2))`. Since side effects are preserved at each step, this form is the final resulting code for `(x init: a b)`!

4.3.2 Multiple Inheritance

Multiple inheritance is defined here not as dynamic method lookup (which is possible, but messier), but rather as static inclusion. That is, when a type is defined, the super-types are queried for their instance variables and method definitions, which are then syntactically incorporated into the sub-type. An example is:

```
(defmodule super1 (var1) ()
           (init: (a)
             (setq var1 a))
           (var1: ()
             var1))

(defmodule super2 (var2) ()
           (init: (a)
             (setq var2 a))
           (var2: ()
             var2))

(defmodule ex1 () (super1 super2)
           (init: (a b)
             (sequence (setq var1 a)
                       (setq var2 b))))
```

The static inclusion implies that for any change in a type to be reflected in a sub-type, the sub-type must be redefined.

The last module definition is transformed into:

```
(define (ex1 type-keyword . args)      ; module creator
        (assert evaluate (read-only type-keyword args))
  (cond ((eq? type-keyword create-instance:)
         (let ((var1 nil)
               (var2 nil))
           (sequence
             (define (self instance-keyword . args)   ; newly created module
                     (assert (beta-expand instance-keyword)
                             (read-only instance-keyword))
               (cond ((eq? type-keyword init:)
                      (apply (lambda (self a b)
                               (assert beta-expand (read-only self))
                               (sequence (setq var1 a)
                                         (setq var2 b)))
                             (cons self args)))
                     ((eq? type-keyword var1:)
                      (apply (lambda (self)
                               (assert beta-expand (read-only self))
                               var1)
```

```
                    (cons self args)))
                ((eq? type-keyword var2:)
                 (apply (lambda (self)
                          (assert beta-expand (read-only self))
                          var2)
                        (cons self args)))
                ((eq? type-keyword type:)
                 'ex1)
                (true "unknown instance-keyword")))
        self)))
    ((eq? type-keyword type:)
     'module)
    ((eq? type-keyword ivars:)
     '(var1 var2))
    ((eq? type-keyword methods:)
     '((init: (a b)
         (sequence (setq var1 a)
                   (setq var2 b)))
       (var1: () var1)
       (var2: () var2)))
    (true "unknown module keyword")))
```

The definition, usage, and partial evaluation of type instances are the same as the no-inheritance case.

The instance variables and methods are collected using code such as the following:

```
(append-all (mapcar (lambda (super-name)
                      ((symeval super-name) ivars:))
                    (third form)))
```

Append-all concatenates the lists together, eliminating duplicates in some order. The algorithm used by **append-all** to eliminate duplicates determines the priority rules by which two methods with the same name from different classes are dis-ambiguated.

The above mechanism could quite easily be improved to include such feature as method extension, initialization options, etc. By adding dynamic method definition and lookup, one could simulate the ZetaLisp flavor system [26] quite closely.

4.4 Compiling by Partial Evaluation

Partial evaluation can in theory be applied to compilation. Say R is the partial evaluator, X is a program in language A, and Y is the input to X. Say P is an interpreter for language A and written in language B. $R(P,X) = P_X$ is then a program in language B, which when given Y, computes X(Y). P_X is then a compiled version of X, and the partial evaluator has acted as a compiler. One can thus think of a partial evaluator as a compiler-schema, which when given an interpreter (or equivalently, an operational semantics), becomes a compiler.

According to the above definition of partial evaluation, R(P,X) = R(R,P)(X). $R(R,P) = R_P$ is thus a compiler, translating programs in language A to programs in language B. One can go one step further, and write R(R,R)(P)(X)(Y) = P(X,Y) = X(Y). Thus $R(R,R) = R_R$ is a compiler generator, which when given an interpreter P produces a compiler.[10]

Partially evaluating an interpreter together with a partial specification of its input is thus equivalent to compilation. This is an important property of partial evaluation, since arbitrary extensions to a language can be implemented by writing interpreters for the extensions. This chapter illustrates the idea by implementing a simple extension to IBL as an interpreter: a declarative programming construct that accepts a list of input variables, a list of input values, a list of formulae, and a list of output variables.

---

[10]The above discussion is taken from [4]. While the authors do not mention the application to compiler generation, they have written a version of R(R,R) to act as a "partial evaluation compiler", i.e. to speed up the partial evaluation of the same program with different inputs. Ershov [13, 14, 15] does mention the possibility of generating compilers from operational semantics by partial evaluation.

4.4.1 The Rules Interpreter

The rules-interpreter (called ri) creates a dependency graph on the formulae, sets the input variables to the input values, evaluates the formulae in the appropriate order, and finally prints out the values of the output variables. An example of its use is:

```
(define (test-ivals)
        (let ((ivals (read)))
          (ri '(a b c) ivals '((d . (* a b))
                               (e . (+ b d))
                               (f . (- e c))
                               (g . (* h e))
                               (h . (- d b)))
              '(f g h))))
```

The partial evaluator converts the above function definition to:

```
(define (test-ivals)
        (let ((ivals (read)))
          (sequence (print (iminus (iplus (cadr ivals)
                                          (itimes (car ivals)
                                                  (cadr ivals)))
                                   (caddr ivals)))
                    (print (itimes (iminus (itimes (car ivals)
                                                   (cadr ivals))
                                           (cadr ivals))
                                   (iplus (cadr ivals)
                                          (itimes (car ivals)
                                                  (cadr ivals)))))
                    (print (iminus (itimes (car ivals)
                                           (cadr ivals))
                                   (cadr ivals))))))
```

Note that all run-time interpretation and type-checking has been eliminated. This example illustrates both the strengths of partial evaluation (the optimizing power of multiple beta-expansion and partial knowledge propagation) and a major weakness (common subexpression introduction). See Appendix III for a complete listing of the code.

4.4.2 Partially Evaluating the Rules Interpreter

The partial evaluation of the above example proceeds as follows:

1. Ri's definition is:

```
(define (ri invars invals rules outvars)
        (assert beta-expand (read-only invars invals outvars rules))
  (display outvars (calculate invars invals rules (tsort rules nil))))
```

   so the call to ri is beta-expanded.

2. The call to **tsort** is then partially evaluated. The definition is

```
(define (tsort rules clist)
        (assert evaluate (beta-expand rules) (read-only rules clist))
        (if (nil? rules) clist
            (tsort (cdr rules) (insert (car rules) clist))))
```

Since all the arguments are known and applicative, the invocation is evaluated. The partial evaluation results in a partial whose value is the dependency graph and whose code is **(tsort rules nil)**, since the value may be mutable.

3. Next to be partially evaluated is the call to **calculate**, whose definition is:

```
(define (calculate invars invals rules clist)
        (assert beta-expand (read-only invars invals rules clist) applicative)
  (let ((env (init-env invars invals)))
    (assert beta-expand (read-only env))
    (calculate1 env rules clist)))
```

The invocation is beta-expanded, as is the let-form, so after the call to **init-env** has been handled, the partial evaluation reduces to that of the call to **calculate1**.

4. The definition of **init-env** is:

```
(define (init-env vars vals)
        (assert (beta-expand vars) (read-only vars vals) applicative)
  (if (nil? vars) nil
      (cons (cons (car vars) (car vals))
            (init-env (cdr vars) (cdr vals)))))
```

Since the **vars** are known, the invocation is beta-expanded. The value returned is:

```
(cons (cons 'a (car ivals))
      (cons (cons 'b (cadr ivals))
            (cons (cons 'c (caddr ivals))
                  nil)))
```

5. **Calculate1** is defined by:

```
(define (calculate1 env rules clist)
        (assert (beta-expand clist) (read-only env rules clist applicative)
                applicative)
  (if (nil? clist) env
      (let ((name (car clist))
            (formula (lookup (car clist) rules)))
        (assert beta-expand (read-only name formula))
        (calculate1 (cons (cons name (evaluate formula env))
                          env)
                    rules (cdr clist)))))
```

Clist (the dependency graph) is known, so the invocation is beta-expanded. Since the function is recursive, the beta-expansion will result in code replication. The let-form is also beta-expanded, so the partial evaluation reduces to the partial evaluation of the recursive call to **calculate1**.

6. First the arguments must be partially evaluated, most crucially **(evaluate formula env)**. **evaluate** is defined by:

```
(define (evaluate exp env)
        (assert (beta-expand exp) applicative (read-only exp env))
  (if (atom? exp) (if (symbol? exp) (lookup-fixnum exp env)
                      exp)
      (let ((func (car exp))
            (args (eval-list (cdr exp) env)))
        (assert beta-expand open-specialize (read-only func args))
        (cond ((eq? func '+) (+ (first args) (second args)))
              ((eq? func '-) (- (first args) (second args)))
              ((eq? func '*) (* (first args) (second args)))
              (true "Unknown function."))))))
```

Since **exp** (the formula) is known, the invocation is beta-expanded. Again the definition is recursive (through **eval-list**), so there will be code replication. The recursion bottoms out where **exp** is a symbol, in which case the invocation of **evaluate** reduces to an invocation of **lookup-fixnum.**

7. The definition of **lookup-fixnum** is:

```
(define (lookup-fixnum name env)
        (assert (beta-expand name (caar env))) (read-only name env)
                (returns fixnum:) evaluate applicative)
  (cond ((nil? env) unknown:)
        ((eq? (caar env) name) (cdar env))
        (true (lookup-fixnum name (cdr env)))))
```

The invocation is beta-expanded, and **env** is (eventually, using the partial-code mechanism to take apart **init-env**'s value) reduced to a form involving **ivals**, so a call to **lookup-fixnum** reduces to a form such as **(cadr ivals)**. **Lookup-fixnum** has a **(returns fixnum:)** clause, so the type checks in **+**, etc. can be eliminated. The final result of partially evaluating the original invocation of **evaluate** is a partial whose value is unknown, but whose code is the compiled version as above.

8. Finally, the invocation of **display** is partially evaluated. **Display** is defined by:

```
(define (display outvars env)
        (assert beta-expand (read-only outvars env))
  (sequence
    (mapcar (lambda (outvar)
              (assert (beta-expand outvar) (read-only outvar))
              (print (lookup-fixnum outvar env)))
            outvars)
    true))
```

The call is beta-expanded, so the side-effects of the invocation of **mapcar** are incorporated into a new sequence form.

9. **Mapcar** is defined by:

```
(define (mapcar func lst)
        (assert (beta-expand lst) (read-only func lst))
  (if (nil? lst) nil
      (cons (func (car lst))
            (mapcar func (cdr lst)))))
```

Using the partial-code mechanism, the result of partially evaluating **calculate1 (lst)** is broken up through the calls to **lookup-fixnum (func)**. Since only the side-effects of this invocation are needed, only the final **print** statements remain.

Note the power of beta-expanding recursive functions: not only is automatic loop-unrolling achieved, but even more general recursion-elimination is possible. Conditional beta-expansion is used to guard against the possibility of infinite recursion while beta-expanding recursive function calls.

4.4.3 Other Examples

The rules interpreter can be partially evaluated with different arguments left unknown. For example, consider leaving the output variables unknown while completely specifying the rules and the input variables and values. The output values can be calculated by the partial evaluator; all that is necessary is to display the appropriate results.

```
(define (test-ovars)
        (let ((ovars (read)))
          (sequence
            (mapcar
              (lambda (outvar)
                (assert (beta-expand outvar))
                (print (lookup-fixnum
                         outvar
                         (cons
                           (cons 'f 1)
                           (cons
                             (cons 'g 0)
                             (cons (cons 'e 4)
                                   (cons (cons 'h 0)
                                         (cons (cons 'd 2)
                                               (cons (cons 'a 1)
                                                     (cons (cons 'b 2)
                                                           (cons (cons 'c 3)
                                                                 nil)))))))))))
              ovars)
            true)))
```

A great deal of time and space is wasted by re-creating the association list of variables and values for each output variable, though the optimized program is still quite a bit faster than the original version.

Leaving the input variables unknown results in:

```
(define (test-ivars)
  (let ((ivars (read)))
    (sequence
      (print
        (iminus (iplus (lookup-fixnum 'b (init-env ivars '(1 2 3)))
                       (itimes (lookup-fixnum 'b (init-env ivars '(1 2 3)))
                               (lookup-fixnum 'a (init-env ivars '(1 2 3)))))
                (lookup-fixnum 'c (init-env ivars '(1 2 3)))))
      (print
        (itimes (iminus (itimes (lookup-fixnum 'a (init-env ivars '(1 2 3)))
                                (lookup-fixnum 'b (init-env ivars '(1 2 3))))
                        (lookup-fixnum 'b (init-env ivars '(1 2 3))))
                (iplus (lookup-fixnum 'b (init-env ivars '(1 2 3)))
                       (itimes (lookup-fixnum 'a (init-env ivars '(1 2 3)))
                               (lookup-fixnum 'b (init-env ivars '(1 2
                                                                   3)))))))
      (print
        (iminus (itimes (lookup-fixnum 'a (init-env ivars '(1 2 3)))
                        (lookup-fixnum 'b (init-env ivars '(1 2 3))))
                (lookup-fixnum 'b (init-env ivars '(1 2 3)))))))
```

The list relating the input variables and input values (`(init-env ivars '(1 2 3))`) is re-created for each reference to an input variable, which is highly inefficient, though this optimized function still runs faster than the original version. While the rules interpreter could probably be redefined somewhat to ameliorate the problem, the large-scale introduction of common sub-expressions is a frequent problem in partial evaluation.

4.5 Summary

In the extensible language project of which this thesis is a part, it is planned that partial evaluation will be used to implement language extensions in two ways:

1. to optimize (in the classical sense) the results of simple syntactic transformations, and

2. to "compile" by partially evaluating an interpreter.

Examples of the above techniques have been implemented, and the resulting code partially evaluated. Though the examples are unrealistically simple, the results are encouraging. Some obvious problems of the present partial evaluator are excess code replication and structure creation.

5. Summary and Conclusions

5.1 Recapitulation

The following list briefly summarizes the contents of this thesis.

- A partial evaluator has been designed and implemented which optimizes applicative code while retaining all side-effects. Unchecked assertions are used to guide the partial evaluator. The partial evaluator operates on a lexically-scoped dialect of Lisp.

- This thesis is part of a larger project devoted to language extensibility. We propose implementing language extension mechanisms on top of the language. Surface language constructs will be transformed into kernel language constructs which will then be optimized by the partial evaluator. The transformations will preserve user-specified side-effects and will add applicative code for type-checking, etc.

- In order to test the proposed language extension techniques and the partial evaluator, two simple examples of extension mechanisms were implemented. Though these examples were extremely simple, the preliminary results were quite encouraging.

- To be effective, the partial evaluator must be able to represent and carefully propagate partially-specified program values. Special attention must be devoted to component access from partially-specified structures.

- Another problem encountered in the design of the partial evaluator was the difficulty of efficiently representing evaluated results. The existence of mutation and the possibility of structure sharing often preclude the simple use of literals and force the retention of the generating code.

5.2 Future Work

The present partial evaluator suffers from a number of limitations. Some major missing optimizations are closed specialization and dead variable elimination. Introducing these optimizations would eliminate much of the current common-subexpression introduction problem. Closed specialization would enable function calls to replace code repetition, and dead variable elimination would allow the

introduction of new local variables to hold temporary results without fear of variable build-up.

The partial evaluator presently takes a very conservative approach to side-effects: *all* side-effects are preserved, even though some may conceivably be eliminated without changing program semantics. Another problem is that the partial evaluator does not keep track of the values of assignable variables. These two limitations imply that code must be largely applicative for effective optimization to take place. The value-numbering scheme mentioned in [8] may be useful in this area.

The user-supplied assertions are quite critical to the operation of the partial evaluator. At the moment, these assertions must be carefully hand-tuned to achieve good optimization. The reliance of the partial evaluator on some of these assertions could be eliminated by giving it more powerful inference capabilities. Another improvement would be the incorporation of facilities for automatically deciding between various optimizations in order to replace assertions such as **beta-expand** and **evaluate** (see Scheifler's paper on inline expansion [28] for an example). A mitigating factor is that only the implementor of extension mechanisms need deal with assertions, as they are invisible to the surface language.

Partially-specified values are fairly limited in the present partial evaluator. Some of the extensions mentioned by Haraldsson [17] might be worth incorporating. His partially-specified values include value ranges, which have also been found useful in [2]. Other possibilities include impossible values and the implications of the value evaluating to certain values (e.g. if `(eq? a 1)` is true, then `a` is 1). A recent paper by Babad and Hoffer [3] explores the use of partially specified values in other contexts.

At the moment, the handling of evaluated results is inadequate. Code must frequently be emitted to generate structures that are fully-known at partial-

evaluation-time. Herlihy's scheme [20] for an external representation of values that preserves sharing should be incorporated to alleviate this problem.

The partial evaluator is currently quite inefficient. A major performance problem is the frequent construction of structures that are never used. Another source of inefficiency is the frequent re-optimization of code that has already been processed. The use of lazy evaluation might improve performance by avoiding partial evaluation until the result is actually needed.

5.3 Conclusion

The partial evaluator provides a simple, coherent framework for many traditional optimization techniques such as constant folding and propagation, in-line expansion, dead code elimination and function specialization. The power of this approach is indicated by the excellent optimization achieved by our very small partial evaluator (which is implemented in just a few hundred lines of code).

I. The IBL Utilities

```
;;; -*- Mode: LISP -*-

;;; A collection of utility syntax rules and functions to make life easier.

(def add-syntax (user-syntax (form)
                             (cons 'def
                                   (cons (car form)
                                         (cons (cons 'user-syntax
                                                     (cdr form))
                                               (cons 'true nil))))))
  true)

(add-syntax define (form)
            (cons 'def
                  (cons (car (car form))
                        (cons (cons 'lambda
                                    (cons (cdr (car form))
                                          (cdr form)))
                              (cons 'true nil)))))

;; List-processing functions:

(define (caar x)
        (assert evaluate applicative macro-like)
  (car (car x)))

(define (cadr x)
        (assert evaluate applicative macro-like)
  (car (cdr x)))

(define (cdar x)
        (assert evaluate applicative macro-like)
  (cdr (car x)))

(define (cddr x)
        (assert evaluate applicative macro-like)
  (cdr (cdr x)))

(define (list . x)
        x)

(define (first x)
        (assert evaluate applicative macro-like)
  (car x))

(define (second x)
        (assert evaluate applicative macro-like)
  (cadr x))

(define (third x)
        (assert evaluate applicative macro-like)
  (car (cddr x)))

(define (fourth x)
        (assert evaluate applicative macro-like)
  (cadr (cddr x)))

(define (length lst)
        (assert evaluate applicative (returns fixnum:))
  (if (nil? lst) 0
      (+ 1 (length (cdr lst)))))
```

```
(define (mapcar func lst)
        (assert (beta-expand lst) (read-only func lst))
  (if (nil? lst) nil
      (cons (func (car lst))
            (mapcar func (cdr lst)))))

;; Type-checking functions:

(define (fixnum? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) fixnum:))

(define (pair? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) pair:))

(define (atom? x)
        (assert evaluate applicative macro-like)
  (not (eq? (type x) pair:)))

(define (nil? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) nil:))

(define (keyword? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) keyword:))

(define (symbol? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) symbol:))

(define (vector? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) vector:))

(define (string? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) string:))

(define (built-in-func? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) built-in-func:))

(define (lambda? x)
        (assert evaluate applicative macro-like)
  (eq? (type x) lambda:))

(add-syntax cond (form)              ; multi-way conditional
            (list 'if
                  (first (first form))
                  (second (first form))
                  (if (eq? (cdr form) nil) nil
                      (cons 'cond (cdr form)))))

(add-syntax let (form)               ; local variable binding
            (cons (cons 'lambda
                        (cons (names (first form))
                              (cdr form)))
                  (values (first form))))

(define (names let-list)
        (mapcar first let-list))

(define (values let-list)
        (mapcar second let-list))
```

```
;; "Careful" arithmetic functions.

(define (+ a b)
        (assert evaluate applicative beta-expand)
  (if (and (fixnum? a) (fixnum? b))
      (iplus a b)
      unknown:))

(define (- a b)
        (assert evaluate applicative beta-expand)
  (if (and (fixnum? a) (fixnum? b))
      (iminus a b)
      unknown:))

(define (* a b)
        (assert evaluate applicative beta-expand)
  (if (and (fixnum? a) (fixnum? b))
      (itimes a b)
      unknown:))

(define (< a b)
        (assert evaluate applicative beta-expand)
  (if (and (fixnum? a) (fixnum? b))
      (iless? a b)
      unknown:))

(define (> a b)
             (assert evaluate applicative beta-expand)
  (if (and (fixnum? a) (fixnum? b))
      (igreater? a b)
      unknown:))

(add-syntax back-quote (form)         ; syntax template
            (sequence
              (define (transform exp)
                      (cond ((atom? exp) (list 'quote exp))
                            ((eq? (car exp) 'comma) (second exp))
                            ((tree-memq? 'comma exp) (list 'cons
                                                           (transform (car exp))
                                                           (transform (cdr exp))))
                            (true (list 'quote exp))))
              (transform (first form))))

(define (memq? elt lst)
        (cond ((nil? lst) false)
              ((eq? (car lst) elt) true)
              (true (memq? elt (cdr lst)))))

(define (tree-memq? elt tree)
        (cond ((atom? tree) (eq? elt tree))
              ((tree-memq? elt (car tree)) true)
              (true (tree-memq? elt (cdr tree)))))

(add-syntax setq (form)               ; assignment
            (let ((var (first form))
                  (value (second form)))
              (if (atom? var) '(assign ,var nil 0 ,value)
                  '(assign ,(second var) ,(third var) ,(fourth var) ,value))))

(define (append a b)
        (if (nil? a) b
            (cons (car a) (append (cdr a) b))))

(add-syntax defvar (form)             ; global variable definition
            '(def ,(first form) ,(second form) false))
```

```
(add-syntax defconst (form)              ; global constant definition
            '(def ,(first form) ,(second form) true))
```

II. A User-Defined Syntax Macro

This is the definition of the syntax macro that implements the no-inheritance version of the **defmodule** form mentioned in section 4.3.1:

```
;; Lexically scoped module definition syntax.

(add-syntax defmodule (form)
            (let ((name (first form))
                  (ivars (second form))
                  (method-list (cddr form)))
              (sequence
                (define (make-let-list vars)
                        (mapcar (lambda (var)
                                  '(,var nil))
                                vars))
                (define (make-cond-list method-list)
                        (append
                          (mapcar
                            (lambda (method)
                              '((eq? instance-keyword ,(first method))
                                (apply (lambda (self . ,(second method))
                                         (assert beta-expand
                                                 (read-only self))
                                         ,(third method))
                                       (cons self args))))
                            method-list)
                          '(((eq? instance-keyword type:)
                             ,(list 'quote name))
                            (true "unknown instance keyword"))))
                '(define (,name type-keyword . args)
                         (assert evaluate (read-only type-keyword args))
                   (cond ((eq? type-keyword create-instance:)
                          (let ,(make-let-list ivars)
                            (sequence
                              (define (self instance-keyword . args)
                                      (assert (beta-expand instance-keyword)
                                              (read-only instance-keyword))
                                (cond . ,(make-cond-list method-list)))
                              self)))
                         ((eq? type-keyword type:)
                          'module)
                         (true "unknown module keyword"))))))
```

III. The Rules Interpreter

```
;; Rule-based interpreter.

;; RI - Rules-interpreter
;;
;;    INVARS - List of input variables.
;;
;;    INVALS - List of input values.
;;
;;    RULES - List of formulae.
;;
;;    OUTVARS - List of output variables.

(define (ri invars invals rules outvars)
        (assert beta-expand (read-only invars invals outvars rules))
  (display outvars (calculate invars invals rules (tsort rules nil))))

;; TSORT - Topologically sort the constraint set.

(define (tsort rules clist)
        (assert evaluate (beta-expand rules) (read-only rules clist))
        (if (nil? rules) clist
            (tsort (cdr rules) (insert (car rules) clist))))

;; INSERT - Insert a value in a list maintaining dependencies.

(define (insert head clist)
        (sequence
          (define (dependencies exp)
                  (if (atom? exp) (cond ((nil? exp) nil)
                                        ((fixnum? exp) nil)
                                        ((eq? exp '+) nil)
                                        ((eq? exp '-) nil)
                                        ((eq? exp '*) nil)
                                        (true (list exp)))
                      (append (dependencies (car exp))
                              (dependencies (cdr exp)))))
          (if (overlap? (dependencies (cdr head)) clist)
              (cons (car clist) (insert head (cdr clist)))
              (cons (car head) clist))))

;; OVERLAP? - Does a formula depend on later formulae?

(define (overlap? deps clist)
        (cond ((nil? clist) false)
              ((nil? deps) false)
              ((memq? (car deps) clist) true)
              (true (overlap? (cdr deps) clist))))

;; CALCULATE - Calculates the results.

(define (calculate invars invals rules clist)
        (assert beta-expand (read-only invars invals rules clist) applicative)
  (let ((env (init-env invars invals)))
    (assert beta-expand (read-only env))
    (calculate1 env rules clist)))

;; INIT-ENV - Initialize the environment with the input variables and values.

(define (init-env vars vals)
        (assert (beta-expand vars) (read-only vars vals) applicative)
  (if (nil? vars) nil
      (cons (cons (car vars) (car vals))
```

```
          (init-env (cdr vars) (cdr vals)))))

(define (calculate1 env rules clist)
        (assert (beta-expand clist) (read-only env rules clist) applicative)
  (if (nil? clist) env
      (let ((name (car clist))
            (formula (lookup (car clist) rules)))
        (assert beta-expand (read-only name formula))
        (calculate1 (cons (cons name (evaluate formula env))
                          env)
                    rules (cdr clist)))))

;; LOOKUP - Lookup a formula in a rules-list.

(define (lookup name rules)
        (assert (beta-expand (cons name (caar rules)))
                (read-only name rules) evaluate applicative)
  (cond ((nil? rules) unknown:)
        ((eq? (caar rules) name) (cdar rules))
        (true (lookup name (cdr rules)))))

;; LOOKUP-FIXNUM - Lookup a value in an a-list.

(define (lookup-fixnum name env)
        (assert (beta-expand (cons name (caar env)))
                (read-only name env) (returns fixnum:)
                evaluate applicative)
  (cond ((nil? env) unknown:)
        ((eq? (caar env) name) (cdar env))
        (true (lookup-fixnum name (cdr env)))))

;; EVALUATE - Evaluate a formula.

(define (evaluate exp env)
        (assert (beta-expand exp) applicative (read-only exp env))
  (if (atom? exp) (if (symbol? exp) (lookup-fixnum exp env)
                      exp)
      (let ((func (car exp))
            (args (eval-list (cdr exp) env)))
        (assert beta-expand open-specialize (read-only func args))
        (cond ((eq? func '+) (+ (first args) (second args)))
              ((eq? func '-) (- (first args) (second args)))
              ((eq? func '*) (* (first args) (second args)))
              (true (list "Unknown function." func args))))))

(define (eval-list exps env)
        (assert (beta-expand exps) applicative (read-only exps env))
  (mapcar (lambda (exp)
            (assert (read-only exp) beta-expand)
            (evaluate exp env))
          exps))

;; DISPLAY - Display the results.

(define (display outvars env)
        (assert beta-expand (read-only outvars env))
  (sequence
    (mapcar (lambda (outvar)
              (assert (beta-expand outvar) (read-only outvar))
              (print (lookup-fixnum outvar env)))
            outvars)
    true))

(define (test-ri)
        (let ((ivals (read)))
          (ri '(a b c) ivals '((d . (* a b))
```

```
                                (e . (+ b d))
                                (f . (- e c))
                                (g . (* h e))
                                (h . (- d b)))
              '(f g h))))

(define (test-ovars)
        (let ((ovars (read)))
          (ri '(a b c) '(1 2 3) '((d . (* a b))
                                  (e . (+ b d))
                                  (f . (- e c))
                                  (g . (* h e))
                                  (h . (- d b)))
              ovars)))

(define (test-ivars)
        (let ((ivars (read)))
          (ri ivars '(1 2 3) '((d . (* a b))
                               (e . (+ b d))
                               (f . (- e c))
                               (g . (* h e))
                               (h . (- d b)))
              '(f g h))))
```

IV. The IBL Implementation

```
;;; -*- Mode: LISP; Base: 10.; Package: (iblc) -*-

;;;
;;; PARSER ************************************************
;;;

;; PARSE - Parses an expression. (S-expression -> I-code)

(defun parse (exp)
  (if (atom exp) (cond ((eq exp 'true) (make-boolean t))
                       ((eq exp 'false) (make-boolean nil))
                       ((symbol? exp) (make-variable exp nil 0))
                       (t exp))
      (let ((head (first exp))
            (tail (rest1 exp)))
        (let ((parse-func (send global-environment :get-value head)))
          (selectq (typep parse-func)
            (built-in-syntax (funcall (send parse-func :code) tail))
            (user-syntax (parse (apply-lambda (send parse-func :code) (list tail))))
            (otherwise (make-combination (parse head) (parse-list tail))))))))

;; PARSE-LIST - Parses a list pf expressions.

(defun parse-list (exps)
  (mapcar #'parse exps))

;; UNPARSE - Unparses an expression. (I-code -> S-expression)

(defun unparse (exp)
  (selectq (typep exp)
    (def-form '(def ,(send exp :name) ,(unparse (send exp :value))
                    ,(unparse (send exp :read-only?))))
    (lambda-form '(lambda (,@(send exp :required) . ,(send exp :rest))
                    (assert ,@(send exp :asserts))
                    ,(unparse (send exp :body))))
    (sequence-form '(sequence ,@(unparse-list (send exp :forms))))
    (if-form '(if ,(unparse (send exp :predicate))
                  ,(unparse (send exp :consequent))
                  ,(unparse (send exp :alternative))))
    (combination '(,(unparse (send exp :function))
                   ,@(unparse-list (send exp :args))))
    (variable (let ((name (send exp :name))
                    (closure (unparse (send exp :closure)))
                    (contours (send exp :contours)))
                (if (and (null closure) (zerop contours)) name
                    '(variable ,name ,closure ,contours))))
    (assignment '(assign ,(send exp :name) ,(unparse (send exp :closure))
                         ,(unparse (send exp :contours))
                         ,(unparse (send exp :value))))
    (lambda '(lambda (,@(send exp :required) . ,(send exp :rest))
               (assert ,@(send exp :asserts)) ,(unparse (send exp :body))))
    (built-in-func '(bif ,(send exp :name)))
    (user-syntax (let ((code (send exp :code)))
                   '(user-syntax ,(send code :required)
                                 ,(unparse (send code :body)))))
    (built-in-syntax 'built-in-syntax)
    (boolean (if (send exp :true?) 'true 'false))
    (partial '(value ,(unparse (send exp :value))
               code ,(unparse (send exp :code))
               side-effects ,(unparse-list (send exp :side-effects))
               known? ,(unparse (make-boolean (send exp :known?)))
               applicative? ,(unparse (make-boolean
```

```
                                        (send exp :applicative?)))))
    (frame '(handle ,(unparse (send exp :handle))
                    args? ,(unparse (send exp :arguments?))
                    alist ,(send exp :alist)))
    (unknown '(unknown ,(send exp :type)))
    (:list '(quote ,exp))
    (:symbol (if (symbol? exp) '(quote ,exp) exp))
    (:array 'vector)
    (otherwise exp)))

;; UNPARSE-LIST - Unparses a list of expressions.

(defun unparse-list (exps)
  (mapcar #'unparse exps))

;; INIT-SYNTAX - Loads the built-in parsing functions into the global environment.

(defun init-syntax ()
  (mapcar #'(lambda (clause)
              (rec-augment global-environment (first clause)
                           (make-built-in-syntax (second clause))
                           t))
          '((def ,#'(lambda (form)
                      (make-def-form (first form)
                                     (parse (second form))
                                     (parse (third form)))))
            (lambda ,#'(lambda (form)
                         (make-lambda-form (get-required (first form))
                                           (get-rest (first form))
                                           (get-asserts (rest1 form))
                                           (parse (get-body (rest1 form))))))
            (sequence ,#'(lambda (form)
                           (make-sequence-form (parse-list form))))
            (if ,#'(lambda (form)
                     (make-if-form (parse (first form))
                                   (parse (second form))
                                   (parse (third form)))))
            (quote ,#'(lambda (form)
                        (first form)))
            (bif ,#'(lambda (form)
                              (send global-environment :get-value
                                    (first form))))
            (user-syntax ,#'(lambda (form)
                              (make-user-syntax
                                (make-lambda (first form) nil nil
                                             nil (parse (second form))
                                             global-environment))))
            (variable ,#'(lambda (form)
                           (make-variable (first form)
                                          (parse (second form))
                                          (third form))))
            (assign ,#'(lambda (form)
                         (make-assignment (first form)
                                          (parse (second form))
                                          (third form)
                                          (parse (fourth form)))))))

;; GET-REQUIRED - Get the list of required variables from a lambda-list.

(defun get-required (lambda-list)
  (if (atom lambda-list) nil
      (cons (car lambda-list)
            (get-required (cdr lambda-list)))))

;; GET-REST - Get the rest-variable (if any) from a lambda-list.
```

```
(defun get-rest (lambda-list)
  (if (atom lambda-list) lambda-list
      (cdr (last lambda-list))))

;; GET-ASSERTS - Get the assertion-list from a function definition.

(defun get-asserts (def)
  (if (null (cdr def)) nil
      (cdar def)))

;; GET-BODY - Get the body from a function definition.

(defun get-body (def)
  (if (null (cdr def)) (first def)
      (second def)))
```

```
;;;
;;; TYPES ************************************************
;;;

(defun nil? (exp)
  (null exp))

(defun fixnum? (exp)
  (numberp exp))

(defun pair? (exp)
  (listp exp))

(defun string? (exp)
  (stringp exp))

(defun vector? (exp)
  (and (arrayp exp)
       (not (stringp exp))))

;; SYMBOL? - Is exp a non-keyword symbol?

(defun symbol? (exp)
  (and exp                                    ; not NIL
       (symbolp exp)
       (not (keyword? exp))))

;; KEYWORD? - Is exp a self-evaluating keyword?

(defun keyword? (exp)
  (and (symbolp exp)
       (member (aref (string exp)
                     (sub1 (string-length exp)))
               '(#/. #/:))))

;; BOOLEAN - A true-or-false value.

(defflavor boolean (true?) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-boolean (true?)
  (make-instance 'boolean :true? true?))

(defun boolean? (exp)
  (typep exp 'boolean))

;; VARIABLE - A program variable.
;;
;;   NAME - The print-name of the variable.
;;
;;   CLOSURE - The code of the closure whose environment contains the variable, or
;;       NIL if the containing environment is the current one (the usual case).
;;
;;   CONTOURS - The number of environment frames to skip before searching for the
;;       name-value binding.

(defflavor variable (name closure contours) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-variable (name closure contours)
  (make-instance 'variable :name name :closure closure :contours contours))

(defun variable? (exp)
  (typep exp 'variable))
```

```
;; ASSIGNMENT - A code-form denoteing assignment to a program variable.
;;
;;   NAME - The print-name of the variable.
;;
;;   CLOSURE - The code of the closure whose environment contains the variable, or
;;       NIL if the containing environment is the current one (the usual case).
;;
;;   CONTOURS - The number of environment frames to skip before searching for the
;;              name-value binding.
;;
;;   VALUE - The value to which the variable is to be assigned.

(defflavor assignment (name closure contours value) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-assignment (name closure contours value)
  (make-instance 'assignment :name name :closure closure
                 :contours contours :value value))

;; BUILT-IN-FUNC - A built-in function.

(defflavor built-in-func (name eval-func pe-func arg-types return-type) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-built-in-func (name eval-func pe-func arg-types return-type)
  (make-instance 'built-in-func :name name :eval-func eval-func :pe-func pe-func
                 :arg-types arg-types :return-type return-type))

(defun built-in-func? (exp)
  (typep exp 'built-in-func))

;; LAMBDA - A user-defined function.

(defflavor lambda (required rest asserts read-onlys body env) ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(defun make-lambda (required rest asserts read-onlys body env)
  (make-instance 'lambda :required required :rest rest
                 :asserts asserts :read-onlys read-onlys :body body :env env))

(defun lambda? (exp)
  (typep exp 'lambda))

;; DEF-FORM - An internal definition form.

(defflavor def-form (name value read-only?) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-def-form (name value read-only?)
  (make-instance 'def-form :name name :value value :read-only? read-only?))

;; LAMBDA-FORM - An internal function-value form.

(defflavor lambda-form (body asserts required rest) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-lambda-form (required rest asserts body)
  (make-instance 'lambda-form :required required :rest rest
                 :asserts asserts :body body))
```

```
;; COMBINATION - An internal function-invocation form.

(defflavor combination (function args) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-combination (function args)
  (make-instance 'combination :function function :args args))

;; SEQUENCE-FORM - An internal sequence form.

(defflavor sequence-form (forms) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-sequence-form (forms)
  (make-instance 'sequence-form :forms forms))

;; IF-FORM - An internal conditional form.

(defflavor if-form (predicate consequent alternative) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-if-form (predicate consequent alternative)
  (make-instance 'if-form :predicate predicate
                 :consequent consequent :alternative alternative))

;; FRAME - An environment frame.

(defflavor frame (next alist handle arguments?) ()
  :gettable-instance-variables
  :initable-instance-variables)

;; INSERT - Inserts a (name value read-only?) triple into the frame.

(defmethod (frame :insert) (name value read-only?)
  (setq alist (cons (cons name (cons value read-only?))
                    alist)))

;; GET-VALUE - Given a name, returns (VALUE FOUND? READ-ONLY?).

(defmethod (frame :get-value) (name)
  (let ((slot (assq name alist)))
    (if (null slot) (values nil nil nil)
        (values (cadr slot) t (cddr slot)))))

;; SET-VALUE - Given a name and a value, overwrites name's old value.

(defmethod (frame :set-value) (name value)
  (let ((slot (assq name alist)))
    (rplacd slot (cons value nil))))

;; GET-NAME - Given a value, returns (NAME FOUND?).

(defmethod (frame :get-name) (value)
  (let ((slot (get-triple value alist)))
    (if (null slot) (values nil nil)
        (values (car slot) t))))

(defun get-triple (value alist)
  (cond ((null alist) nil)
        ((eq value (cadar alist)) (car alist))
        (t (get-triple value (cdr alist)))))

(defun make-frame (next handle args?)
```

```
  (make-instance 'frame :next next :alist nil :handle handle :arguments? args?))

;; GLOBAL-FRAME - The top-level environment frame , implemented as a hash table.

(defflavor global-frame (name->value value->name) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defmethod (global-frame :insert) (name value read-only?)
  (send name->value :put-hash name (cons value read-only?))
  (send value->name :put-hash value name))

(defmethod (global-frame :get-value) (name)
  (multiple-value-bind (pair found?) (send name->value :get-hash name)
    (if found? (values (car pair) t (cdr pair))
        (values nil nil nil))))

(defmethod (global-frame :set-value) (name value)
  (send name->value :rem-hash name)
  (send value->name :rem-hash value)
  (send self :insert name value nil))

(defmethod (global-frame :get-name) (value)
  (send value->name :get-hash value))

(defmethod (global-frame :next) ()
  nil)

(defmethod (global-frame :arguments?) ()
  nil)

(defmethod (global-frame :handle) ()
  nil)

(defun make-global-frame ()
  (make-instance 'global-frame :name->value (make-hash-table)
                 :value->name (make-hash-table)))

;; PARTIAL - A partially-specified value.
;;
;;   VALUE - The partially-specified value: a structure of IBL values
;;           and possibly *UNKNOWN values.
;;
;;   PARTIAL-CODE - A structure of nullary lambdas which when invoked return the
;;                  PARTIAL corresponding to the component of the VALUE.
;;
;;   CODE - The code necessary to produce the side effects (if any) and the
;;          fully-specified value at run-time
;;
;;   SIDE-EFFECTS - The list of forms necessary to produce the side-effects
;;                    (if any) at run-time.
;;
;;   KNOWN? - A flag indicating that VALUE is an IBL value.

(defflavor partial (value partial-code code side-effects known?) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-partial (value partial-code code side-effects known?)
  (make-instance 'partial :value value :partial-code partial-code :code code
                 :side-effects side-effects :known? known?))

(defmethod (partial :applicative?) ()
  (null side-effects))

(defun partial? (exp)
```

```
  (typep exp 'partial))

;; UNKNOWN - An unknown value.

(defflavor unknown (type) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-unknown (type)
  (make-instance 'unknown :type type))

(defun unknown? (exp)
  (typep exp 'unknown))

;; BUILT-IN-SYNTAX - A built-in parsing function.

(defflavor built-in-syntax (code) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-built-in-syntax (func)
  (make-instance 'built-in-syntax :code func))

(defun built-in-syntax? (exp)
  (typep exp 'built-in-syntax))

;; USER-SYNTAX - A user-defined parsing function.

(defflavor user-syntax (code) ()
  :gettable-instance-variables
  :initable-instance-variables)

(defun make-user-syntax (func)
  (make-instance 'user-syntax :code func))
```

```
;;;
;;; INTERPRETER ************************************************
;;;

;; START - Initializes everything and starts the interaction loop.

(defun start ()
  (pkg-goto 'iblc)
  (let ((global-env (make-global-env))
        (readtable (copy-readtable)))
    (set-syntax-from-description #/: 'si:alphabetic readtable)
    (set-syntax-macro-char #/' #'(lambda (list-so-far instream)
                                   (list 'back-quote (read instream)))
                           readtable)
    (set-syntax-macro-char #/, #'(lambda (list-so-far instream)
                                   (list 'comma (read instream)))
                           readtable)
    (read-eval-print standard-input global-env)))

;; READ-EVAL-PRINT - The user-interface to the interpretor.

(defun read-eval-print (in-stream env)
  (do ((input nil (progn
                    (terpri)
                    (princ "ibl ->")
                    (parse (read-for-top-level in-stream "quit")))))
      ((equal input "quit") nil)
    (terpri)
    (grind-top-level (unparse (*catch 'error (ibl-eval input env))))))

;; IBL-EVAL - The IBL expression evaluator.

(defun ibl-eval (exp env)
  (selectq (typep exp)
    (def-form (let ((value (ibl-eval (send exp :value) env)))
                (rec-augment env (send exp :name) value
                             (send (send exp :read-only?) :true?))
                value))
    (lambda-form (let ((required (send exp :required))
                       (rest (send exp :rest))
                       (asserts (send exp :asserts)))
                   (make-lambda required rest asserts
                                (make-read-onlys required rest asserts)
                                (send exp :body) env)))
    (sequence-form (car (last (eval-list (send exp :forms) env))))
    (if-form (if (send (ibl-eval (send exp :predicate) env) :true?)
                 (ibl-eval (send exp :consequent) env)
                 (ibl-eval (send exp :alternative) env)))
    (combination (let ((func (ibl-eval (send exp :function) env))
                       (args (eval-list (send exp :args) env)))
                   (if (typep func 'built-in-func)
                       (apply-built-in func args env)
                       (apply-lambda func args))))
    (variable (lookup (send exp :name) (send exp :contours)
                      (let ((closure (send exp :closure)))
                        (if (null closure) env
                            (send (ibl-eval closure env) :env)))))
    (assignment (let ((value (ibl-eval (send exp :value) env)))
                  (update (send exp :name) (send exp :contours)
                          (let ((closure (send exp :closure)))
                            (if (null closure) env
                                (send (ibl-eval closure env) :env)))
                          value)
                  value))
    (otherwise exp)))
```

```
;; EVAL-LIST - Evaluate a list of forms.

(defun eval-list (exps env)
  (if (null exps) nil
      (cons (ibl-eval (first exps) env)
            (eval-list (rest1 exps) env))))

;; APPLY-BUILT-IN - Evaluate a built-in function invocation.

(defun apply-built-in (func args env)
  (type-check args (send func :arg-types) (send func :name))
  (funcall (send func :eval-func) args env))

;; TYPE-CHECK - Type-check the actual arguments against the signature.

(defun type-check (actuals types name)
  (cond ((null types) (if (null actuals) nil
                          (*throw 'error '(too many arguments to ,name))))
        ((null actuals) (*throw 'error '(too few arguments to ,name)))
        ((or (eq (car types) '|ANY:|)
             (eq (car types) (get-type (car actuals))))
         (type-check (cdr actuals) (cdr types) name))
        (t (*throw 'error '(type mismatch in ,name
                                ,(get-type (car actuals)) should be
                                ,(car types))))))

;; APPLY-LAMBDA - Evaluate a user-defined function invocation.

(defun apply-lambda (func args)
  (ibl-eval (send func :body)
            (lambda-augment (send func :env) (send func :required)
                            (send func :rest) (send func :read-onlys)
                            args)))

;; REC-AUGMENT - Add a variable binding to the current environment frame.

(defun rec-augment (frame name value read-only?)
  (send frame :insert name value read-only?))

;; LAMBDA-AUGMENT - Add a new frame with lambda-bindings to the environment.

(defun lambda-augment (env required rest read-onlys args)
  (augment-frame (make-frame env nil nil) required rest args read-onlys))

(defun augment-frame (frame required rest args read-onlys)
  (if (null required) (if (null rest) frame
                          (progn (send frame :insert rest args (first read-onlys))
                                 frame))
      (progn
        (send frame :insert (first required) (first args) (first read-onlys))
        (augment-frame frame (rest1 required) rest (rest1 args)
                       (rest1 read-onlys)))))

;; LOOKUP - Lookup a name in an environment.

(defun lookup (name contours env)
  (cond ((null env) (*throw 'error '(unbound var ,name)))
        ((zerop contours) (multiple-value-bind (value found?)
                              (send env :get-value name)
                            (if found? value
                                (lookup name 0 (send env :next)))))
        (t (lookup name (- contours 1) (send env :next)))))

;; UPDATE - Update a slot in an environment.

(defun update (name contours env value)
```

```
  (cond ((null env) (*throw 'error '(unbound var ,name)))
        ((zerop contours) (multiple-value-bind (old-value found?)
                              (send env :get-value name)
                            (if found? (send env :set-value name value)
                                (update name 0 (send env :next) value))))
        (t (update name (- contours 1) (send env :next) value))))

;; GLOBAL-ENVIRONMENT - The global environment frame.

(defvar global-environment nil)

;; MAKE-GLOBAL-ENV - Create and initialize the top-level environment.

(defun make-global-env ()
  (setq global-environment (make-global-frame))
  (init-built-ins)
  (init-syntax)
  global-environment)

;; INIT-BUILT-INS - Load built-in function definitions into the global environment.

(defun init-built-ins ()
  (mapcar #'(lambda (clause)
              (rec-augment global-environment (first clause)
                           (make-built-in-func (first clause)
                                               (second clause)
                                               (third clause)
                                               (fourth clause)
                                               (fifth clause))
                           t))
          '((load ,#'(lambda (args env)
                       (with-open-file (in-stream (first args))
                         (read-eval-print in-stream env)))
                  ,#'pe-built-in-noeval-default
                  (|STRING:|)
                  nil)
            (save ,#'(lambda (args env)
                       (with-open-file (out-stream (first args) :direction :output)
                         (send (send global-environment :name->value) :map-hash
                               (let-closed ((out-stream out-stream))
                                 #'(lambda (name pair)
                                     (let ((value (car pair))
                                           (read-only? (cdr pair)))
                                       (if (not (or (built-in-func? value)
                                                    (built-in-syntax? value)))
                                           (progn
                                             (grind-top-level
                                               '(def ,name ,(unparse value)
                                                  ,(unparse
                                                     (make-boolean read-only?)))
                                               nil out-stream)
                                             (terpri out-stream)
                                             (terpri out-stream)))))))))
                  ,#'pe-built-in-noeval-default
                  (|STRING:|)
                  nil)
            (apply ,#'(lambda (args env)
                        (ibl-eval (make-combination (first args) (second args))
                                  env))
                   ,#'pe-apply
                   (|ANY:| |PAIR:|)
                   nil)
            (symeval ,#'(lambda (args env)
                          (lookup (first args) 0 env))
                     ,#'pe-built-in-eval-default
                     (|SYMBOL:|)
```

```
        nil)
(car ,#'(lambda (args env) (car (first args)))
     ,#'pe-car
     (|PAIR:|)
     nil)
(cdr ,#'(lambda (args env) (cdr (first args)))
     ,#'pe-cdr
     (|PAIR:|)
     nil)
(cons ,#'(lambda (args env) (cons (first args) (second args)))
      ,#'pe-cons
      (|ANY:| |ANY:|)
      |PAIR:|)
(set!-car ,#'(lambda (args env) (rplaca (first args) (second args)))
          ,#'pe-built-in-noeval-default
          (|PAIR:| |ANY:|)
          |PAIR:|)
(set!-cdr ,#'(lambda (args env) (rplacd (first args) (second args)))
          ,#'pe-built-in-noeval-default
          (|PAIR:| |ANY:|)
          nil)
(eq? ,#'(lambda (args env)
          (make-boolean (eq (first args) (second args))))
     ,#'pe-built-in-eval-default
     (|ANY:| |ANY:|)
     |BOOLEAN:|)
(and ,#'(lambda (args env)
          (make-boolean (and (send (first args) :true?)
                             (send (second args) :true?))))
     ,#'pe-and
     (|BOOLEAN:| |BOOLEAN:|)
     |BOOLEAN:|)
(or ,#'(lambda (args env)
         (make-boolean (or (send (first args) :true?)
                           (send (second args) :true?))))
    ,#'pe-or
    (|BOOLEAN:| |BOOLEAN:|)
    |BOOLEAN:|)
(not ,#'(lambda (args env)
          (make-boolean (not (send (first args) :true?))))
     ,#'pe-built-in-eval-default
     (|BOOLEAN:|)
     |BOOLEAN:|)
(iplus ,#'(lambda (args env) (+ (first args) (second args)))
       ,#'pe-built-in-eval-default
       (|FIXNUM:| |FIXNUM:|)
       |FIXNUM:|)
(iminus ,#'(lambda (args env) (- (first args) (second args)))
        ,#'pe-built-in-eval-default
        (|FIXNUM:| |FIXNUM:|)
        |FIXNUM:|)
(itimes ,#'(lambda (args env) (* (first args) (second args)))
        ,#'pe-built-in-eval-default
        (|FIXNUM:| |FIXNUM:|)
        |FIXNUM:|)
(idiv ,#'(lambda (args env) (// (first args) (second args)))
      ,#'pe-built-in-eval-default
      (|FIXNUM:| |FIXNUM:|)
      |FIXNUM:|)
(irem ,#'(lambda (args env) (\ (first args) (second args)))
      ,#'pe-built-in-eval-default
      (|FIXNUM:| |FIXNUM:|)
      |FIXNUM:|)
(iless? ,#'(lambda (args env)
             (make-boolean (< (first args) (second args))))
        ,#'pe-built-in-eval-default
```

```
        (|FIXNUM:| |FIXNUM:|)
        |BOOLEAN:|)
(igreater? ,#'(lambda (args env)
                (make-boolean (> (first args) (second args))))
           ,#'pe-built-in-eval-default
           (|FIXNUM:| |FIXNUM:|)
           |BOOLEAN:|)
(iequal? ,#'(lambda (args env)
              (make-boolean (= (first args) (second args))))
         ,#'pe-built-in-eval-default
         (|FIXNUM:| |FIXNUM:|)
         |BOOLEAN:|)
(vector-cons ,#'(lambda (args env)
                  (make-array (first args)
                              :initial-value (second args)))
             ,#'pe-built-in-eval-default
             (|FIXNUM:| |ANY:|)
             |VECTOR:|)
(vector-ref ,#'(lambda (args env)
                 (aref (first args) (second args)))
            ,#'pe-built-in-eval-default
            (|VECTOR:| |FIXNUM:|)
            nil)
(vector-size ,#'(lambda (args env) (array-length (first args)))
             ,#'pe-built-in-eval-default
             (|VECTOR:|)
             |FIXNUM:|)
(vector-set! ,#'(lambda (args env)
                  (aset (third args) (first args) (second args)))
             ,#'pe-built-in-noeval-default
             (|VECTOR:| |FIXNUM:| |ANY:|)
             nil)
(print ,#'(lambda (args env)
            (print (unparse (first args))))
       ,#'pe-print
       (|ANY:|)
       nil)
(read ,#'(lambda (args env)
           (read))
      ,#'pe-built-in-noeval-default
      ()
      nil)
(type ,#'(lambda (args env) (get-type (first args)))
      ,#'pe-type
      (|ANY:|)
      |KEYWORD:|)
(pe-func ,#'(lambda (args env) (pe-function-body (first args)))
         ,#'pe-built-in-noeval-default
         (|ANY:|)
         nil)
(pe-exp ,#'(lambda (args env) (pe-exp (parse (first args)) env env))
        ,#'pe-built-in-noeval-default
        (|ANY:|)
        nil)
(pe-all ,#'(lambda (args env)
             (send (send global-environment :name->value) :map-hash
                   #'(lambda (name pair)
                       (print name)
                       (let ((value (car pair)))
                         (cond ((typep value 'lambda)
                                (pe-function-body value))
                               ((typep value 'user-syntax)
                                (pe-function-body
                                  (send value :code))))))))
        ,#'pe-built-in-noeval-default
        ()
```

```
                    nil)
            )
          )
  )

;; GET-TYPE - Get the type of a value, NIL if unknown.

(defun get-type (value)
  (if (null value) '|NIL:|
      (selectq (typep value)
        (lambda '|LAMBDA:|)
        (built-in-func '|BUILT-IN-FUNC:|)
        (built-in-syntax '|BUILT-IN-SYNTAX:|)
        (user-syntax '|USER-SYNTAX:|)
        (boolean '|BOOLEAN:|)
        (unknown (send value :type))
        (:array '|VECTOR:|)
        (:string '|STRING:|)
        (:symbol (if (keyword? value) '|KEYWORD:|
                     '|SYMBOL:|))
        (:list '|PAIR:|)
        (:fixnum '|FIXNUM:|)
        (otherwise nil))))

;; MAKE-READ-ONLYS - Make a boolean list to indicate read-only variables.

(defun make-read-onlys (required rest asserts)
  (make-read-only-list required rest (cdr (find-name 'read-only asserts))))

;; MAKE-READ-ONLY-LIST

(defun make-read-only-list (required rest read-onlys)
  (if (null required) (list (memq rest read-onlys))
      (cons (memq (car required) read-onlys)
            (make-read-only-list (cdr required) rest read-onlys))))
```

```
;;;
;;; PARTIAL EVALUATOR ************************************************
;;;

;; PE-FUNCTION-BODY - Partially evaluate a function body.

(defun pe-function-body (func)
  (let ((required (send func :required))
        (rest (send func :rest)))
    (let ((env1 (lambda-augment (send func :env) required rest
                                (send func :read-onlys)
                                (make-unknowns required rest))))
      (send func :set-body (send (pe-exp (send func :body) env1 env1)
                                 :code)))))

;; PE-EXP - Partially evaluate an expression with respect to the
;;   partial-evaluation-time environment (PE-ENV).  RUN-ENV is the
;;   eventual run-time environment.

(defun pe-exp (exp pe-env run-env)
  (selectq (typep exp)
    (variable
     (let ((name (send exp :name))
           (closure (send exp :closure))
           (contours (send exp :contours)))
       (if closure (pe-var-in-closure name closure pe-env run-env)
           (pe-var name contours pe-env run-env))))
    (combination
     (let ((func (pe-exp (send exp :function) pe-env run-env))
           (args (pe-list (send exp :args) pe-env run-env)))
       (let ((func-value (send func :value))
             (func-code (send func :code)))
         (selectq (typep func-value)
           (built-in-func (funcall (send func-value :pe-func)
                                   func-code func-value args run-env))
           (lambda (pe-lambda func-code func-value args run-env))
           (otherwise (let ((code (make-combination func-code (get-codes args))))
                        (make-partial (make-unknown nil) nil
                                      code (list code) nil)))))))
    (if-form
     (let ((predicate (pe-exp (send exp :predicate) pe-env run-env)))
       (if (send predicate :known?)
           (if (send (send predicate :value) :true?)
               (pe-exp (send exp :consequent) pe-env run-env)
               (pe-exp (send exp :alternative) pe-env run-env))
           (let ((consequent (pe-exp (send exp :consequent) pe-env run-env))
                 (alternative (pe-exp (send exp :alternative) pe-env run-env)))
             (let ((code (make-if-form (send predicate :code)
                                       (send consequent :code)
                                       (send alternative :code)))
                   (applicative? (and (send predicate :applicative?)
                                      (send consequent :applicative?)
                                      (send alternative :applicative?))))
               (make-partial (make-unknown nil) nil code (if applicative? nil
                                                             (list code))
                             nil)))))))
    (def-form
     (let ((name (send exp :name))
           (value (send exp :value))
           (read-only? (send exp :read-only?)))
       (let ((pe-value (pe-exp value pe-env run-env)))
         (rec-augment pe-env name pe-value read-only?)
         (rec-augment run-env name pe-value read-only?)
         (let ((code (make-def-form name (send pe-value :code) read-only?)))
           (make-partial pe-value nil code (list code) t)))))
    (lambda-form
```

```
     (let ((required (send exp :required))
           (rest (send exp :rest))
           (asserts (send exp :asserts))
           (body (send exp :body)))
       (let ((read-onlys (make-read-onlys required rest asserts)))
         (let ((pe-value (pe-exp body
                                 (lambda-augment pe-env required rest read-onlys
                                                 (make-unknowns required rest))
                                 (lambda-augment run-env required rest read-onlys
                                                 (make-unknowns required rest)))))
           (make-partial (make-lambda required rest asserts read-onlys
                                      (send pe-value :code) run-env)
                         nil (make-lambda-form required rest asserts
                                               (send pe-value :code))
                         nil t)))))
     (sequence-form
      (let ((forms (pe-list (send exp :forms) pe-env run-env)))
        (optimize-sequence (butlast forms) (car (last forms)))))
     (assignment
      (let ((name (send exp :name))
            (closure (send exp :closure))
            (contours (send exp :contours))
            (value (send exp :value)))
        (let ((variable (send (if closure (pe-var-in-closure name closure
                                                             pe-env run-env)
                                  (pe-var name contours pe-env run-env))
                              :code))
              (value (send (pe-exp value pe-env run-env) :code)))
          (let ((code (make-assignment (send variable :name)
                                       (send variable :closure)
                                       (send variable :contours) value)))
            (make-partial (make-unknown nil) nil code (list code) nil)))))
     (:list (make-list-literal exp))
     (otherwise (make-constant exp))))

;; MAKE-LIST-LITERAL - Make a PARTIAL for a list literal.

(defun make-list-literal (lit)
  (make-partial lit (if (atom lit) nil
                        (cons (let-closed ((lit (car lit)))
                                #'(lambda () (make-list-literal lit)))
                              (let-closed ((lit (cdr lit)))
                                #'(lambda () (make-list-literal lit))))))
                lit nil t))

;; UP-ENV - Go up the environment CONTOURS frames.

(defun up-env (env contours)
  (if (zerop contours) env
      (up-env (send env :next) (- contours 1))))

;; PE-LIST - Partially evaluate a list of forms.

(defun pe-list (forms pe-env run-env)
  (if (null forms) nil
      (cons (pe-exp (car forms) pe-env run-env)
            (pe-list (cdr forms) pe-env run-env))))

;; PE-LAMBDA - Partially evaluate a user-defined closure invocation.

(defun pe-lambda (proc-code proc-value pe-args run-env)
  (let ((asserts (send proc-value :asserts))
        (code (make-combination proc-code (get-codes pe-args))))
    (cond ((and (memq 'evaluate asserts)
                (all-known? pe-args))
           (let ((value (apply-lambda proc-value (get-values pe-args))))
```

```
            (cond ((immutable? value)
                   (optimize-sequence pe-args (make-constant value)))
                  ((in-env? value run-env)
                   (optimize-sequence pe-args
                                      (make-variable-partial value run-env)))
                  (t (make-partial value nil code (get-imp-codes pe-args)
                                   t)))))
          ((or (memq 'macro-like asserts)
               (and (beta-expand? asserts (lambda-augment
                                             run-env (send proc-value :required)
                                             (send proc-value :rest)
                                             (send proc-value :read-onlys) pe-args))
                    (all-applicative? pe-args)))
           (beta-expand proc-code proc-value pe-args run-env))
          ((memq 'open-specialize asserts)
           (open-specialize proc-code proc-value pe-args run-env))
          (t (let ((applicative? (memq 'applicative asserts)))
               (make-partial (make-unknown (result-type asserts))
                             nil code (if applicative? (get-imp-codes pe-args)
                                                       (list code))
                             nil))))))

;; RESULT-TYPE - The result-type of the function, as declared in the ASSERT list.

(defun result-type (asserts)
  (second (find-name 'returns asserts)))

;; BETA-EXPAND? - Should the invocation be beta-expanded?

(defun beta-expand? (asserts env)
  (and asserts
       (or (memq 'beta-expand asserts)
           (let ((test (cadr (find-name 'beta-expand asserts))))
             (and test
                  (cond ((atom test) (send (pe-exp (parse test) env env) :known?))
                        ((eq (car test) 'b-or)
                         (one-known? (pe-list (parse-list (cdr test)) env env)))
                        ((eq (car test) 'b-and)
                         (all-known? (pe-list (parse-list (cdr test)) env env)))
                        (t (send (pe-exp (parse test) env env) :known?))))))))

;; FIND-NAME - Find the list corresponding to a keyword.

(defun find-name (keyword lst)
  (and lst (if (and (listp (car lst))
                    (eq (caar lst) keyword))
               (car lst)
               (find-name keyword (cdr lst)))))

;; BETA-EXPAND - Beta-expand a function invocation.

(defun beta-expand (proc-code proc-value pe-args run-env)
    (pe-exp (send proc-value :body)
            (pe-lambda-augment (send proc-value :env)
                               (send proc-value :required)
                               (send proc-value :rest) pe-args
                               (send proc-value :read-onlys) proc-code t)
            run-env))

;; OPEN-SPECIALIZE - Open-specialize a function invocation.

(defun open-specialize (proc-code proc-value pe-args run-env)
  (let ((env (send proc-value :env))
        (required (send proc-value :required))
        (rest (send proc-value :rest))
        (read-onlys (send proc-value :read-onlys)))
```

```
    (let ((pe-body (pe-exp (send proc-value :body)
                           (pe-lambda-augment env required rest pe-args
                                              read-onlys proc-code nil)
                           (lambda-augment run-env required rest
                                           read-onlys pe-args))))
      (let ((value (send pe-body :value))
            (applicative? (send pe-body :applicative?))
            (known? (send pe-body :known?)))
        (if (and known? (immutable? value) applicative?)
            (optimize-sequence pe-args (make-constant value))
            (let ((code (make-combination
                          (make-lambda-form required rest
                                            (send proc-value :asserts)
                                            (send pe-body :code))
                          (get-codes pe-args))))
              (make-partial value nil code (if applicative?
                                               (get-imp-codes pe-args)
                                               (list code))
                            known?)))))))

;; PE-LAMBDA-AUGMENT - Augment the environment for expansion or specialization.

(defun pe-lambda-augment (env required rest pe-args read-onlys handle args?)
  (pe-augment-frame (make-frame env handle args?)
                    required rest pe-args read-onlys))

(defun pe-augment-frame (frame required rest pe-args read-onlys)
  (if (null required) (if (null rest) frame
                          (progn (send frame :insert rest (rest-list pe-args)
                                       (first read-onlys))
                                 frame))
      (progn (send frame :insert (first required) (first pe-args)
                   (first read-onlys))
             (pe-augment-frame frame (rest1 required) rest (rest1 pe-args)
                               (rest1 read-onlys)))))

;; REST-LIST - Make a partially-specified object denoting a list of REST args.

(defun rest-list (pe-args)
  (make-partial (get-values pe-args)
                (if (not (all-applicative? pe-args)) nil
                    (cons (let-closed ((args pe-args))
                            #'(lambda () (car args)))
                          (let-closed ((args pe-args))
                            #'(lambda () (rest-list (cdr args))))))
                (make-list-code (get-codes pe-args))
                (get-imp-codes pe-args) (all-known? pe-args)))

;; MAKE-LIST-CODE - Make the code for a list of codes.

(defun make-list-code (codes)
  (if (null codes) nil
      (make-combination (parse '(bif cons))
                        (list (car codes)
                              (make-list-code (cdr codes))))))

;; PE-VAR - Partially evaluate a variable reference.

(defun pe-var (name skip pe-env run-env)
  (multiple-value-bind (value contours read-only? argument? handle)
      (pe-lookup name skip pe-env 0 nil)
    (cond ((and read-only? (immutable? value))
           (make-constant (get-value value)))
          (argument? (if (zerop contours) value
                         (pe-exp (send value :code)
                                 (up-env run-env contours) run-env)))
```

```
        (t (let ((code (find-variable name value handle run-env)))
             (if read-only? (make-partial (get-value value) nil code nil
                                          (if (partial? value) (send value :known?)
                                              (not (unknown? value))))
                 (make-partial (make-unknown nil) nil code nil nil)))))))

;; PE-VAR-IN-CLOSURE - Partially evaluate a variable reference within a
;;   closure's  environment.

(defun pe-var-in-closure (name closure pe-env run-env)
  (let ((pe-closure (pe-exp closure pe-env run-env)))
    (let ((closure-value (send pe-closure :value))
          (closure-code (send pe-closure :code)))
      (if (send pe-closure :known?)
          (multiple-value-bind (value contours read-only?)
              (pe-lookup name 0 (send closure-value :env) 0 nil)
            (if (and read-only? (immutable? value))
                (make-constant (get-value value))
                (let ((code (make-variable name closure-code contours)))
                  (if read-only? (make-partial (get-value value) nil code nil
                                               (if (partial? value) (send value :known?)
                                                   (not (unknown? value))))
                      (make-partial (make-unknown nil) nil code nil nil)))))
          (make-partial (make-unknown nil) nil
                        (make-variable name closure-code 0)
                        nil nil))))))

;; PE-LOOKUP - Instrumented version of LOOKUP, returns
;;   (VALUE CONTOURS READ-ONLY? ARGUMENT? HANDLE).

(defun pe-lookup (name skip env num-contours handle)
  (if (null env) (values (make-unknown nil) num-contours nil nil nil)
      (multiple-value-bind (value found? read-only?)
          (if (zerop skip) (send env :get-value name)
              (values nil nil nil))
        (if found? (values value num-contours read-only?
                           (send env :arguments?) handle)
            (pe-lookup name (if (zerop skip) 0 (- skip 1)) (send env :next)
                       (1+ num-contours)
                       (let ((new-handle (send env :handle)))
                         (or new-handle handle)))))))

;; FIND-VARIABLE - Makes the appropriate variable code for a value by looking
;;   up the value in the RUN-ENV.

(defun find-variable (name value handle run-env)
  (multiple-value-bind (new-name found? contours)
      (reverse-lookup value run-env 0)
    (if found? (make-variable new-name nil contours)
        (make-variable name handle 0))))

;; REVERSE-LOOKUP - Given a value, tries to find a name, returns
;;   (NAME FOUND? CONTOURS).

(defun reverse-lookup (value env contours)
  (if (null env) (values nil nil nil)
      (multiple-value-bind (name found?)
          (send env :get-name value)
        (if found? (values name t contours)
            (reverse-lookup value (send env :next) (1+ contours))))))

;; IN-ENV? - Is a value in the environment?

(defun in-env? (value env)
  (multiple-value-bind (name found?)
      (reverse-lookup value env 0)
```

```
    found?))

;; MAKE-VARIABLE-PARTIAL - Make a variable for a value in the environment.

(defun make-variable-partial (value env)
  (make-partial value nil (find-variable nil value nil env) nil t))

;; PE-BUILT-IN-EVAL-DEFAULT - Partially evaluate a built-in function,
;;   eval if possible.

(defun pe-built-in-eval-default (proc-code proc-value pe-args run-env)
  (let ((value (if (all-known? pe-args)
                   (apply-built-in proc-value (get-values pe-args) run-env)
                   (make-unknown (send proc-value :return-type)))))
    (cond ((immutable? value)
           (optimize-sequence pe-args (make-constant value)))
          ((in-env? value run-env)
           (optimize-sequence pe-args (make-variable-partial value run-env)))
          (t (make-partial value nil
                           (make-combination proc-code (get-codes pe-args))
                           (get-imp-codes pe-args) (all-known? pe-args))))))

;; PE-BUILT-IN-NOEVAL-DEFAULT - Partially evaluate a built-in function,
;;   don't eval because of side-effects.

(defun pe-built-in-noeval-default (proc-code proc-value pe-args run-env)
  (let ((code (make-combination proc-code (get-codes pe-args))))
    (make-partial (make-unknown (send proc-value :return-type)) nil
                  code (list code) nil)))

;; PE-PRINT - Partially evaluate an invocation of PRINT.

(defun pe-print (proc-code proc-value pe-args run-env)
  (let ((arg (first pe-args))
        (code (make-combination proc-code (get-codes pe-args))))
    (make-partial (send arg :value) nil code (list code) (send arg :known?))))

;; PE-AND - partially evaluate an invocation of AND.

(defun pe-and (proc-code proc-value pe-args run-env)
  (cond ((send (first pe-args) :known?)
         (if (send (send (first pe-args) :value) :true?)
             (optimize-sequence (list (first pe-args)) (second pe-args))
             (optimize-sequence pe-args (make-constant (make-boolean nil)))))
        ((send (second pe-args) :known?)
         (if (send (send (second pe-args) :value) :true?)
             (optimize-sequence (list (second pe-args)) (first pe-args))
             (optimize-sequence pe-args (make-constant (make-boolean nil)))))
        (t (let ((code (make-combination proc-code (get-codes pe-args))))
             (make-partial (make-unknown '|BOOLEAN:|) nil code
                           (get-imp-codes pe-args) nil)))))

;; PE-OR - Partially evaluate an invocation of OR.

(defun pe-or (proc-code proc-value pe-args run-env)
  (cond ((send (first pe-args) :known?)
         (if (send (send (first pe-args) :value) :true?)
             (optimize-sequence pe-args (make-constant (make-boolean t)))
             (optimize-sequence (list (first pe-args)) (second pe-args))))
        ((send (second pe-args) :known?)
         (if (send (send (second pe-args) :value) :true?)
             (optimize-sequence pe-args (make-constant (make-boolean t)))
             (optimize-sequence (list (second pe-args)) (first pe-args))))
        (t (let ((code (make-combination proc-code (get-codes pe-args))))
             (make-partial (make-unknown '|BOOLEAN:|) nil code
                           (get-imp-codes pe-args) nil)))))
```

```
;; PE-CAR - Partially evaluate an invocation of CAR.

(defun pe-car (proc-code proc-value pe-args run-env)
  (let ((arg (first pe-args)))
    (if (listp (send arg :partial-code)) (funcall (car (send arg :partial-code)))
        (let ((value (if (listp (send arg :value)) (car (send arg :value))
                         (make-unknown nil))))
          (cond ((immutable? value)
                 (optimize-sequence pe-args (make-constant value)))
                ((in-env? value run-env)
                 (optimize-sequence pe-args (make-variable-partial value run-env)))
                (t (make-partial value nil
                                 (make-combination proc-code (get-codes pe-args))
                                 (send arg :side-effects) (send arg :known?)))))))))

;; PE-CDR - Partially evaluate an invocation of CDR.

(defun pe-cdr (proc-code proc-value pe-args run-env)
  (let ((arg (first pe-args)))
    (if (listp (send arg :partial-code)) (funcall (cdr (send arg :partial-code)))
        (let ((value (if (listp (send arg :value)) (cdr (send arg :value))
                         (make-unknown nil))))
          (cond ((immutable? value)
                 (optimize-sequence pe-args (make-constant value)))
                ((in-env? value run-env)
                 (optimize-sequence pe-args (make-variable-partial value run-env)))
                (t (make-partial value nil
                                 (make-combination proc-code (get-codes pe-args))
                                 (send arg :side-effects) (send arg :known?)))))))))

;; PE-CONS - Partially evaluate an invocation of CONS.

(defun pe-cons (proc-code proc-value pe-args run-env)
  (let ((arg1 (first pe-args))
        (arg2 (second pe-args)))
    (make-partial (cons (send arg1 :value) (send arg2 :value))
                  (cons (let-closed ((larg1 arg1)
                                     (larg2 arg2))
                          #'(lambda () (optimize-sequence (list larg2) larg1)))
                        (let-closed ((larg1 arg1)
                                     (larg2 arg2))
                          #'(lambda () (optimize-sequence (list larg1) larg2))))
                  (make-combination proc-code (get-codes pe-args))
                  (get-imp-codes pe-args)            ; side-effects
                  (all-known? pe-args))))

;; PE-TYPE - Partially evaluate an invocation of TYPE.

(defun pe-type (proc-code proc-value pe-args run-env)
  (let ((arg (first pe-args)))
    (let ((result (get-type (send arg :value))))
      (if result (optimize-sequence pe-args (make-constant result))
          (make-partial (make-unknown '|KEYWORD:|) nil
                        (make-combination proc-code (get-codes pe-args))
                        (send arg :side-effects) nil))))))

;; PE-APPLY - Partially evaluate an invocation of APPLY.

(defun pe-apply (proc-code proc-value pe-args run-env)
  (let ((func (first pe-args))
        (nospread-args (second pe-args)))
    (if (and (send func :known?)
             (send nospread-args :known?)
             (send nospread-args :applicative?))
        (let ((func-value (send func :value))
              (func-code (send func :code))
```

```
            (args (spread-args (send nospread-args :value) nospread-args run-env
                               (lookup 'car 0 global-environment)
                               (lookup 'cdr 0 global-environment))))
          (selectq (typep func-value)
            (built-in-func (funcall (send func-value :pe-func)
                                    func-code func-value args run-env))
            (lambda (pe-lambda func-code func-value args run-env))
            (otherwise (let ((code (make-combination func-code (get-codes args))))
                         (make-partial (make-unknown nil) nil code
                                       (list code) nil)))))
        (let ((code (make-combination proc-code (get-codes pe-args))))
          (make-partial (make-unknown nil) nil code (list code) nil))))

;; SPREAD-ARGS - Given a PARTIAL representing an argument-list, and
;;   the value of the argument-list, create a list of PARTIALs corresponding
;;   to that argument-list.

(defun spread-args (args nospread run-env car-value cdr-value)
  (if (null args) nil
      (cons (pe-car car-value car-value (list nospread) run-env)
            (spread-args (cdr args)
                         (pe-cdr cdr-value cdr-value (list nospread) run-env)
                         run-env car-value cdr-value))))

;; OPTIMIZE-SEQUENCE - Make an optimized SEQUENCE form by eliminating applicative
;;   sub-forms.

(defun optimize-sequence (forms result)
  (let ((imp-codes (get-imp-codes forms)))
    (if (null imp-codes) result
        (make-partial (send result :value) nil
                      (make-sequence-form '(,@imp-codes ,(send result :code)))
                      '(,@imp-codes ,@(send result :side-effects))
                      (send result :known?)))))

;; GET-IMP-CODES - Get the list of imperative expressions from a list of
;;   partially-evaluated forms.

(defun get-imp-codes (forms)
  (apply #'append (mapcar #'(lambda (form)
                              (send form :side-effects))
                          forms)))

;; MAKE-UNKNOWNS - Makes a list of unknown values corresponding to the input.

(defun make-unknowns (required rest)
  (if (null required) (make-unknown nil)  ; for the REST parameter (if any)
      (cons (make-unknown nil)
            (make-unknowns (cdr required) rest))))

;; COLLECT - Collect all items from a list that pass a predicate.

(defun collect (lst pred)
  (cond ((null lst) nil)
        ((funcall pred (car lst)) (cons (car lst) (collect (cdr lst) pred)))
        (t (collect (cdr lst) pred))))

;; ALL? - Checks if all elements of a list pass a predicate.

(defun all? (lst pred)
  (if (null lst) t
      (and (funcall pred (car lst))
           (all? (cdr lst) pred))))

;; ONE? - Checks if at least one element of a list passes a predicate.
```

```
(defun one? (lst pred)
  (if (null lst) nil
      (or (funcall pred (car lst))
          (one? (cdr lst) pred))))

;; GET-CODES - Get the code part of a list of partially evaluated values.

(defun get-codes (values)
  (mapcar #'(lambda (value) (send value :code)) values))

;; GET-VALUES - Get the value part of a list of partially evaluated values.

(defun get-values (values)
  (mapcar #'(lambda (value) (send value :value)) values))

;; ALL-KNOWN? - Checks if all values are known.

(defun all-known? (values)
  (all? values #'(lambda (value) (send value :known?))))

;; ONE-KNOWN? - Checks if at least one value is known.

(defun one-known? (values)
  (one? values #'(lambda (value) (send value :known?))))

;; ALL-APPLICATIVE? - Checks if all values are applicative.

(defun all-applicative? (values)
  (all? values #'(lambda (value) (send value :applicative?))))

;; MAKE-CONSTANT - Make a partially-specified value that denotes a constant.

(defun make-constant (value)
  (make-partial value nil value nil t))

;; GET-VALUE -

(defun get-value (exp)
  (if (typep exp 'partial) (send exp :value)
      exp))

;; IMMUTABLE? - Is a value immutable?

(defun immutable? (value)
  (or (fixnum? value)
      (nil? value)
      (boolean? value)
      (string? value)
      (built-in-func? value)
      (symbol? value)
      (keyword? value)))
```

References

[1] Allen, F.E. and Cocke, J.
A Catalogue of Optimizing Transformations.
In Randall Rustin (editor), *Design and Optimization of Compilers*, pages 1-30. Prentice-Hall, Inc., 1971.

[2] Allen, F.E. and Carter, J.L. and Harrison, W.H. et al.
The Experimental Compiling Systems Project.
Technical Report RC 6718 (# 28922), IBM Thomas J. Watson Research Center, September, 1977.

[3] Babad, Yair M. and Hoffer, Jeffrey A.
Even No Data Has a Value.
Communications of the ACM 27(8):748-758, August, 1984.

[4] Beckman, L. et al.
A Partial Evaluator, and its Use as a Programming Tool.
Artificial Intelligence (7):319-357, 1976.

[5] Boehm, H. and Demers, A. and Donahue, J.
An Informal Description of Russell.
Technical Report 80-430, Dept. of Computer Science, Cornell University, Ithaca, NY, October, 1980.

[6] Borning, Alan H. and Ingalls, Daniel H. H.
Multiple Inheritance in Smalltalk-80.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 234-237. American Association for Artificial Intelligence, August, 1982.

[7] Carter, J. Lawrence.
A Case Study of a New Code Generation Technique for Compilers.
Communications of the ACM 20(12):914-920, December, 1977.

[8] Chow, Frederick C.
A Portable Machine-Independent Global Optimizer - Design and Measurements.
Technical Report 83-254, Computer Systems Laboratory, Stanford University, December, 1983.

[9] Cocke, John.
Partial Execution.
draft of paper - IBM Yorktown.

[10] Darlington, J. and Burstall, R.M.
A System which Automatically Improves Programs.
In *Third International Joint Conference on Artificial Intelligence*, pages 479-485. International Joint Council on Artificial Intelligence, 1973.

[11] Demers, A. and Donahue, J.
The Russell Semantics: An Exercise in Abstract Data Types.
Technical Report 80-431, Dept. of Computer Science, Cornell University, Ithaca, NY, September, 1980.

[12] Deutsch, L. Peter.
An Interactive Program Verifier.
Technical Report CSL-73-1, Xerox Palo Alto Research Center, 1973.

[13] Ershov, A.P.
On the Essence of Compilation.
In *IFIP Working Conference on Formal description of programming concepts.* IFIP, August, 1977.

[14] Ershov, A.P. and Intkin, V.E.
Correctness of Mixed Computation in Algol-like Programs.
Lecture Notes in Computer Science. Volume 53.*Proceedings of the 6th MFCS Symposium.*
Springer-Verlag, 1977.

[15] Ershov, A.P.
On the partial computation principle.
Information Processing Letters 6(2):38-41, April, 1977.

[16] Goldberg, A. and Robson, D.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, MA, 1983.

[17] Haraldsson, Anders.
A Program Manipulation System Based on Partial Evaluation.
PhD thesis, Linkoping University, 1977.

[18] Haraldsson, Anders.
A Partial Evaluator, and Its Use for Compiling Iterative Statements in Lisp.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 195-202. ACM, January, 1978.

[19] Harrison, William H.
A New Strategy for Code Generation - the General-Purpose Optimizing Compiler.
IEEE Transactions on Softare Engineering SE-5(4):367-373, July, 1979.

[20] Herlihy, Maurice P.
Transmitting Abstract Values in Messages.
Technical Report 234, MIT Laboratory for Computer Science, 1980.

[21] Kernighan, Brian W. and Ritchie, Dennis M.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, NJ, 1978.

[22] Komorowski, H. Jan.
A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.
PhD thesis, Linkoping University, 1981.

[23] Komorowski, H. Jan.
Partial Evaluation as a means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the case of Prolog.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 255-267. ACM, January, 1982.

[24] Liskov, B. et al.
Lecture Notes in Computer Science. Volume 114: *CLU Reference Manual.*
Springer-Verlag, Berlin, 1981.

[25] Lombardi, L.A. and Raphael, B.
Lisp as the Language for an Incremental Computer.
In Berkley, E. and Bobrow, D. (editors), *The Programming Language Lisp: Its Operation and Application*. MIT Press, Cambridge, MA, 1964.

[26] Moon, D. and Stallman, R.M. and Weinreb, D.
Lisp Machine Manual
1983.

[27] Reynolds, John C.
Definitional Interpreters for Higher-Order Programming Languages.
ACM Conference Proceedings 1972.

[28] Scheifler, Robert W.
An Analysis of Inline Substitution for a Structured Programming Language.
Communications of the ACM 20(9):647-654, September, 1977.

[29] Scherlis, W.L.
Program Improvement by Internal Specialization.
In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 41-49. ACM, January, 1981.

[30] Steele, G. and Sussman, G.
Lambda: The Ultimate Imperative.
AI Memo 353, Massachusetts Institute of Technology, 1976.

[31] Steele, G.
Lambda: The Ultimate Declarative.
AI Memo 379, Massachusetts Institute of Technology, 1976.

[32] Steele, G.
Debunking The Expensive Procedure Call Myth or, Procedure Call Implementations Considered Harmful or, Lambda: The Ultimate Goto.
AI Memo 443, Massachusetts Institute of Technology, 1977.

[33] Steele, G. and Sussman, G.
The Revised Report on Scheme, A Dialect of LISP.
AI Memo 452, Massachusetts Institute of Technology, 1978.

[34] Steele, Guy L.
Rabbit: A Compiler for Scheme (A Study in Compiler Optimization).
Technical Report AI-474, Massachusetts Institute of Technology, March, 1978.

[35] Sussman, G. and Steele, G.
Scheme: An Interpreter for Extended Lambda Calculus.
AI Memo 349, Massachusetts Institute of Technology, 1975.

[36] *Lisp Machine Summary, 3600 Edition*
Symbolics, Inc., Cambridge, MA, 1983.

[37] Teitelman, W.
INTERLISP Reference Manual
Xerox PARC, 1974.

[38] Wegbreit, Ben.
The Treatment of Data Types in EL1.
Communications Of The ACM 17(5):251-264, May, 1974.

[39] Winston, P.H. and Horn, B.K.P.
LISP.
Addison-Wesley, Reading, MA, 1983.

[40] Wirth, N.
The Programming Language Pascal.
Acta Informatica (1):35-63, 1971.