A METHODOLOGY FOR INVESTIGATING EVALUATION MODELS FOR FUNCTIONAL PROGRAMS

by

Vojislav Radonjić B.A.Sc.

a thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

ISBN 0-612-00564-X

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

The central aspect of implementing a functional programming language is the design of an evaluation model: an abstract view of how the underlying machine executes functional programs. To make informed decisions in designing an evaluation model, the implementation designer needs to have a good understanding of mechanisms constituting the available evaluation models and their relationship to functional programming and computer architecture.

This thesis provides one way of arriving at that understanding through analyzing the *execution behavior* of a set of functional programs on implementations based on two contrasting evaluation models: an *environment* one and a *reduction* one. The essential difference between the environment and reduction evaluation models, is that the former centralizes the dynamic context[1] while the latter distributes it. The consequences of choosing either to centralize or distribute the dynamic context are studied using data on execution behavior of functional programs.

In the experimental part of the thesis, environment and reduction evaluators are built and instrumented to generate execution profiles, which contain measures of computational resource usage. It is demonstrated how the execution profiles can be used to compute costs incurred by the two evaluators.

[1] In general, the dynamic context of a computation is that part of the overall context which changes during a computation and, therefore, reflects the progress of the computation.

Acknowledgments

I am deeply indebted to my supervisor Dr. Moshe Krieger for his generous support and constant encouragement.

I would like to thank my family, and in particular my father, who has stood by me throughout.

I thank my friend and colleague Charles A. Gauthier for his readiness to help and prod on.

I thank my special friend Helen Slegr for her company, generosity, and understanding.

Finally, I thank the Department of Electrical Engineering of the University of Ottawa, and the National Science and Engineering Research Council (NSERC) of Canada for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Early on in the era of electronic computers, programming language designers and implementors had to work within the boundaries imposed by the von Neumann hardware architecture. This strongly affected what programming languages could be efficiently implemented. In fact, it has been said that our continued dependency on von Neumann computers has seriously limited development of non-von Neumann, or alternative, programming languages [4].

The dominance of von Neumann computers may have limited, but it certainly did not extinguish the development of alternative programming languages. As early as 1960 it was clear that, at least in one application area, namely symbolic computing, the dominant language of the time, Fortran, would not do and that there was a need for a more expressive programming language.

The earliest significant result of the efforts at designing a programming language for symbolic computing was Lisp [24] [17]. The honours go to Lisp for demonstrating the viability of an alternative, mostly functional, style of programming despite the efficiency problems of its implementation on von Neumann style computers. Since the emergence of Lisp mary other programming languages have appeared that supported programming in one or more of the alternative paradigms common today: functional, logic, and object based. Paralleling the rise of these new languages was work in language implementations and in language oriented computer architectures (both sequential and parallel). Once the viability of alternative programming languages was established, computer architects could start developing hardware structures for supporting alternative computational models.

It is widely recognized that high performance programming systems can only be attained by designing computer architectures that make it feasible for compilers to generate code that will execute fast [26]. RISC architectures owe their success in large measure to this philosophy which recognizes that most of the code executed by today's processors is generated by compilers and that, consequently, the job of a computer architect is to provide a simple and useful execution platform for the design of a high level language implementation. To find out the simplest yet useful set of architectural features, the computer architect needs to know the static and dynamic characteristics of code generated from typical applications, which in turn implies an ability to study execution behavior of these applications.

Analysis of execution behavior for purposes of designing commercially successful high-performance processors has been confined to applications coded in conventional imperative programming languages. No such analysis by designers of commercial processors has been done for functional programming applications even though, for a significant number of symbolic and numeric applications, functional programs have definite advantages over imperative ones—they can be written quicker, are concise, are higher level, are amenable to formal reasoning and analysis, and are amenable to parallel execution [17]. In the Lisp community functional programming techniques have proven themselves over many years and over numerous applications. Despite that, however, there remains a problem of relatively high execution costs incurred by functional programs on conventional processors when compared to optimized imperative programming language implementations. High-performance RISC architectures offer no new hope to functional programming because they were designed for supporting conventional imperative programming languages. As a result, the performance gap between conventional and functional programming languages remains alive on most general-purpose computing platforms.

It is clearly essential to future success of functional programming languages that there be a significant improvement in the efficiency and performance of their implementations. This thesis proposes an approach to analysis, comparison and prototyping of functional programming implementations that leads towards that goal. The approach is based on the use of quantitative data on execution behavior of typical functional applications in analyzing and comparing implementation approaches, and as the starting point for development of efficient functional programming language

implementations.

1.1 A Perspective on Programming Languages and their Implementation

This section is the first step in the developing the approach: it gives a perspective on the problem of programming language implementation in general.

A computer programming language is both a *medium for communicating* with a computer and a *system of abstractions* that is used to build programs that execute on that computer. Viewed as a communication medium, a programming language is a set of syntax rules that specify valid sentences or programs; the syntactic rules define the *form*, as opposed to the *content*, of programs.

A programming language is not simply an accidental gathering of concepts; rather, it is a *system*, an organized collection of abstractions with one, unifying function: to provide building blocks for designing programs. The origin of requirements for this system of abstractions is the expected application domain of the programming language. The nature of the applications for which the programs are built ultimately determine the characteristics of those abstractions that will be most useful. The origin of constraints, on the other hand, is the machine that executes the programs, because it determines what abstractions can or cannot be supported effectively. Accordingly, a programming language is the result of meeting the demands of the application area with a system of abstractions that can be implemented within the constraints imposed by the underlying machine.

An understanding of the relationship between applications and abstractions is critical to the design of programming languages. This understanding is also critical to the programmer, because it is he that will build programs out of these abstractions. Furthermore, if he is to build programs that do what is intended, the programmer needs to have a full understanding of the semantics of both the abstractions and of their combinations.

The design problem in programming language implemention is the construction of a machine that can execute programs expressed in the programming language, or some intermediate language into which the programs are first translated [27]. Normally, for

high-level programming languages, this executor consists of a hierarchy of machines, each with its own language [27]. The programming language, its translator and executor constitute a *programming system.*[1]

A useful way of defining a programming language semantics is as a set of evaluation rules for the expressions of the language. Evaluation rules can be used by the machine designer to determine what it *means* to execute programs, i.e. help him understand the behavior of programming language abstractions. These same evaluation rules can also help the programmer build programs that behave as he intended them to. This type of programming language semantics is normally termed operational semantics, which is in contrast to denotational or axiomatic semantics [20] [21], that are mathematical descriptions of meaning. These formal semantics, being an abstract definition of meaning, are independent of any particular implementation approach.

An *evaluation model* for a programming language is a high-level model of execution behavior of the language components and their combinations. The model specifies evaluation rules for the language, which define the behavior of any evaluators based on that evaluation model. In the hierarchy of machines that together constitute the executor for the programming language, the evaluator is the first level machine, which is normally referred to as an *abstract* or *virtual machine* [27]. At the bottom of the hierarchy is the hardware executor, and in the middle, there may be any number of intermediate machines.

Each layer of a programming system is a machine which internally consists of a processor-memory pair, but externally behaves like a memory responding to requests [28]. Or each layer has a processor that interprets outside requests, and submits requests to its memory, which is itself a processor-memory pair. The whole system looks like

> ...a hierarchy of data abstractions, each implemented in terms of a more primitive one. [28]

Under this view, an evaluation model for a programming language is a memory that

[1] A broader definition of a programming system would also include various program developments tools such as debuggers and smart editors. In programming, the tools are often just as important as the programming language itself; however, here the definition of a programming system is restricted because the focus in this thesis is on design of executors for the class of functional programming languages.

responds to requests; or, equivalently, a data abstraction where the actual datum is an expression in the language and evaluation is an operation on it. Consequently, comparing two different evaluation models is like comparing two different data abstractions for the "same" data object. This data abstraction view is easily extended to the whole programming system, which under the broader definition above, includes the program development environment and the executor. That is, in a programming system for a language, the datum is a program in the language and the operations may include the following: store, edit, retrieve, compile, evaluate or execute, debug, instrument, optimize, visualize and so on.

1.2 The Purpose and the Methodology

In the terminology of the previous section, the purpose of this thesis is to develop an approach to analysis, comparison and prototyping of evaluation models for functional programming; an approach that will lead toward an improvement in the efficiency and performance of functional programming implementations.

The analysis aims to understand the mechanisms used by the two evaluation models to satisfy the requirements of functional programming languages. The language used to embody the essential abstractions of functional programming is fScheme, a functional subset of the Scheme dialect of Lisp[2]. fScheme is a language that consists of a small set of abstractions centered on the twin concepts of function and data as they appear in functional programming (significantly different from their counterparts in imperative programming). This permits the thesis to focus on the implementation of the core elements of functional programming. While the analysis mainly deals with the design alternatives at the evaluation model level, it can be extended to include an examination of how possible hardware machine designs affect the feasibility of the evaluation models; and, hence, how hardware machine designs indirectly, through the evaluation model, affect the feasibility of the abstractions that make fScheme.

The main tool for analysis, comparison and prototyping consists of two instrumented evaluators for functional programs. These evaluators generate execution profiles that characterize the computational activities in evaluating fScheme programs.

[2]Scheme is discussed in Chapter 2 and Chapter 4, and fScheme in Chapter 4.

The execution behavior of fScheme programs is used to study how the various abstractions that constitute fScheme can be implemented, and how expensive these implementations are in terms of computing resources. In this way the consequences of design decisions at the evaluation model level are exposed, while at the same time providing an estimate of the cost of execution for functional programs.

In this thesis, execution behavior is central to answering the question

> How effective is an evaluation model in bridging the gap between a programming language and an underlying computer?

The answers will be useful in understanding, and, consequently, improving both the evaluation model and the computer. Execution behavior can also be used in estimating the costs of new language features.

1.3 Thesis Outline

Chapter 2 presents the basic system of functional programming abstractions and some of the major languages that embody them. It deals with such issues as the concepts of function and data in functional programming, state in imperative versus state in functional programming, and higher order functions.

Chapter 3 discusses evaluation models and computer architectures for functional programming. It classifies the evaluation models into those that centralize their dynamic context and those that distribute it.

Chapter 4 defines the fScheme programming language and its operational semantics in terms of the environment evaluation model (an archetypical centralized implementation approach).

Chapter 5 defines the Environment and Reduction Evaluators for fScheme which embody, respectively, the environment and reduction evaluation models.

Chapter 6 presents the execution profiles of each of the benchmarks running on the two machines, and a detailed cost analysis based on one of these execution profiles. This chapter concludes with the reasons for the differences in computational costs incurred by the two evaluators. It is also suggested where to concentrate when optimizing for space and time costs in the two evaluation models.

Chapter 7 concludes the thesis and suggest directions for further research.

Chapter 2

Functional Programming

The purpose of this chapter is to present (1) the key concepts of functional programming in the context of environment model evaluation rules and (2) some of the programming languages based on these concepts.

2.1 The Functional Programming System of Abstractions

Functional programming is building programs using a system of abstractions centered on the twin concepts of *function* and *data*. Accordingly, the core concepts of functional programming languages are *functional abstraction* and *function composition*, and their analogues in the data domain, *data abstraction* and *data composition*.

This section presents the basic abstractions that appear in functional programming languages with emphasis on their meaning rather than form. One notable abstraction that is missing from functional programming languages is the concept of mutable state and the associated assignment operation. The consequences of this absence are discussed in this section.

What follows, is a somewhat more extensive presentation of basic elements of functional programming than may appear necessary at first glance. The reason for dwelling on the fundamentals of functional programming is to provide sufficient background for understanding the primary requirements placed on any proposed evaluation model for functional programming.

2.1.1 Functional Abstraction

The purpose of this section is to elaborate and illustrate the concept of a function as it appears in functional programming. In computation, a function is an abstraction of values over some common pattern of operations [17].The power of the function as a building block for programs stems from the fact that it permits the use of the same pattern of operations (expression) for different values (arguments). For example, rather than writing[1] over and over again every time we want to double a number

```
(+ 3 3) (+ 4 4) (+ 5 5)
```

we can, instead, define a function `double`[2]

```
(define (double x) (+ x x))
```

to do the same operation for any number `x`. The expression for applying `double` to number `3` is

```
(double 3)
```

In the act of defining the function `double`, the actual numerical values have been abstracted away and in their place a variable `x` was used; the operations used in doubling a number are hidden inside the function, so that a user of the function only needs to know the function name and *what* it does rather than *how* it does it (the body of `double` could have been `(* 2 x)` rather than `(+ x x)`). This is the essence of programming language abstractions: hide the implementation details of whatever

[1]Unless otherwise specified, programming examples are in the syntax of the Scheme dialect of Lisp, both of which are discussed in the following two sections.

[2]Scheme syntax for function definition is

`(define (`⟨name⟩ ⟨variables⟩`)` ⟨body⟩`)`

and for function application is

`(`⟨name⟩ ⟨arguments⟩`)`

where ⟨name⟩, ⟨variables⟩, ⟨body⟩, and ⟨arguments⟩ are part of the language used for describing the syntax of Scheme, not actual Scheme expressions. The report on Scheme [6] refers to them as syntactic variables, i.e. variables which denote valid Scheme expressions.

computational object is being defined and make visible to the user of the object only those things that are necessary in using the object or in building with it other, more complex, computational objects.

For example, to use `double` in building `product-of-doubles`, the programmer needs to know how to apply `double` to an argument, and how to combine simpler functions into more complex ones:

```
(define (product-of-doubles a b) (* (double a) (double b)))
```

The ability to define and then apply a function is called *functional abstraction*, and the facility for combining simpler functions into complex expressions is called *function composition.*

Function definition, application, and composition are the basic building blocks for functional programs: a program is a set of function definitions and a main expression, that is normally a single function application or a composition of function applications. An example program is

```
(define (double x) (+ x x))
(define (product-of-doubles a b) (* (double a) (double b)))
(product-of-doubles 3 4)
```

In the previous chapter it was observed that for the purposes of aiding the machine designer in understanding the intended behavior of programming language abstractions, and of aiding the programmer in correctly using the programming language, a useful way of defining semantics of a language is with a set of evaluation rules for the expressions of the language. Accordingly, the next thing to be considered are the rules for evaluating function definition and application.

To evaluate a function definition is to establish a correspondence between the function name and its value (which is some object that contains an implementation dependent representation for the variables and body of the the function). The correspondence, in case of global function definitions, is visible throughout the program, i.e. in the main expression and within all the functions, including the function being defined. The fact that a function is visible within itself means that it can call itself recursively.

A function application is evaluated by first associating the arguments with the corresponding variables in the function definition, and then evaluating the function body with that association (also called binding). For example, the application

```
(double 3)
```

is evaluated by first associating the argument `3` with the variable `x`, and then evaluating the body of function `double`, i.e. `(+ x x)`, with the established association. The result `6` is then the value of the expression `(double 3)`.

A function composition is an expression of nested function applications. It is evaluated by recursively evaluating all the nested applications.

A critical issue, highlighted by function composition, is the time at which an argument to a function is evaluated: before the function body itself is evaluated, or only once it is needed. The first approach is termed *eager evaluation*, and the second one *lazy evaluation*. The reason this issue is highlighted by function composition, is that an argument to a function, in a function composition, may be an expression in itself. If the function never requires the argument (is non-strict in that argument) the work done in evaluating the argument is wasted [20] [17]. This work may be particularly pointless when the argument is a large expression. Lazy evaluation avoids this problem because, unlike eager evaluation, lazy evaluation only evaluates arguments if they are needed. The other advantage of having lazy evaluation is that it supports streams (a list-like infinite data structure, further discussed in section 2.1.3 on page 19) [20] [1] [15].

To implement an eager evaluation strategy the evaluator performs the following: (1) evaluates the argument in a function application, (2) binds the result of this evaluation to the corresponding variable of the function, and (3) evaluates the function body with this binding. Implementing lazy evaluation is more involved because lazy evaluation requires that the argument be evaluated *only* the first time its value is needed. Any subsequent references to the argument will refer to its evaluated version rather than reevaluating it. Satisfying this requirement is inherently more expensive than satisfying the requirement for implementing the eager evaluation strategy. This is so, since, under lazy evaluation the evaluator needs to perform the following: (1) the expression representing the unevaluated argument in a function application is associated with the corresponding context (the association of expression and context

is variously referred to as a closure or a thunk), (2) this association is bound to the variable of the function, and (3) function body is evaluated with this binding, making sure that the variable that corresponds to the argument is evaluated only once its value is needed. Contrast this to the relatively simple steps in implementing the eager evaluation strategy.

Some programming languages make lazy evaluation the norm, while others make eager evaluation the norm and provide special primitives to support lazy evaluation when desired.[3]

Means for abstraction and composition are features exhibited by all powerful programming languages [1]. Facilities for abstraction permit naming and manipulating of complex computational objects as if they were a single object, while, the facilities for composition permit building of complex computational objects from simpler ones [1].

A pattern of operations and operands that constitutes a function body, built using function composition, is only one of many complex computational objects that exist in high-level languages; patterns of data objects that constitute data structures, or patterns of statements that constitute procedure bodies are other types of compound objects. In addition to means for abstraction and composition, a programming language has to provide a set of primitive computational objects that a programmer can abstract over and combine into useful computational structures. These primitives are a collection of basic operations or functions and basic data objects. Here, the word "basic" is a relative term, because, strictly speaking, the "basic" operations are abstractions of some even lower level objects in the underlying machine.

[3]Abelson and Sussman [1] show (on p264) how `delay` (a special form in Scheme) can be used to implement lazy evaluation. There they also show how `delay` can be implemented as an ordinary Scheme function, which is not sufficient for true laziness, because the delayed argument would be reevaluated every time its value is required. For supporting true laziness, the delayed argument should be evaluated only once, if at all. Hence, they present an implementation of `delay` as a special-purpose memoized procedure which implements true laziness. When called, this procedure wraps a procedure object with local state around the argument to be delayed, and returns that as the value of the delayed argument. Whenever the value of the argument is required, the procedure object representing it is invoked. This procedure object remembers whether it was already called, and if it was, it simply looks up a local state variable that contains the result of the first and only evaluation of the delayed argument rather than reevaluating it; if, however, this is the first time the procedure object is invoked, the delayed argument is evaluated and the value saved (*memoized*) for any subsequent references. It is interesting that a language, such as Scheme, which follows an eager evaluation strategy, can support lazy evaluation, albeit with use of procedures with side-effects rather than pure functions.

The next section contrasts the concept of state in functional and imperative programming, and the following two sections deal with the remainder of the functional system: the concepts of data abstraction and composition, and higher order functions.

2.1.2 State in Imperative versus State in Functional Programming

To maintain the property of functional abstraction that a user of a function only needs to know the function name and *what* it does rather than *how* it does it, the function cannot have side-effects. This is in contrast to procedural abstraction in imperative programming. In fact, imperative computing relies heavily on procedures with side-effects, evidenced by the fact that imperative computations normally have an implicit state, maintained as a collection of variables, that gets updated by the ubiquitous assignment operation to reflect the progress of computation. With presence of mutable state the order of operations becomes significant; consequently, the imperative system of abstractions includes the concept of sequencing of operations to ensure "...precise and deterministic control over the state. [17]"

Clearly then, the main difference between the functional and imperative systems of abstractions is that the imperative system has a history-sensitive [4] or mutable [1] state while the functional does not. Therefore, despite the fact that both functions and procedures share the property of being abstractions over some common pattern of operations, they differ in that procedures are executed for their effect, and their behavior may be affected by an implicit state (i.e. state that is not passed to the procedure in the arguments of the procedure), while functions are executed for their values, and the behavior of functions is wholly determined by their arguments.

An imperative program consists of a set of procedure definitions, with one of these being the main procedure, and calls on procedures to update some aspect of the computation state. An imperative computation is the process of first establishing procedure definitions, and second of invoking the defined procedures whose effect is to update the state of computation, normally held in a collection of variables. This is in contrast to a functional computation which, is the process of first establishing function definitions and second of applying the defined functions to their arguments. Expressions consisting of function applications are evaluated for their values rather

than their effect. The initial context of the functional computation does not get updated, rather it grows. On the other hand, the initial context of an imperative computation is repeatedly updated.

In functional programming, with the exception of the definition expression `(define ...)`, all the expressions and their compositions, are executed for their values rather than their effect. For example, all the primitive operations in fScheme (listed on page 54) are functions. A definition is an important exception: evaluation of a definition expression sets up the fixed part of the state or context of a functional computation. Most functional languages have, in addition to function definitions, expressions for defining data structures.

In the imperative system of abstractions, those abstractions that support a mutable state are the following: variable as a named location, assignment operation for setting a named location, and *sequencing* of operations. The concept of a variable as an abstraction for a memory location is central to supporting mutable state. Contrast this with a variable in functional programming, which is simply a placeholder or name for the argument in a function application. While this type of a variable is less powerful, it can support a weakened concept of state: a functional variable can be used to carry state around, however, this state is always explicit [17], being clearly exposed in the list of variables of the function, and it is not mutable (it cannot be updated, it can only be created, because there is no assignment statement in functional programming). This has the disadvantage of making state-oriented computations arising from functional programs inherently more memory hungry than those arising from imperative programs, because every time a state-change is made, a new instance of a variable has to be created to reflect the change. This requires allocation of new memory locations rather than reuse of the locations already allocated to the previous instance of the variable.

To effect complex changes in state, imperative computations normally repetitively execute a sequence of operations; this pattern is captured by the various iteration constructs available in imperative programming languages, such as `for`, `while`, and `repeat-until` constructs. In functional programming, repetition is accomplished through recursion, which can give rise to both linear and non-linear types of computational patterns. An example of a recursive function that generates a non-linear tree-like computational pattern is `fib` taken from p35 of [1]

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

In lacking the concept of a mutable state, the purely functional system of abstractions does not support building of computational objects with local state. This in turn means that the powerful and useful object-oriented view of systems is not used in purely functional programming [1]. The functional system of abstractions does, however, provide support for an alternative view of systems as streams of information flowing between functions [1]. Streams (discussed in the section on data abstraction) are data structures that can be used to represent "time history of successive states of an object" [1]. Even with streams, applications that require large-scale sharing of objects with changing local state, such as allocation of resources and implementation of cooperating and competing systems, are not expressible in purely functional languages [16]. For these applications, abstractions that incorporate concepts of mutable state are required, such as "resource managers" [2].

However, absence of mutable state and associated side-effecting primitives, makes functional programs referentially transparent,

> ...each expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of a program to share the expression [8].

An important consequence is that references to the value of such an expression are transparent to the history of the computation; i.e. in a given context, an expression always evaluates to the same value. Equational reasoning can be applied to such expressions [17]: an expression can be replaced by its equivalent (an expression that evaluates to the same value) without changing its own value or the value of an expression that references it. This is not generally true in the presence of the assignment operation because assignment makes possible destructive updates to the evaluation context of an expression. In turn, existence of destructive updates makes a value of an expression depend not only on the value of its subexpressions but also on the

history of updates to the evaluation context. This history-sensitivity destroys referential transparency of expressions and, consequently, not only makes reasoning about programs more difficult, but also complicates their design and debugging [17]. The crucial distinguishing feature of functions as they appear in functional programming and mathematics from functions in imperative programming is that they are referentially transparent [8]. Functions in imperative programming, while they do return values, are also permitted to change the context in which further computation will proceed.

The following program illustrates how referential transparency can be violated when assignment operation (`set!`)[4] is introduced into a language.

```
(define a 0)
(define (g) (set! a (+ a 1)))
(= (g) (g))
```

The expression (= (g) (g)) evaluates to `#f` (false)— expression (g) clearly does not "denote a single value which cannot be changed by evaluating" (g). In fact, (g) evaluates to a different value every time. The problem is not in that (g) references a, but that it destructively updates a, which is part of the evaluation context of (g). Function g is an example of a function as it appears in imperative programming, i.e. one that changes its own evaluation context.

Absence of side-effects also means that potential parallelism in functional programs is more easily detected than in imperative because, in lacking side-effects, functional computations also lack implicit constraints on ordering of operations or functions. In functional computations, all the ordering constraints are entirely due to data dependencies that are clearly exposed in functional expressions. For example, function `f1` depends on the output of function `f2` and `f3`, which in turn depend on variable `var`

```
(f1 (f2 var) (f3 var))
```

[4]Scheme syntax for the `set!` expression is

`(set!` ⟨name⟩ ⟨expression⟩)

and its semantics is that variable ⟨name⟩ is set to the value of ⟨expression⟩, which is also the value of the `set!` expression as a whole.

Observe that there can be no dependencies among functions `f1`, `f2`, and `f3` other than those clearly exposed in the expression. On the other hand, imperative languages make it easy to create implicit data dependencies between seemingly independent procedures, making it difficult for the programmer to control these data dependencies and for a parallel implementation to detect them. Unintended and unforeseen data dependencies are a common source of synchronization errors in parallel computations that arise from programs that contain side-effects. Accordingly, unlike imperative programming languages, functional programming languages free the programmer from worries associated with synchronization errors that arise whenever variables are updated in the wrong order [1].

2.1.3 Data Abstraction

Data abstraction is the analogue of functional abstraction in the data domain [1]—it permits abstracting over patterns of data in much the same way that functional abstraction permits abstracting over patterns of operations and operands. Data abstraction consists of the ability to name and manipulate compound data objects, or data structures. *Data composition*, much like function composition, is that facility which permits building of a data structure out of simpler data objects. The data composition facility of a language is realized through its data construction operations, which exist for each type of data structure. Each type of data structure also has data selection operations. In functional programming, selection, and composition are the only permitted operations on data structures—in purely functional languages there is no assignment operation. This section aims to illustrate the concepts of data abstraction and composition using three types of data structures: pairs, lists, and streams.

The ability to construct a compound data object, and then name it, makes it possible to treat a complex structure as a single object. Functional programming makes use of this by having functions regard data structures as single entities [15]: functions can both accept and return whole data structures. This capability

> "...is of fundamental importance to the usability of the purely functional language and requires a great deal of care in implementation if adequate efficiency is to be gained."[15]

It is clear how inefficient an implementation can become if it keeps allocating new memory locations for whole data structures as they are passed as arguments to functions or returned as their results. Allocation of new memory is not only inefficient but is also unnecessary, because there is no assignment statement in functional programming, and so, there is no danger of a function side-effecting a global, shared data structure that is passed to it as an argument.

The simplest data structure one can build is a pair. A pair is denoted by (*el1* . *el2*). For example, `rectangle` is a pair of two numbers representing the dimensions of the rectangle

```
(define rectangle (cons 3 4))
```

The expression `define` associates the name `rectangle` with a compound data object, that consists of a pair of numbers, `(3 . 4)`. The data constructor for pair is `cons`, and the data selectors are `car`, which selects the first element of the pair, and `cdr`[5] which selects the second. For example, the following expression evaluates to the first element of `rectangle`

```
(car rectangle)          ⟹ 3
```

Function `area` returns the area of a rectangle

```
(define (area rect) (* (car rect) (cdr rect)))
```

The function `area` is an example of a function that accepts a data structure, `rect`, and treats it as a single object. Functions can also return data structures. For example, consider function `scale` which accepts a rectangle and a scaling factor and returns the scaled version of the rectangle

```
(define (scale rect sfactor)
  (cons (* sfactor (car rect))
        (* sfactor (cdr rect))))
```

The following expression defines `rectangle2`, a scaled version of `rectangle`

[5]Reasons for using these cryptic names are historical: `car` and `cdr` were first used in the oldest versions of Lisp, and they came to be part of all its descendants, even the youngest ones such as Scheme. Data selectors `car` and `cdr` are acronyms for content of address part of register and content of decrement part of register, respectively. These acronyms are related to the machine organization of the IBM 704 on which Lisp was first implemented [24].

```
(define rectangle2 (scale rectangle 2))
```

Then, the expression `rectangle2` evaluates to

```
rectangle2                    ⟹ (6 . 8)
```

A common pattern of organizing data is in terms of an ordered sequence or a list. Here is a list of names

```
(Jane Bob Mary Kevin)
```

A list is a special case of a pair data structure, terminated by a special end-of-list marker, `'()`; the above syntax for a list is an easier to read version of

```
(Jane . (Bob . (Mary . (Kevin . '()))))
```

Data constructor and selector operations are the same as for pairs. The following expression constructs the above list

```
(cons Jane (cons Bob (cons Mary (cons Kevin '()))))

        ⟹ (Jane Bob Mary Kevin)
```

The expression for selecting the first element of the list is

```
(car '(Jane Bob Mary Kevin))⟹ Jane
```

while `cdr` returns the list minus its first element

```
(cdr '(Jane Bob Mary Kevin))⟹ (Bob Mary Kevin)
```

which is the complementary operation to `car`. The backquote character (`'`) indicates that what follows is a literal argument and should not be evaluated before the function is applied.[6] There is also a test for an empty list, `null?`, that returns `#t` (true) if the list is empty and `#f` (false) otherwise.

[6]The backquote character ' is a short form of `quote` expression. When applied to an argument, `quote` returns as its result the argument as is, rather than its value. This permits building of expressions that build and use other expressions; a powerful language feature. However, the problem with the `quote` expression is that its presence in a language violates the property that equals can be substituted for equals, and, therefore, makes reasoning about a language difficult (as indicated in [1] on p101).

```
(null? '())                    ⟹ #t
(null? '(Jane Bob Mary Kevin))
                               ⟹ #f
```

Another type of list-like data structure is a stream. It is different from a list in that it can model infinite sequences because, in constructing a stream, the evaluation of the tail is delayed until selection time. Contrast this with lists, where the head (car-part) and the tail (cdr-part) are both evaluated at construction time (when cons is applied) [1]. The motivation for delaying the evaluation of the tail of a stream is to make it possible for programs to operate on only that part of the stream that they need. Because of that evaluate-by-need property, a stream can be considered as a sort of lazy list. In what follows, streams are illustrated through examples adapted from Chapter 3 of [1].

One way to generate an infinite sequence of integers starting with k is as follows[7]

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ 1 n))))
(define integers (integers-starting-from k))
```

Another way of generating a stream is to do so implicitly rather than using a function. A stream of ones can be defined as

```
(define ones (cons-stream 1 ones))
```

The stream ones is a pair consisting of 1 and delayed expression ones. Evaluating the tail of ones returns a pair of 1 and the delayed expression ones, and so on. In a way, the definition of ones is recursive, much like a recursive function, and it only makes sense because the tail remains unevaluated until selected.

Another way of defining a sequence of integers starting with k adds the ones stream to the stream of integers being generated as that stream is being generated.

[7]Stream operations are analogous to list operations:

cons-stream corresponds to **cons**
head corresponds to **car**
tail corresponds to **cdr**
empty-stream? corresponds to **null?**

A null-stream is indicated by the symbol **the-empty-stream**.

```
(define (add-streams s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else
         (cons-stream (+ (head s1) (head s2))
                      (add-streams (tail s1) (tail s2))))))
(define integersk (cons-stream k (add-streams ones integersk)))
```

Streams lend themselves well to modeling signal flow type of systems [1]. For example, an integrator, whose block diagram is shown in figure 2.1 and whose equation is

$$S_i = C + \sum_{j=1}^{i} x_j \, dt$$

is defined as

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream dt integrand)
                              int)))
  int)
```

Function `add-streams` was defined above. Function `scale-stream` is

```
(define (scale-stream c stream)
  (map-stream (lambda (x) (* x c)) stream))
```

The function `map-stream`, and the `lambda` expression are defined in the next section.

Streams can be used to model state by representing successive states as successive elements of a stream [1]. For example to model the state of a bank account a function `stream-update` is defined which produces a stream of balances, given an initial balance, `balance`, and a stream of requests, `request-stream`, where the individual requests consist of a pair of request type (withdraw or deposit) and amount. A dataflow type of diagram for this function is given in figure 2.2.

```
(define (stream-update balance request-stream)
    (cons-stream balance
                 (stream-update
```

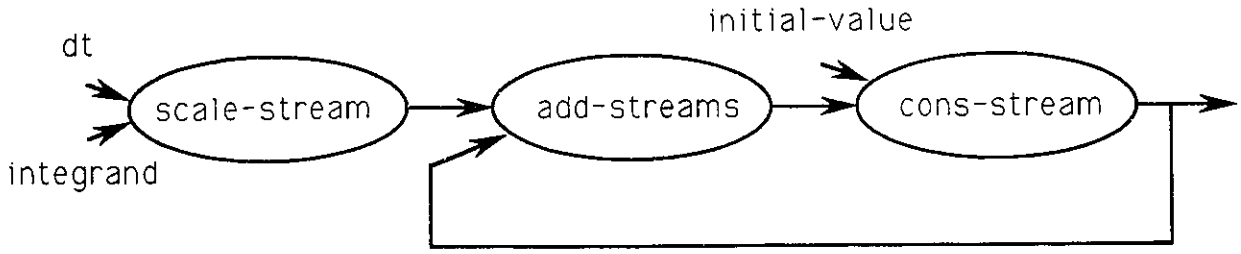


Figure 2.1: Integrator Signal Flow Diagram

```
(if (eq (car (head request-stream)) 'withdraw)
    (- balance (cdr (head request-stream)))
    (+ balance (cdr (head request-stream))))
(tail request-stream))))
```

2.1.4 Higher Order Functions

There is no need to segregate computational objects into functions and data. In fact, a very useful class of computational objects arises when functions are allowed to share the properties normally associated with data: properties that permit functions

> ...to be stored in data structures, passed as arguments, and returned as results [17]

Functions that have these properties are called higher order functions. An interesting effect of using higher order functions is that interpretation of an object as a function or a datum depends on the context in which the object is being used.

Consider two functions, `map` and `reduce`, commonly used to demonstrate the expressive advantage of higher order functions [15]. The function `map` accepts a list `x`

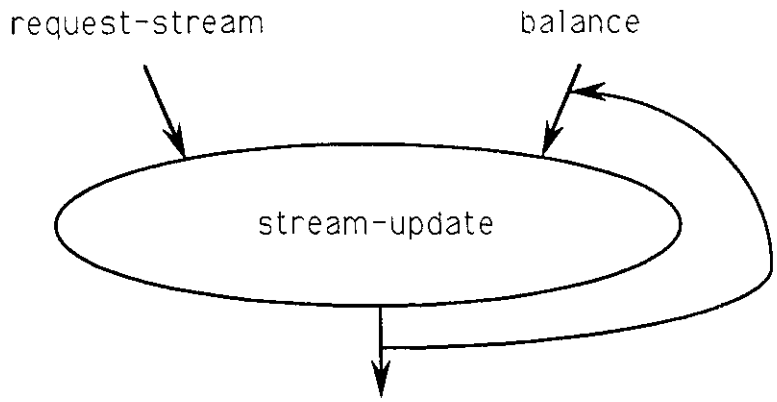


Figure 2.2: Bank Account Update

and a function `f` and returns a list which is the result of applying `f` to each element of `x`. Its definition is:

```
(define (map x f)
        (if (null? x)
            '()
            (cons (f (car x)) (map (cdr x) f))))
```

A similar function on streams can be defined

```
(define (map-stream x f)
        (if (empty-stream? x)
            the-empty-stream
            (cons-stream (f (head x)) (map-stream (tail x) f))))
```

One of the many useful functions that can be defined using `map` is incrementing by one each element of a list. The function `inclist` is simply

```
(define (inclist x)
        (map x (lambda (el) (add 1 el))))
```

Here `map` is passed an anonymous or nameless function defined using the lambda expression. The syntax of a lambda definition is

`(lambda` ⟨list of variables⟩ ⟨body⟩)

The reason for using an anonymous function is to avoid having to define the function globally when it is only needed locally in the function `inclist`. The evaluation rules for a function application with a function defined using the lambda expression are the same as those for a function defined using a define expression: first, the variables are bound to the actual arguments, and second, the body is evaluated with those bindings.

The function `reduce` is a bit more complex. It accepts a list `x`, a binary function `g`, and a constant `a` and returns

$$g(x_1,g(x_2,\ldots g(x_n,a))\ldots)$$

The definition of reduce is:

```
(define (reduce x g a)
        (if (eq x '())
            a
            (g (car x) (reduce (cdr x) g a))))
```

One of the many useful functions that can be defined using reduce is adding all the elements of a list. The function `addlist` is simply

```
(define (addlist x)
        (reduce x (lambda (x y) (add x y)) 0)
```

Higher order functions allow the programmer to express concisely many complex computations [15].

Allowing functions to have data-like properties is a powerful mechanism for building ever higher levels of abstraction; abstractions not only over patterns of fixed operations and variable operands, as is the case with first order functions, but also abstractions over patterns of variable operations and variable operands (or, put differently, abstractions over functional behavior). In [17] Hudak brings up a compelling argument by Hughes:

> ...higher order functions increase modularity by serving as a mechanism for gluing program fragments together. That gluing property comes not just from the ability to *compose* but also from the ability to abstract over functional behavior [17]

Because higher order functions are so important in functional programming, it is critical that they be implemented efficiently.

If functions are permitted to share properties of data why not have functions and data share the same form so that the facilities for manipulating data can be used to manipulate functions? This is exactly what has been done in Lisp, where, in fact, all expressions in the language have the same form: every expression is either a string or a list. As a result, Lisp has a very simple syntax. Because lists are not only a form of all expressions, but are also a basic data abstraction (ability to name, and, in particular, manipulate) Lisp programs can be manipulated by other Lisp programs[8]. Any programming language that has programs in a form of one or more data structures available in the language itself, has the power of manipulating its own programs. However, in most high level programming languages, the organization of procedures and functions is such that it would not be straightforward to provide data structures (in the language itself) that are an abstraction over different forms of procedures or functions. By the same token, it would be even less straightforward to build programs that manipulate these representations.

2.2 Functional Programming Languages

The purpose of this section is to briefly discuss functional programming languages by relating them to the system of abstractions of the previous section. With few exceptions (Backus' FP [4] is one), functional programming languages have at their core the above functional system of abstractions. Accordingly, the focus of this section is on how the core abstractions appear in these languages, and what additional abstractions, if any, these languages have. Most of the information about the various functional programming languages presented below can be found in more complete form in a survey article [17] written by Paul Hudak.

Lisp is the oldest programming language (in fact, together with Fortran, Lisp is the oldest high level programming language) that contains a significant functional

[8]It is important to note that in Lisp, this property of self-interpretation or self-understanding is limited only to the language level. For example, only expressions that define functions can be manipulated, and not functions themselves; that is, objects in the implementation that represent functions.

subset. Many dialects of Lisp have appeared since it was invented by McCarthy in the late fifties [24]; one of the most recent ones is Scheme [6], a language which provides one version of the basic system of functional abstractions. These are some of the significant characteristics of Lisp:

- Lisp follows an eager evaluation strategy for all user defined functions and most primitives. The conditional is the most notable exception to the eager evaluation rule: in a conditional expression the predicate is evaluated first, and then, depending on the value of the predicate, one of the consequents. This type of laziness with respect to the second and third arguments of a conditional is necessary for expressing recursion (otherwise a recursive call would diverge) [17].

- Lisp has a basic data abstraction mechanism consisting of the ability to name and build pair and list structures. These structures can be created dynamically which necessitates run-time reclamation of unused cons cells (a cons cell is a pair of pointers, one to the first element of the pair and the other to the second). The process of run-time storage reclamation is called garbage collection.

- Higher order functions are fully supported in Lisp, and are widely used in Lisp programs.

- Syntax of Lisp expressions is very simple: everything is an S-expression (symbolic expression). In the words of the report on Scheme [6]:

> Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs.

Even though Lisp does support the concept of mutable state and sequencing, and has primitives that have side-effects, it has a strong functional subset that makes it possible to program in the functional style. The Scheme dialect is openly advertised as supporting functional programming:

A wide variety of programming paradigms, including imperative, functional, and message passing[9] styles, find convenient expression in Scheme [6].

Since the appearance of Lisp numerous other programming languages that were either purely functional or contained a significant functional subset were created. Iswim, designed by Landin in mid sixties, was the first language of importance to functional programming to appear after Lisp [17]. Through Iswim Landin introduced several important ideas: support for equational reasoning[10] with rules for reasoning about expressions, a language design that emphasized generality, and the definition of the SECD[11] abstract machine for executing functional programs. Iswim also contained syntactic improvements over Lisp: use of infix notation, use of `let` and `where` clauses (incorporating mutually recursive definitions), and use of an off-side rule. The off-side rule relies on indentation rather than commas, or other separators, to denote scope of declarations and expressions [17]. Landin claimed that Iswim encouraged a declarative style of programming that was preferable to the sequential imperative style.

A little over a decade after Iswim, Backus presented FP in his Turing Award lecture [4] as a functional programming answer to the challenge of modern software development. A challenge, that according to him, could not be met by "word-at-a-time programming" promoted by imperative languages. FP consists of a set of combining forms (essentially higher-order functions [17]) used in building complex functions from simpler ones. The idea was to provide a small set of often used combining forms, such as `map` and `compose`, rather than an ability to define any type of function. According to Backus, the full generality and power of defining any type of function had the drawback of not making the programmer conscious of the most useful combining forms and the style of programming they encourage. This was one of the criticisms he leveled against lambda calculus based languages. The others were that it was difficult to efficiently implement the notion of reduction (or substitution) and handling of large data structures, and that the absence of abstraction for mutable state limited expressiveness. As a result he augmented the purely functional FP with the concept of state in what he called Applicative State Transition (AST) System.

[9] Also known as the object-oriented style

[10] Informally defined by Hudak as the "...ability to replace equals with equals." [17]

[11] SECD is an acronym for Stack, Environment, Control list, and Dump [15].

Most subsequent functional programming languages did not follow FP's philosophy of a few combining forms [17], and instead opted for the lambda calculus style of power and generality.

ML, which was developed at about the same time as FP, came about as a side-effect of work on a proof-generating system that was used to reason about recursive functions in the context of programming languages [17]. ML was the metalanguage or command language for this proof-generating system, but it was soon realized that ML was an interesting programming language in its own right. The most significant feature of ML was its type system, which is strongly[12] and statically typed, and which introduced type inference and polymorphism. Like Scheme and FP, ML included the concept of mutable state, while at the same time promoting the functional style of programming.

Types can be associated either with objects such as functions and data, or types can be associated with names denoting these objects and with expressions containing these names. In Scheme programs, types are associated with objects (values) rather than with names (variables) [6] or expressions containing these names. Unlike Scheme, ML has a type system where types are associated with variables and expressions and type inference is used to derive these types. Type of a variable is derived from the defining expression of the object the variable names, and type of an expression is derived from the types of its subexpressions. The advantage of a programming language with type inference is that the programmer need not specify types even though all variables and expressions in a program have a type. Another advantage of type inference is that it accomplishes both checking and derivation of types. In fact, the more familiar concept of type-checking can be seen as a special case of type inference [14]: if the type of an expression can be inferred in the context in which the expression occurs then, by definition, that expression is well typed, and if the type cannot be inferred, the expression is ill-typed.

In addition to type inference, ML also incorporated polymorphism in its type system, i.e. polymorphic functions and data structures. (A polymorphic function accepts arguments of arbitrary type, and a polymorphic data structure contains elements of

[12]Strongly typed refers to languages where type checking is "complete with no chance of mismatch" [22]

arbitrary type.) Polymorphism gives a statically typed language such as ML the ability to apply a function to arguments of different types. In absence of polymorhism, this advantage was reserved for dynamically typed languages such as Lisp where types are associated with values rather than with variables [6] (in particular, variables used as parameters in functions). It is perfectly acceptable to apply a Lisp function to arguments of different types (provided, of course, that applying the function to different argument types makes sense in the first place) because a functional variable in Lisp, by not having a type itself, can be bound to a value of any type. The advantage of polymorphism and type inference is that it gives a statically typed language such as ML the flexibility in typing reserved normally for a dynamically typed language, without having to incur the type-checking cost at run-time.

ML evolved into Standard ML (SML). One of the significant new features in SML, borrowed from Hope[13] was augmenting the basic function definition and application facilities with pattern matching in the arguments of the function. Contrast this with a Lisp function definition which simply lists the function variables; this is because a function application in Lisp only involves associating the function's variables with the corresponding arguments before the function body is executed. In languages that have pattern matching, before the function body is executed, the function's arguments may additionally need to be destructured to properly match the pattern of the function variables (for example, one function variable may be matched to the head and the other to the tail of an argument list to which the function is applied).

At about the time of FP and ML, David Turner began work on a series of three languages, the last of which is Miranda (notably, it is probably the only commercially available, purely functional programming language). The key influence of this series of languages was in promoting the use of lazy evaluation, higher order functions and recursion equations. Significantly, these languages were first to use list comprehensions as succinct ways of representing lists. List comprehensions are, essentially, a way of describing lists in such a way that its elements can be generated (if required). The obvious advantage in case of lists that can be represented as list comprehensions, is that functions do not have to be written to generate elements of such lists. List

[13]A programming language developed by Burstall, MacQueen, and Sanella at Edinburgh University in late seventies. It is strongly typed, supports polymorphism, supports pattern matching (even against data types defined by the user) and, unlike ML, requires that all functions be explicitly typed. It is strict, but it also directly supports lazy lists.

comprehensions, like pattern matching, are a syntactic feature that facilitates writing and reading of programs but adds no new power to a language [30]. Programs containing list comprehensions are easily transformed into equivalent programs that do not [30].

Many other functional programming languages appeared in the late seventies and early eighties, which led to a need for a standard functional programming language. The result was Haskell[14],

> ... a general-purpose, purely functional programming language exhibiting many of the recent innovations in functional (as well as other) programming language research, including higher order functions, lazy evaluation, user-defined data types, pattern matching, and list comprehensions. [17]

Functional programming languages have proven to have two outstanding advantages. They not only express a large class of useful applications in both numeric [2] and symbolic computing but, unlike programming languages with side effects, also permit relatively inexpensive detection and exploitation of parallelism [20].

Furthermore, unlike imperative programming languages, whose origins are in the von Neumann computer architecture [4], functional programming languages have as direct ancestors mathematical formalisms, the lambda calculus of Church and the recursion equations of Kleene [7]. As a result, a functional program can be manipulated as if it were a mathematical object [4]. Because of this some researchers in the field have noted that functional programming promises to be a sound basis for development of automatic programming techniques such as transformations of specifications to executable programs or transformation of inefficient programs to more efficient ones [7] [4].

Apart from research in functional programming as a source of new languages, there was also the research in dataflow computer architectures as another source of new languages. These languages are closely related to their respective underlying architectures, and as a result some of them have limitations related to implementation difficulties on the dataflow architectures [17]. For example, the language Val arising

[14]Named for Haskell Curry, a mathematician who made significant contributions to combinatory logic.

from work on the static dataflow architecture lacks higher order functions, notion of laziness, and even recursion [29]. On the other hand there is Id, related to the work on tagged-token dataflow model, which, while it does not support laziness, it does support non-strict functions [3], recursion and higher order functions [2] .

2.3 Literature on Functional Programming

Henderson provides an introduction to building programs with the functional system of abstractions in [15]. Abelson and Sussman exercise the functional system of abstractions in a context of programming in general in [1]. They expand the purely functional system of abstractions to include an abstraction for building computational objects with local state, arguing that because this supports the object-oriented view of systems, it is justified even though the new abstractions necessitate a change from a simple and elegant reduction evaluation model, which is sufficient for a purely functional system of abstractions, to a relatively complex environment evaluation model that supports computational objects with local state but has the disadvantage of being less theoretically tractable than the reduction model [1]. The new abstractions they introduce are the concept of a variable as a named location, the assignment operation for setting a named location, and sequencing of operations. In [17] Hudak presents an excellent overview of functional programming concepts and functional programming languages.

2.4 Conclusions

The purpose of this chapter was to describe the essential concepts of functional programming: first, independently of any particular programming language and second, as they appear in programming languages. These core concepts were identified as the functional programming system of abstractions—functional abstraction and functional composition, the corresponding concepts in the data domain, and the primitives.

The functional subset of Lisp (more specifically, the functional subset of its dialect Scheme) comes closest to the functional programming system of abstractions. The

more advanced languages discussed in the previous section tend to elaborate on the elements of that system as they appear in their most basic form in Lisp. These languages provide the following refinements and extensions:

- Lazy evaluation of functions in general, and data constructors in particular.
- Alternative ways of defining functions using pattern matching in functional arguments and using recursive equations. Together they facilitate equational reasoning in building functional programs [17].
- Type system and the associated concepts of type inference and polymorphism. Unlike Scheme where types are associated with values rather than expressions [6], languages such as ML, Miranda and Haskell have a type system where types are associated with variables. These languages use type inference to derive and check types of expressions. They also support polymorphism.
- Data abstraction mechanisms such as user-defined data types (concrete and abstract), and list comprehensions.
- Formal semantics as an integral part of language definition. Because of their roots in Church's lambda calculus [5] functional programming languages generally have formal definitions. In fact, denotational semantics (an approach to formal definition) and functional programming are closely related. Generally, researchers in functional programming stress formal semantics [17]. Denotational semantics is used to define non-functional languages as well; for example, report on Scheme [6] gives "a formal denotational semantics for primitive expression of Scheme and selected built-in procedures".
- Syntactic "improvements" such as use of infix rather than parenthesized prefix notation, and use of the off-side rule.

Chapter 3

Functional Programming Evaluators

This chapter briefly discusses existing functional programming evaluators. These evaluators are based on a variety of evaluation models and they run on machines ranging in organization from conventional sequential architectures to novel reduction and dataflow architectures.

3.1 Requirements for Implementing Functional Programming

An executing functional program is an expression which, to be fully evaluated, needs a related context that contains the values for all the variables and functions referenced by name in the expression. The ensemble of expression and context constitutes the state of computation. The context of a computation is created as the result of the use of function abstraction and function application in the expression being evaluated.

Discounting for the moment internal definitions of functions (functions defined within functions), the source of the static part of the context is the set of top level function definitions. Generally, *static context* is that part of the context that comes into existence prior to evaluation of the expression.

Function application involves establishing a mapping between variables used in

a function body[1] and their corresponding values, which are the arguments in the function application, and then evaluating the function body with above mentioned mappings as part of the context. Because this part of the context arises at execution time it is referred to as *dynamic context.*

The above, seemingly simple, evaluation rule for function application can be implemented in surprisingly different ways. Next section shows how the differences between evaluation models most clearly exhibit themselves in the way they handle the dynamic context arising from function applications.

3.2 Evaluation Models and Computer Architectures

The environment implementation approach is the oldest, the best understood and the most widespread. In its earliest form it was defined by McCarthy as a basis for an early Lisp implementation running on an IBM 709 [24]. In the environment approach, the *environment* structure is used to accumulate bindings (mappings) of variables to their values; bindings resulting from function applications. As the program expression is evaluated the *environment* is written to establish new bindings and read for the value of a variable currently required. Execution stops when the program expression is completely evaluated, i.e. when there are no more function applications left. Because of its effectiveness on conventional machines, the environment implementation approach is a suitable yard-stick against which to judge any new implementation approach. Also, being well understood makes it a convenient reference for presenting new implementations. Of all the existing approaches the environment approach is the one closest to the von Neumann hardware model. Therefore, it is not surprising that it is the most efficient implementation approach on conventional architectures.

String reduction, graph reduction and dataflow are new approaches to the implementation of functional programming languages. They make no assumptions about the underlying hardware model. Rather they assume a top-down, language oriented design of the hardware architecture. In all of them, as in the environment approach, the executing functional program is an expression. In the case of the string and graph

[1] FP is an exception in that it does not have variables

reduction approaches the executing functional program is represented as a string or a graph being reduced rather than evaluated with the help of an auxiliary environment-like data structure. Values of variables are not bound to variable names and placed in the environment as a function is applied; instead, the values, in case of string reduction, or pointers to these values, in case of graph reduction, are written directly into a new copy of the function being applied. Applying that function involves reducing the new function body. Execution terminates when the overall program expression is in an irreducible form. Because of its inefficiency, string reduction has not been a widely researched approach. An exception was the group that worked on development of a parallel machine for direct execution of the FFP (Formal Functional Programming) language [23] [4]. Graph reduction, first presented by Wadsworth in 1971 [20], is proving to be a more serious contender to the established environment implementation approach than the string reduction approach. As a result of the promise of graph reduction, numerous implementations of it have appeared in recent years. Some run on conventional sequential machines while others run on specifically graph reduction parallel machines.

In the dataflow approach the functional expression is represented as a dataflow graph. A result of a function application is not obtained by copying out a whole function body and then reducing this function body with the arguments; instead the result is obtained by passing the arguments to a dataflow graph of the function body and then waiting for the result to flow out. Unlike a function graph in graph reduction, which gets *literally* reduced, a function graph in dataflow remains structurally the same during a function application. However, dataflow has to accommodate in some way the dynamic aspects of a computation. In dataflow the static part is a dataflow graph consisting of the dataflow operators and their interconnections and the dynamic part is represented by data "flowing" along the interconnections between the operators. Another way of looking at data flowing through a dataflow graph is as a context being distributed over that graph. This context has to be dynamically created if multiple invocations of a function are permitted (which is required for support of recursion). For example in the MIT Tagged-Token Dataflow Architecture, each time a function is invoked a new context is created and passed to the function along with the arguments [2]. All the data values "flowing" through the graph carry along a unique name, created at run time, identifying their context. A similar approach is

used in the Manchester Prototype Dataflow Computer [11].

Arvind and Nikhil noted in [2] that "dataflow and graph-reduction are just two different ways of implementing the same abstract concept of reduction". It may be argued that the dataflow approach is a more efficient way of implementing a reduction semantics than the graph reduction approach because the dataflow approach completely avoids the unnecessary copying present in graph reduction. In the dataflow approach only those things that are inherently dynamic, i.e. associations between arguments and variables, are created at run-time.

For creating sufficient parallelism a dataflow implementation may involve copying of dataflow graphs over different nodes.

In contrast to the environment approach where the context is centralized in the environment structure, the dataflow and graph reduction approaches both distribute the context over a function graph. The centralized approach forces serialization of accesses while the distributed approach permits concurrent accesses. In fact this is why the dataflow and graph reduction approaches are both a more natural framework for parallel implementations of functional programming than the environment approach.

> In essence the environment becomes distributed into the program structure itself, thus simplifying execution on a distributed architecture. [18]

From above it can be seen that, while the core requirements for implementation of functional programming are few, the ways in which these requirements can be met are many. Comparing resulting implementations is both a fascinating and necessary exercise. Fascinating because of the variety of implementations all satisfying the same small set of core requirements, and necessary because of the importance of functional programming.

This thesis analyses and compares the environment and graph reduction implementation approaches. These are chosen because they are representative of two opposing design philosophies: execution efficiency vs semantic proximity to a high level language, bottom-up vs top-down design, machine oriented vs language oriented, and so on. A comparison of such diverse approaches presents both opportunities and problems. Problems because it is difficult to find a common measure or dimension along which to compare and opportunities because the comparison may provide guidelines for improving both of the approaches.

Chapter 4

The fScheme Programming Language

The purpose of this chapter is (1) to present fScheme, the programming language for which the environment and reduction evaluators are built, and (2) to present the environment evaluation rules for fScheme.

4.1 fScheme (Functional Scheme)

fScheme is a small and purely functional subset of Scheme. fScheme is small and purely functional because this thesis focuses on implementation of only the basic elements of functional programming. It, however, inherits from Scheme static scoping and the requirement that tail-recursive[1] functions evaluate in constant space, as if they were iterative functions.

> Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme [6].[2]

[1] Static scoping and tail-recursion in fScheme are discussed on pages 38 and 42 respectively.
[2] Throughout this section, the report on Scheme [6] will be extensively quoted.

The core of the fScheme system of abstractions consists of the twin concepts of function and data. The key elements of the system being functional and data abstraction, and function and data composition (as presented in Chapter 2). Of course, in addition to means of abstraction and composition, fScheme has a set of primitive computational objects, over which a programmer can abstract, and compose into useful computational structures.

The form of fScheme is identical to that of Scheme:

> Scheme, like most dialects of Lisp, employs a fully parenthesized prefix notation for programs and (other) data; the grammar of Scheme generates a sublanguage of the language used for data. An important consequence of this simple, uniform representation is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs [6].

It is this last property that permits us to use Scheme not only as a starting point for generating a functional programming language, but also as the language in which the environment and reduction machines are programmed. These machines are simulated as interpreters that process fScheme expressions, i.e. they are Scheme programs that process other Scheme programs.

4.1.1 Variables, Locations, Name-spaces, and Regions

In fScheme, variables are names for locations that are capable of storing any object. A variable that names a location is said to be *bound* to the location.

> The set of all visible bindings in effect at some point in a program is known as the *environment* [name-space][3] in effect at that point. The value stored in the location to which a variable is bound is called the variable's value.
>
> ...
>
> Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use. Whenever this report speaks of storage being allocated for a variable or object, what is meant

[3] In this thesis the term *name-space* is used instead of the term *environment*.

> is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them [6].

Objects are never explicitly destroyed in fScheme. As a result, all the objects created during a course of a computation have unlimited extent. To avoid running out of storage for its computations, fScheme permits the evaluator to reclaim the locations containing those objects for which it "...can prove that the object cannot possibly matter to any future computation [6]".

Note that an fScheme variable is somewhat different from a functional programming variable. In fScheme, a variable is a name for a *location* that contains a value, while in functional programming, a variable is a name for a value. fScheme inherits its view of a variable from Scheme, which associates variables with locations because that is a part of supporting the concept of mutable state. As far as behavior of functional programs is concerned, however, the two views of a variable are equivalent.

A name-space is a collection of name-location bindings. The name-space, at any point, in an fScheme computation is organized as tree-like hierarchy, with the global name-space being at the root of that hierarchy and initially containing the name-location bindings that define the names and values of all the primitive functions of fScheme. Whenever a variable is referenced, the evaluator starts looking for the binding in the local name-space, and if it cannot find the binding there, proceeds up the hierarchy until it finds the binding.

Whenever a function is applied, a new leaf, containing the local name-location bindings, is added to the tree (as detailed in the evaluation rule for a function application, page 40).

> Certain expression types are used to create new locations and to bind variables to those locations. The most fundamental of these *binding constructs* is the lambda expression ... [6].

Others are the let and letrec expressions (block structure constructs) described on pages 46 and 47, respectively. The hierarchical tree-like structure of the name-space arises from nesting these binding constructs.

fScheme follows the same scoping rules as Scheme.

> ...Scheme is a statically scoped language with block structure. To each place where a variable is bound in a program there corresponds a *region* of the program text within which the binding is effective. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every reference to or assignment of a variable refers to the binding of the variable that established the innermost of the regions containing the use. If there is no binding of the variable whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any; if there is no binding for the identifier, it is said to be *unbound* [6].

Note how static scoping is achieved (1) by organizing the name-space as a tree-like hierarchy of name-subspaces, and (2) by applying functions in the (extended) name-space of their definitions (as described in the evaluation rule on page 40), rather than the (extended) name-space of their applications (as is the case in a dynamic scoping).

Memory Model

The static scoping rules of fScheme, and the types of objects available in fScheme (page 50) are what defines the fScheme memory model. The intent of a memory model in any programming language is to impose a structure on the memory of the underlying machine, i.e. to provide a programmer with a useful abstraction of the low level memory model available in hardware.

4.1.2 Functional Abstraction

Functional abstraction is the ability to define and then apply a function. In fScheme, a function is defined as follows[4]

```
(define ⟨name⟩
  (lambda (⟨variables⟩) ⟨body⟩))
```

[4]As remarked in Chapter 2 ⟨name⟩, ⟨variables⟩, and ⟨body⟩ are part of the language used for describing the syntax of Scheme, not actual Scheme expressions. The report on Scheme [6] refers to them as syntactic variables, i.e. variables which denote valid Scheme expressions.

The lambda expression is what gets evaluated to a function, while it is the define expression that names that function. In general, the define expression is used to associate a name with any location (locations contain values).

As part of static scoping, a function value in fScheme is required to remember its definition name-space, because that name-space is used when the function is applied.

The form and the evaluation rule of a function application are, respectively

($\langle$operator$\rangle$ $\langle$operand$_1\rangle$ ...)

1. Evaluate the operator and the operand expressions in the current name-space. The operator expression has to evaluate to a function, to which the evaluated operand expressions are passed as arguments.
2. Extend the function definition name-space with the bindings of variables to fresh locations.
3. Store the argument values in the corresponding locations.
4. Evaluate the expression in the body of the function in the extended name-space.
5. The result of this evaluation is the value of the function application.

The ability to build complex functions from simpler ones is called *function composition.* In fScheme, it is expressed by nesting function applications, as in

```
(f1 (f2 3))
```

The evaluation rule for a function composition is

- Recursively evaluate all the nested applications.

The above evaluation rules are illustrated in figure 4.1. The figure shows the changes in the name-space as the evaluator applies the rules in evaluating the expressions of the following program

```
(define (f1 x) body1)
(define (f2 y) body2)
(f1 (f2 3))                    ⟹ value
```

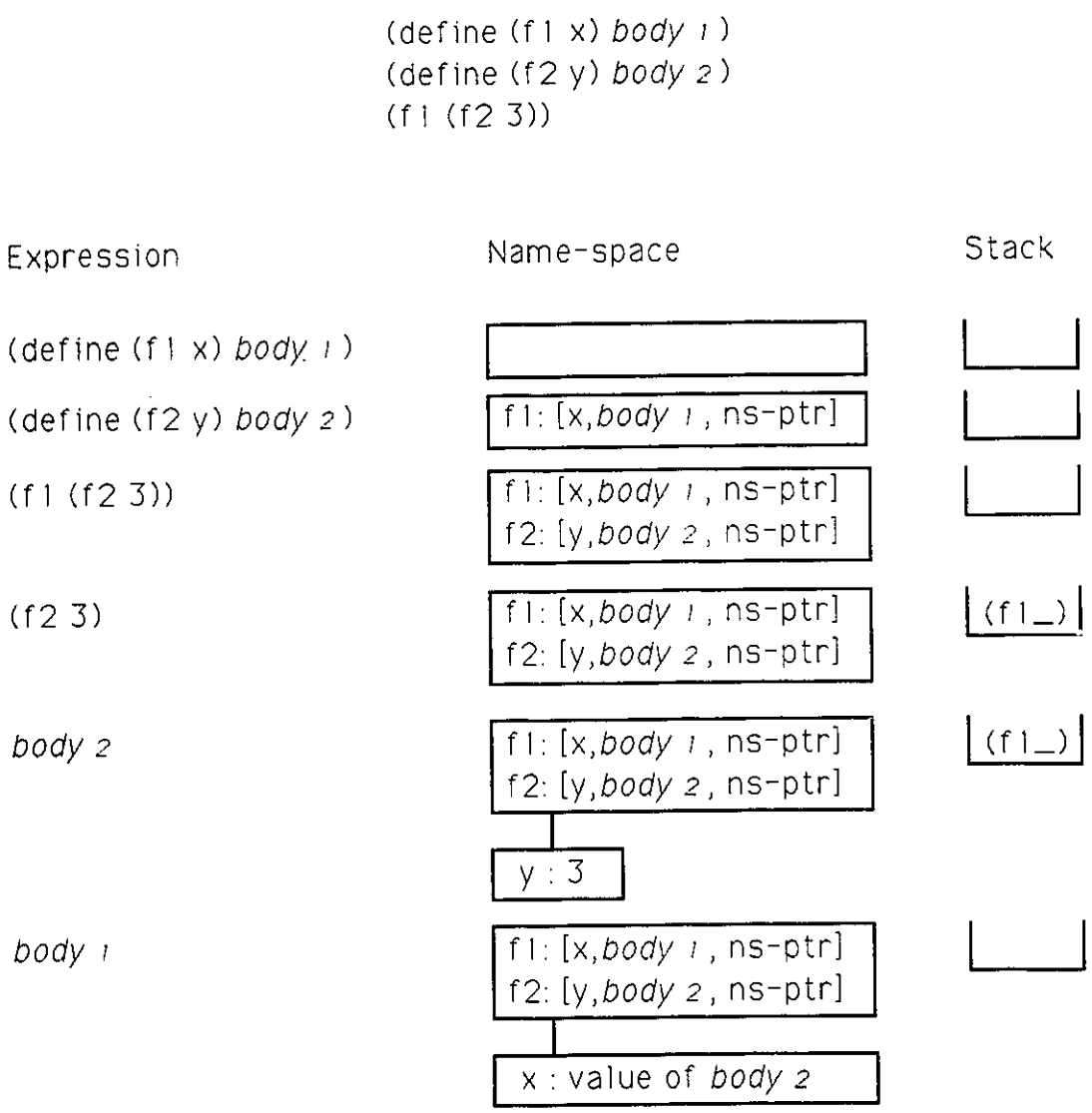


Figure 4.1: Example 1 Evaluation

fScheme requires that the evaluator recognize a tail-recursive function as one that generates a linear iterative process, and evaluate it in constant space. Even though a tail-recursive function refers to itself in its own body, the process it generates has a state that is fully described by a fixed number of variables, i.e. the state does not require a stack to maintain the deferred function calls, as is the case for a recursive process [1]. An example of a tail-recursive function is `fact-iter`[5], which computes the factorial of `counter`

```
(define (fact-iter product counter)
(if (<= counter 0)
    product
    (fact-iter (* counter product)
       (- counter 1))))
```

For example, to compute factorial of 3, `fact-iter` is called with `(fact-iter 1 3)`, which generates the following iterative process

```
(fact-iter 1 3)
(fact-iter 3 2)
(fact-iter 6 1)
(fact-iter 6 0)
6
```

Contrast `fact-iter` with an equivalent function `fact`

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

which generates the following recursive process when applied to 3

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

[5]This example is adapted from page 31 of *Structure and Interpretation of Computer Programs* by Abelson and Sussman [1].

To fully describe the computation state of `fact` the evaluator needs a stack, to maintain the deferred function applications, as well a local name-space which holds the binding of n for each application of `fact`. Since each of the deferred application has an associated name-space, there are as many instances of the binding of n as there are deferred applications. Consequently, the amount storage for maintaining the state of a recursive process grows with the depth of the recursion, i.e. it is not constant. This is in contrast to `fact-iter` which has a state that is fully described by a *fixed* number of variables (namely, `product` and `counter`).

Figures 4.2 and 4.3 illustrate the difference between evaluating application of a tail-recursive function, `fact-iter 1 3`, and application of a recursive function, `fact 3`.

4.1.3 Expressions

> A Scheme expression is a construct that returns a value, such as a variable reference, literal, procedure call [function application in fScheme], or conditional [6].

With the exception of the define expression, all the expressions in fScheme behave *like* functions in that they return some value, although they may be evaluated differently from functions. The fact that they behave like functions, means that they can be combined into larger expressions with the same "parenthesized prefix notation" used above to denote function composition. This leads to a certain economy of expression, but also, some may argue, hides significant differences among expressions types.

fScheme only has a few types of expressions: variable reference, literal expression, function application, lambda expression, conditional expression, let expression, letrec expression, and define expression. The form and evaluation rules of each are

- **variable reference**

  ⟨variable⟩

  Evaluates to the value stored in the location to which the variable is bound.

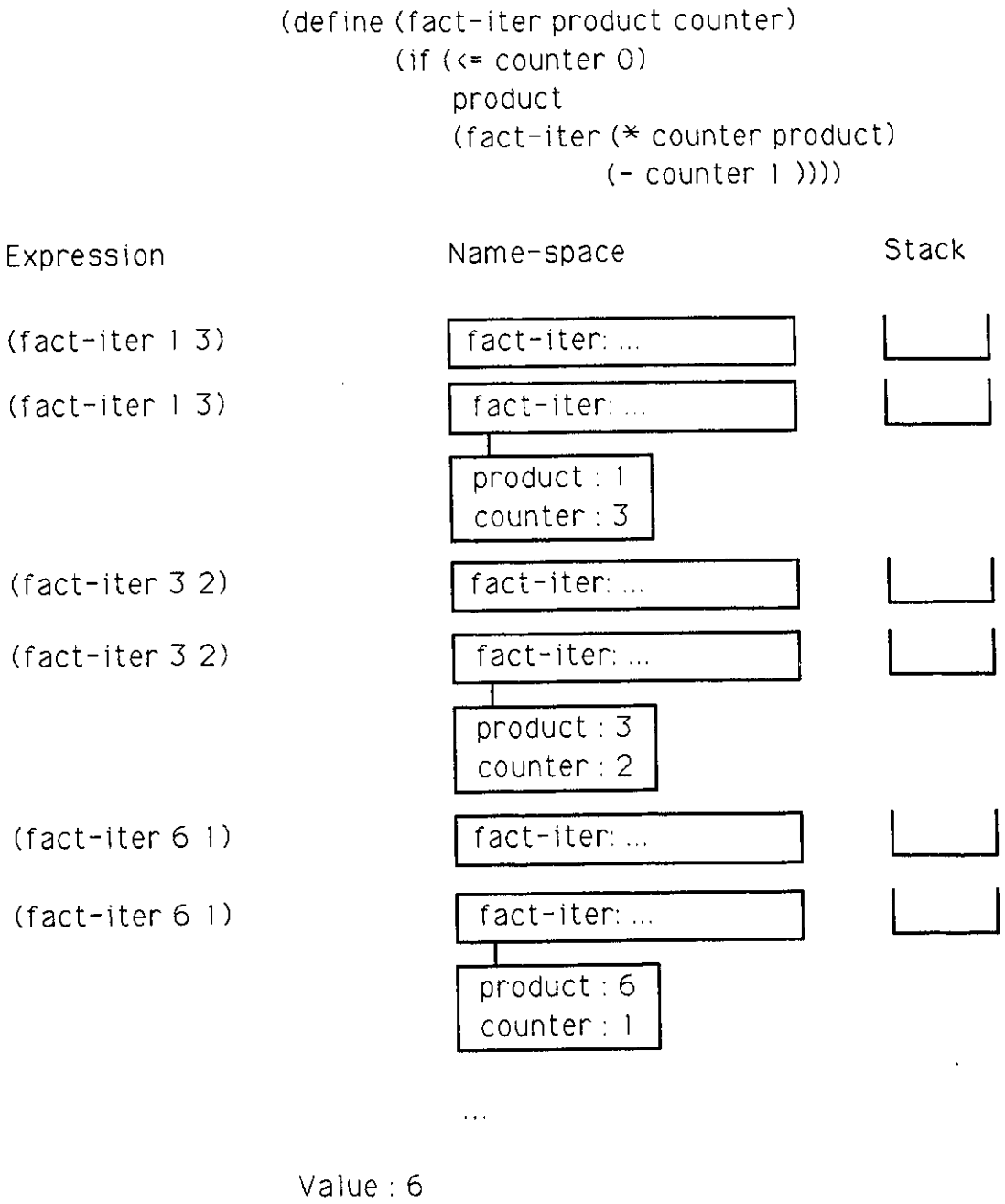


Figure 4.2: Example 2 Evaluation

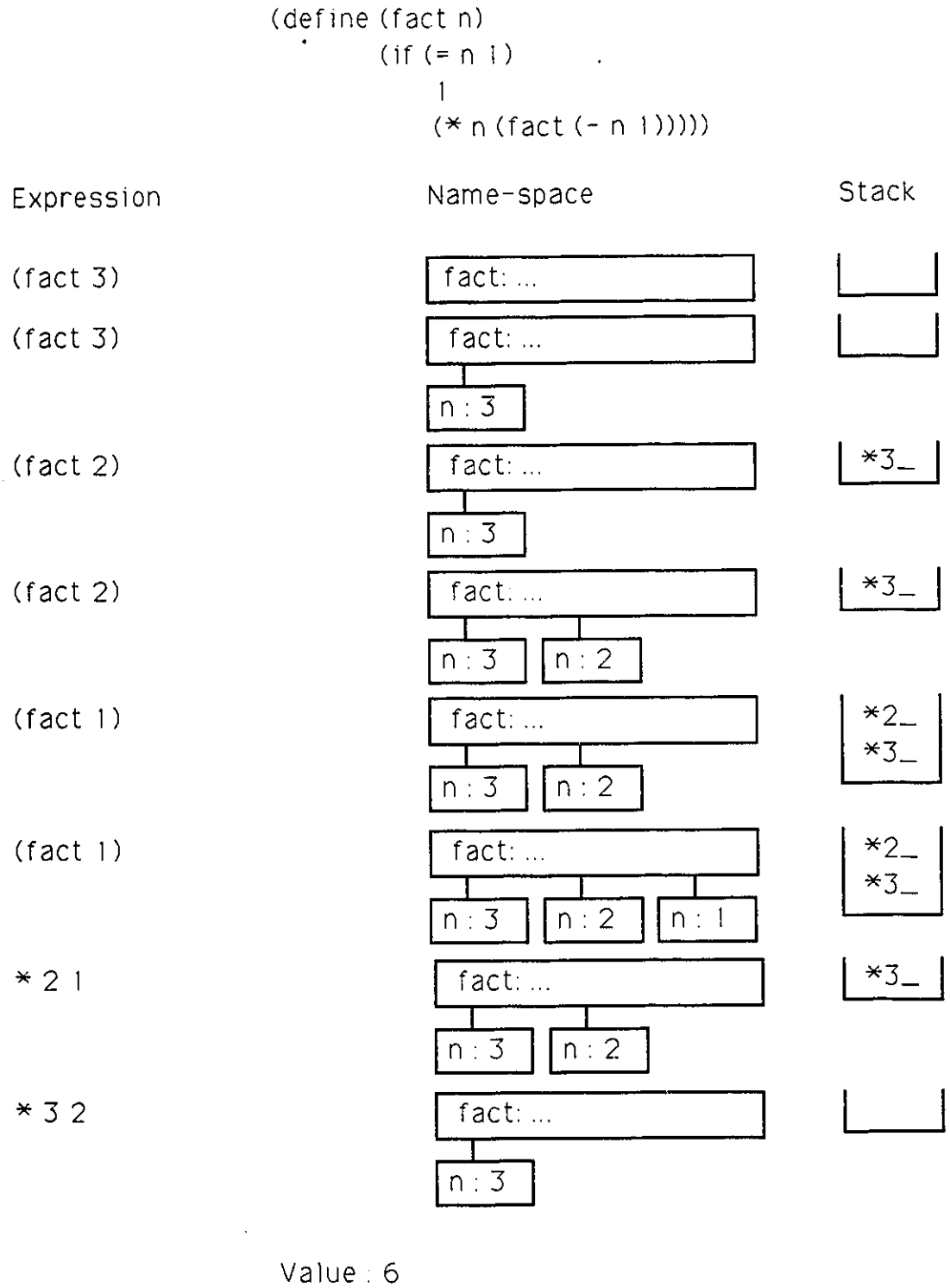


Figure 4.3: Example 3 Evaluation

- **literal expression**

  `(quote` ⟨datum⟩`)` or `'`⟨datum⟩[6] or ⟨constant⟩

  Evaluates to ⟨datum⟩ or, in case of a ⟨constant⟩, to itself.

- **function application**

  (⟨operator⟩ ⟨operand$_1$⟩ ...)

  Evaluation rule is on page 40.

- **lambda expression**

  `(lambda` ⟨formals⟩ ⟨body⟩`)`

  Evaluates to a function, with the name-space in which the lambda expression was evaluated becoming the definition name-space part of the function.

- **conditional expression**

  `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`

  The if expression is the only way to accomplish conditional evaluation in fScheme. The evaluation rule is

  1. Evaluate the ⟨test⟩ part.
  2. If the result is a true value (booleans are defined on page 50) then evaluate the ⟨consequent⟩ part, otherwise evaluate the ⟨alternate⟩ part.
  3. The result from step (2) is the value of the whole if expression.

- **let expression**

  `(let` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ have the form

  (⟨variable$_1$⟩ ⟨init$_1$⟩) ...

  and where each ⟨init⟩ is an expression used to initialize the corresponding ⟨variable⟩, and ⟨body⟩ is the main expression. A ⟨variable⟩ can appear only once in the list of variables being bound.

  Evaluation rule for a let expression is similar to that of the function application expression (page 40), with the ⟨init⟩s being the arguments, the ⟨variable⟩s the function variables, and the ⟨body⟩ the function body:

[6] A quote can be seen as an annotation for controlling the normal eager evaluation, rather than an expression with its own evaluation rule.

1. Evaluate the ⟨init⟩s in the current name-space.
2. Extend the current name-space with the bindings of ⟨variable⟩s to fresh locations.
3. Store the results of evaluating the ⟨init⟩s in the corresponding locations.
4. Evaluate the ⟨body⟩ in the extended name-space.
5. The result of this evaluation is the value of the let expression.

Bindings of the ⟨variable⟩s have the ⟨body⟩ as their region. Like function applications, let expressions can be nested

```
(let ((a 1) (b 2))
     (let ((a 3) (c (+ a 1)))
          (+ a c)))          ⟹ 5
```

- **letrec expression**

  `(letrec` ⟨bindings⟩ ⟨body⟩`)` `where` ⟨bindings⟩ have the same form as for the let expression

  A letrec expression is different from the let expression in that it permits mutually recursive functions to be defined. The evaluation rule for a letrec expression is

  1. Extend the current name-space with the bindings of ⟨variable⟩s to fresh locations holding undefined values.
  2. Evaluate the ⟨init⟩s in the *extended* name-space.
  3. Store the results of evaluating the ⟨init⟩s in the corresponding locations.
  4. Evaluate the ⟨body⟩ in the extended name-space.
  5. The result of this evaluation is the value of the let expression.

  It must be possible to evaluate all the ⟨init⟩s without referring to the value of any of the ⟨variable⟩s. This does not present any problem, because ⟨init⟩s are usually lambda expressions. Bindings of the ⟨variable⟩s have the *entire* letrec expression as their region, which permits definition of mutually recursive functions. In a let expression this is not possible, because the region of the

bindings of ⟨variable⟩s is only the ⟨body⟩ of the let expression, and not the whole expression itself.

An example of a letrec expression is [7]

```
(letrec ((even?
          (lambda (n)
            (if (= 0 n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (= 0 n)
                #f
                (even? (- n 1))))))
  (even? 88))
                              ⟹  #t
```

Figure 4.4 shows the evaluation process for the above letrec expression.

- **define expression**

  (define ⟨name⟩ ⟨expression⟩)

  The define expression is the only expression that does not return a value. The evaluation rule is

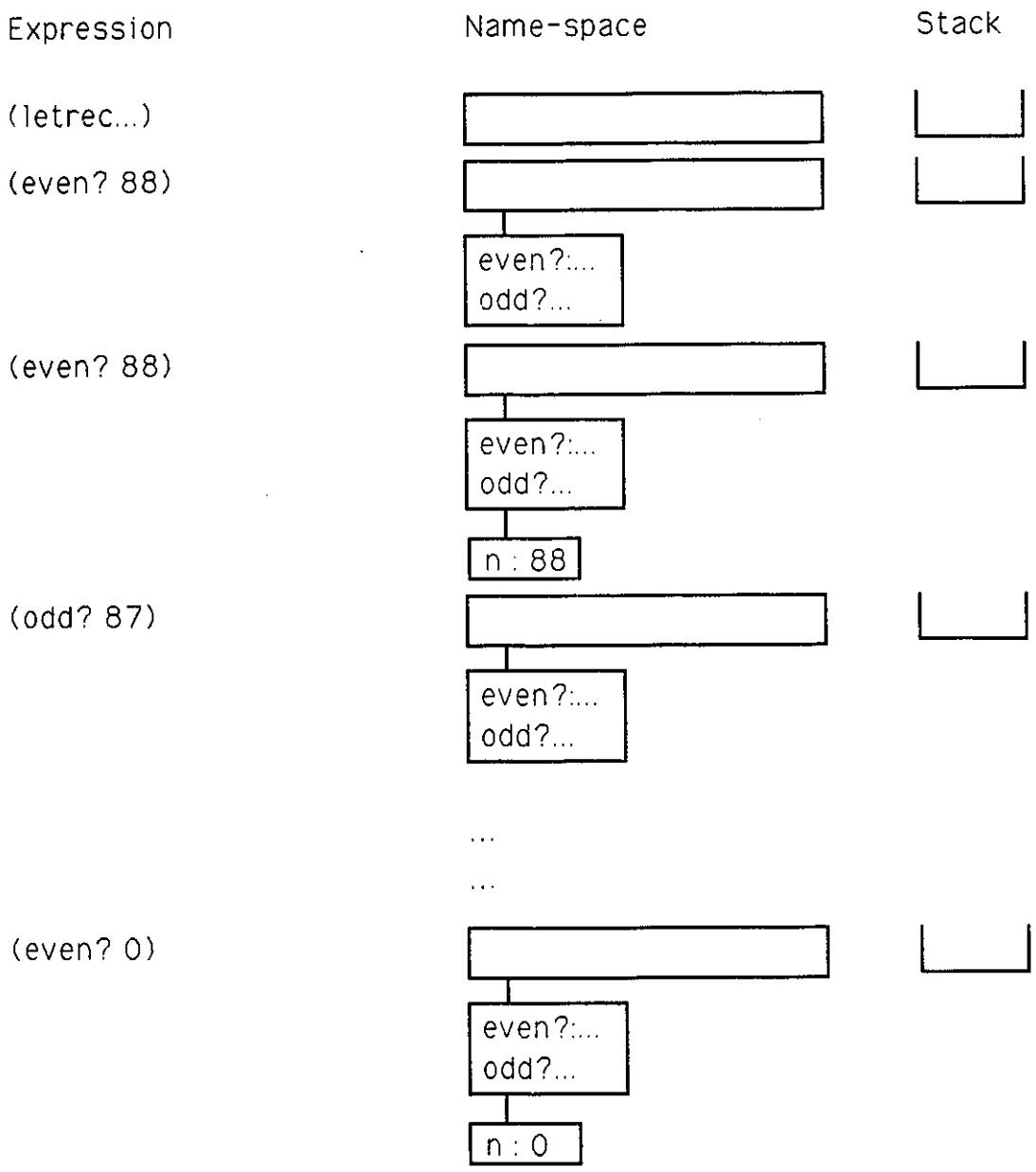
  1. In the global name-space, introduce a new binding of ⟨name⟩ and a fresh location.
  2. Evaluate the expression ⟨expression⟩ and store the value in the location.

  Definitions can only occur at the top level of a program. It is not permitted to use them inside a function or a let or letrec expression.

Only top-level definitions are supported in fScheme, i.e. there are no internal definitions. In particular, there are no functions defined as internal functions to some other functions. However, this is not a limitation because programs containing

[7] This example is taken from page 10 of report on Scheme [6].

```
(letrec ((even? (lambda (n)
                  (if (= 0 n) #t (odd? (- n 1)))))
         (odd? (lambda (n)
                  (if (= 0 n) #f (even? (- n 1))))))
  (even? 88))
```



Value : #t

Figure 4.4: Example 4 Evaluation

functions with internal definitions can be converted to those containing only functions defined at the top level. The conversion is the process of lambda lifting, defined by Johnsson as:

> ...a technique for transforming a functional program with local function definitions, possibly with free variables in the function definitions, into a program consisting only of global function (combinator) definitions which will be used as rewrite rules. [19]

4.1.4 Types of fScheme Objects

fScheme objects have one of the following types: *symbol*, *boolean*, *pair*, *number* (integer), and *function*. [8] For each object type there is a predicate that tests for the type (listed on page 54).

Symbol

Symbols are objects with only one attribute: name. As a result, in fScheme (as in Scheme) symbols are mainly used as identifiers. However, they can also be used "the way enumerated values are used in Pascal." [6] Consider an example of both uses of the symbol type,

```
(define Names '(John Mary Sue))
```

where `Names` is the identifier for an enumeration (a list) of names `(John Mary Sue)`.

Boolean

There are only two *boolean* objects: `#t` and `#f`.

An if expression accepts other than boolean objects as arguments. In an if expression, a *true* value is any value other than an empty list (denoted by `()` or `nil`) or the boolean `#f`; and, a false value is the boolean `#f` or an empty list.

The set of primitive predicates (functions that return boolean values) includes the type predicates, the equivalence predicate, and the null-list predicate. These are listed on page 54.

[8] Scheme has the type *procedure* in place of *function*, additional data types, and a more complex number system [6].

The equivalence predicate eq?,

(eq? obj_1 obj_2)

tests for equivalence of its two arguments. Eq? returns #t if the arguments are equivalent, and #f otherwise.

The only primitive boolean function in fScheme is not, which returns #t when applied to a false value, and returns #f otherwise.

Pair

The *pair* object type, denoted by (*el1* . *el2*), is the only data structure of fScheme, and the sole support it provides for data abstraction (Scheme, additionally, has the string and vector data structures). The fScheme primitives for constructing, and accessing pairs are cons, car, and cdr (all of which were defined in Chapter 2, on page 17).

A subtype of the *pair* object type is the *list* type, denoted by (*el1* *el2* *el3* ...). In addition to above primitives, the list type has an associated test for an empty list, null?. When applied to an object, null? returns #t if the object is an empty list, and returns #f otherwise. The empty list is denoted as () or with the symbol nil.

In fScheme, the empty list serves an important purpose: (1) it is used in terminating lists (as described in Chapter 2, on page 18), and (2) together with the boolean #f, it is treated as a false value by the if expression.

Number

For reasons of simplicity, fScheme only supports integer arithmetic[9]: all the numbers are integers and the operations on them are +, -, *, quotient, remainder[10], and the relational operators =, < and <=.

[9]Scheme, however, has a much more complex number system, with a full complement of numeric types (complex, real, rational, and integer) and associated operations. [6]

[10]When applied to integers n_1 and n_2, which satisfy $n_1 = n_2 n_3 + n_4$, the behavior of quotient and remainder operations is

(quotient n_1 n_2) $\Longrightarrow$ n_3
(remainder n_1 n_2) $\Longrightarrow$ n_4

Numbers evaluate to themselves, i.e. they are self-evaluating, so that they do not need to be quoted.

Function

An object of type *function* is like any other object in fScheme. This view is inherited unchanged from Scheme, whose procedures (the Scheme equivalent of a fScheme functions)

> ...are objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. [6].

Functions have the unique property of being applicable, i.e. able to accept objects as arguments and return objects as results.

4.1.5 Summary of fScheme

Tables 4.1 and 4.2 summarise fScheme by listing its expression types and primitive functions, respectively. The keywords of fScheme (`quote`, `lambda`, `if`, `let`, `letrec`, and `define`) should not be used as variables.

- **variable reference**
  ⟨variable⟩
- **literal expression**
  `(quote` ⟨datum⟩`)` or `'`⟨datum⟩
- **function application**
  `(`⟨operator⟩ ⟨operand_1⟩ ...`)`
- **lambda expression**
  `(lambda` ⟨formals⟩ ⟨body⟩`)`
- **conditional expression**
  `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`
- **let expression**
  `(let` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ have the form

  `(((`⟨variable_1⟩ ⟨init_1⟩`)` ...`)`
- **letrec expression**
  `(letrec` ⟨bindings⟩ ⟨body⟩`)` `where` ⟨bindings⟩ have the same form as for the let expression
- **define expression**
  `(define` ⟨name⟩ ⟨expression⟩`)`

Table 4.1: fScheme Expression Types

- **type predicates**
  `symbol?`, `boolean?`, `pair?`, `number?`, and `function?`
- **equivalence predicate**
  `eq?`
- **null list predicate**
  `null?`
- **boolean function**
  `not`
- **pair operations**
  `cons`, `car`, and `cdr`
- **arithmetic operations**
  `+`, `-`, `*`, `quotient`, and `remainder`
- **arithmetic relational operations**
  `=`, `<` and `<=`

Table 4.2: fScheme Primitives

Chapter 5

Environment and Reduction Evaluators for fScheme

In this chapter two radically different evaluators for fScheme are defined. The first is based on an environment evaluation model and the second is based on a reduction evaluation model. Each evaluator accepts an fScheme expression and returns either a value or a message that a name-value binding was entered into the name-space.

The environment evaluation model was described in the last chapter, so for the Environment Evaluator only the following are discussed: organization, execution behavior, and monitoring execution. For the Reduction Evaluator, the abstract reduction rules for fScheme are described before describing the evaluator itself.

5.1 The Environment Evaluator

The Environment Evaluator for fScheme is based on the environment evaluation model [15] [1]. Recall that the fScheme evaluation rules of Chapter 4 assumed an underlying environment evaluator.

5.1.1 Organization

The evaluator consists of five objects that communicate using message passing. Organization of the Environment Evaluator is shown in figure 5.1, and the code for each of the objects is in Appendix A.

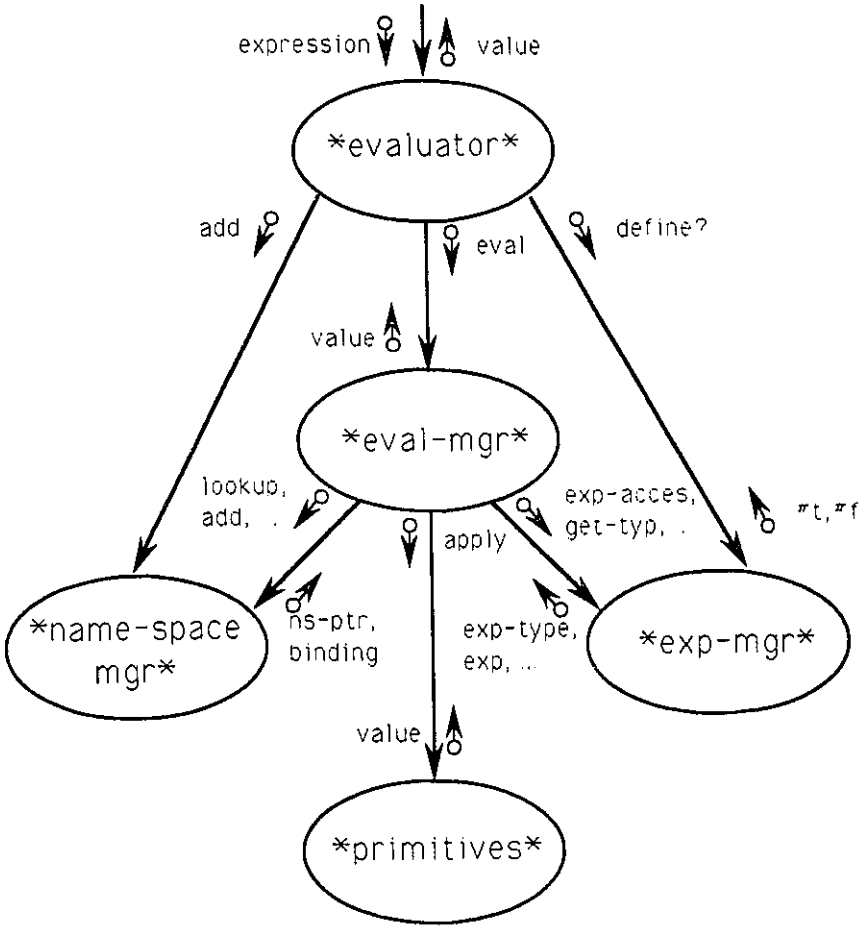


Figure 5.1: Environment Evaluator

Each object in the evaluator is responsible for a significant function in evaluation of an fScheme expression. For example, `*eval-mgr*` manages the evaluation process according to the evaluation rules specified in Chapter 4, `*name-space-mgr*` manages the name-space (name-space corresponds to environment in the environment evaluation model), and so on. In the case of the evaluator, an object-based design permitted a natural division of responsibilities among the objects. Here is a more detailed description of the function of each object:

`*evaluator*` accepts a request to evaluate an fScheme expression, and returns a value or displays a message that a binding was entered into the name-space.

`*evaluator*` first sends an evaluate request to `*eval-mgr*`, and after it receives a value, it either passes that value back to the sender of the request, or (in case of a define expression) sends a request to add a name-value binding to `*name-space-mgr*`.

`*eval-mgr*` manages the evaluation process. It contains the knowledge of evaluation rules of fScheme in the form of a collection of internal procedures.

`*eval-mgr*` accepts a request to evaluate an expression in given name-space, and returns the value of the expression. First, `*eval-mgr*` obtains the type of the expression by sending a `get-type` message to `*exp-mgr*`, and then dispatches on the type to an appropriate procedure. Each expression type (except for the define expression, which is handled in the `*evaluator*`) has a corresponding procedure.

The evaluation routines in `*eval-mgr*` embody the evaluation rules. Any changes to the evaluation rules of fScheme can all be made in `*eval-mgr*`.

`*name-space-mgr*` handles all name-space updates and accesses. It contains the knowledge of name-space representation.

Name-space is an internal state of `*name-space-mgr*`, and `*eval-mgr*` which sends access and update requests as it evaluates an expression in a given environment, has a pointer

into the name-space (`ns-ptr`). This pointer defines that part of the name-space corresponding to the environment in which the expression is evaluated.

`*name-space-mgr*` accepts the following requests: add (add a binding to the global name-space), lookup (lookup a binding in a given name-space), add-list (add a list of bindings to the top frame in the given name-space), extend-ns (extend a given name-space with a new frame containing a given list of bindings), get-global-ns-ptr (get a global name-space pointer), and print-ns (print a given name-space).

`*primitives*` handles application of primitive functions, and their instantiation in the name-space. `*primitives*` contains a table of all the primitive functions, `table-of-primitives`.

`table-of-primitives` is the only place in the environment evaluator where primitives are stored. Any new primitives can simply be added to the table.

`*primitives*` accepts the following requests: apply (apply a given primitive to given arguments), enter-primitives (enter the primitive names in the name-space).

`*exp-mgr*` handles all expression manipulation. It contains all the knowledge of expression representation, so that any changes to expression representation can all be made in `*exp-mgr*`.

`*exp-mgr*` accepts the following requests: exp-access (make a given type of access to a given expression), get-typ (get the type of a given expression), make-closure (make a closure from a given lambda expression and name-space pointer), closure? (test if a given argument is a closure), and define? (test if a given expression is a define expression).

A consequence of the object-based design of the evaluator is that changing the evaluation rules, the set of primitives, expression representation or name-space representation can be done in a systematic way.

5.1.2 Execution Behavior

This section describes the behavior of the Environment Evaluator in terms of actions that are performed for each fScheme expression type.

To evaluate an expression `eval` request is sent to the `*evaluator*` object, which delegates all evaluation to `*eval-mgr*`, and simply passes the result of the evaluation back to the sender. However, in case of the define expression, `*evaluator*` first requests `*name-space-mgr*` to enter a name-value binding in the global name-space, and then displays a message that a binding was entered in the name-space.

The format below follows the description of the form and evaluation rules for fScheme expressions on page 43 of Chapter 4.

- **define expression**
  `(define` ⟨name⟩ ⟨expression⟩)

  1. Obtain a pointer to the global name-space by sending a `get-global-ns-ptr` request to the `*name-space-mgr*`.
  2. Evaluate ⟨expression⟩ in the global name-space by sending an evaluate request to `*eval-mgr*`.
  3. In the global name-space, introduce a new binding of ⟨name⟩ to the result of evaluating ⟨expression⟩, by sending an add-binding request to `*name-space-mgr*`.

- **self-evaluating expression**
  ⟨boolean⟩ or ⟨integer⟩
  Return the expression as is (booleans and integers evaluate to themselves).

- **variable reference**
  ⟨variable⟩
  Lookup the value of the variable in its name-space by sending a `lookup` request to the `*name-space-mgr*`. The `*name-space-mgr*` returns either an association of the variable to its value or a flag indicating that the variable has no association. In case there is no association an error is signaled with a message "`unbound variable`", else the value part of the association is returned.

- **literal expression**
  `(quote` ⟨datum⟩) or '⟨datum⟩
  Extract the literal part of the expression, i.e. ⟨datum⟩, by sending an access request to `*exp-mgr*`. Return that as the result.

- **lambda expression**

  `(lambda` ⟨formals⟩ ⟨body⟩`)`

  Send a `make-closure` request to `*exp-mgr*`. Return that closure as the result. A closure consists of `!closure` tag, the lambda expression and a pointer to the lexical name-space of the lambda expression:

  ```
  (!closure (lambda ⟨formals⟩ ⟨body⟩) . ns-ptr)
  ```

- **function application**[1]

  (⟨function⟩ ⟨argument$_1$⟩ ...)

  1. Send requests to `*exp-mgr*` to access ⟨function⟩ part and ⟨argument$_1$⟩ part.
  2. Recursively invoke the `*eval-mgr*`, by sending an evaluate request for the function-part of the expression and for each of the arguments.
  3. If the function is a primitive, apply the primitive and return that as the result of the application. Otherwise, the function is a closure. Proceed to step 4.

     A primitive is applied by sending an `apply` request to `*primitives*` with a given name and arguments, which either returns the result of applying the primitive function (corresponding to the name) to the arguments, or causes and exception if the name is not a primitive function name.
  4. Extend the name-space of the closure with bindings of variables to argument values, by first sending a request to `*exp-mgr*` to access the variables and the name-space in the closure, and then by sending a request to the `*name-space-mgr*` to extend the closure name-space with the bindings of variables to corresponding argument values.
  5. Evaluate the function body in this extended name-space by sending a request to `*exp-mgr*` to access the function body in the closure, and then send a request to `*eval-mgr*` to evaluate the function body in the extended name-space.

[1]The fScheme evaluation rule in Chapter 4 corresponding to the following steps is on page 40.

- **conditional expression**

  `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`

  1. Send a request to `*exp-mgr*` to access test-part.
  2. Recursively invoke `*eval-mgr*` by sending an evaluate request for the ⟨test⟩ part.
  3. If the result is a true value then recursively invoke `*eval-mgr*` to evaluate the ⟨consequent⟩ part, otherwise evaluate the ⟨alternate⟩ part.

- **let expression**

  `(let` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ have the form

  `((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`

  The behavior in evaluating a let expression is similar to that in evaluating a function application expression, because of the similar semantics of the two expressions. Recall that ⟨variable$_n$⟩ corresponds to a function variable, ⟨init$_n$⟩ corresponds to a function argument, and ⟨body⟩ to function body.

  1. Send requests to `*exp-mgr*` to access ⟨init⟩'s, ⟨variables⟩, and ⟨body⟩.
  2. Recursively invoke `*eval-mgr*` by sending an evaluate request (in the name-space accompanying the let expression) for each of the ⟨init⟩'s.
  3. Extend the name-space of the let expression with bindings of ⟨variables⟩ to ⟨init⟩ values by sending an extend name-space request to the `*name-space-mgr*`.
  4. Evaluate the let body in this extended name-space by sending an evaluate request to `*eval-mgr*`.

- **letrec expression**

  `(letrec` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ are as in the let expression
  Recall that a letrec expression is different from the let expression in that it permits mutually recursive functions to be defined. This is possible because each ⟨init⟩ is evaluated in the name-space extended by bindings of each ⟨variable⟩ to the corresponding ⟨init⟩.

1. Send requests to `*exp-mgr*` to access ⟨init⟩'s, ⟨variables⟩, and ⟨body⟩ (let body).
2. Extend the current name-space with an empty list of bindings by sending an extend name-space request to the `*name-space-mgr*`.
3. Recursively invoke `*eval-mgr*` by sending an evaluate request (in the extended name-space) for each of the ⟨init⟩'s.
4. Add the list of bindings of ⟨variables⟩ to ⟨init⟩ values to the extended name-space by sending an add-list-of-bindings request to `*name-space-mgr*`. Note that this is the same name-space (same ns-ptr) as the one in which the ⟨init⟩'s were evaluated. This is so because in responding to the add-list-of-bindings request, `*name-space-mgr*` does not add a new frame, rather it adds a list of bindings to the top-most frame.
5. Evaluate the let body in this extended name-space by sending an evaluate request to `*eval-mgr*`.

5.1.3 Monitoring Execution Behavior

To be able to analyse the execution behavior of the Environment Evaluator we have instrumented each of the objects in the evaluator. We track significant activity in each of the objects. In chapter 6 we relate these statistics to the computational space and time cost.

- `*eval-mgr*`: maximal recursion depth, number of recursive calls (excluding tail-recursive calls), and number of evaluations.

  Assuming a stack is used to save `*eval-mgr*` state, maximal recursion depth gives an indication of maximal stack size, and number of recursive calls indicates the number of saves and restores from the stack. The number of evaluations gives roughly the number of steps required to evaluate an expression.

  Maximal recursive depth assumes that if `*eval-mgr*` used a stack to explicitly control recursive evaluation of expressions, that `*eval-mgr*` would not stack anything in case of tail-recursive calls. Hence, recursion depth counter is not incremented when a recursive call to `*eval-mgr*` is the last call in an evaluation rule routine.

- `*name-space-mgr*`: number, average and maximal depth of lookups, total number of bindings added, number of frames added, average and maximal depth of frames, and total number of frames added.
- `*primitives*`: number of primitive applies, and in particular number of cons operations.
- `*exp-mgr*`: total number of expression accesses, average and maximal depth of accesses, total number of expression type operations, and total number of cons cells used.

5.2 The Reduction Evaluator

The Reduction Evaluator is based on a version of graph reduction adapted to handle a block-structured, statically scoped functional language such as fScheme. To our knowledge, graph reduction has not been used as a basis for a source-level evaluator (interpreter) for a statically scoped language. In general, graph reduction evaluators accept translated forms of high level functional programming languages. The translation step is used to transform a program into a form in which reduction can be done more efficiently than for a high-level functional language. However, this approach pays for the gain in efficiency by a loss in the ability of directly interacting with the evaluator. For our purposes of studying execution behavior of evaluation models it was more important to be able to interact directly with the evaluators than to have have them evaluate efficiently.

5.2.1 An fScheme Reduction Model

In the case of the string and graph reduction approaches the executing functional program is represented as a string or a graph being reduced rather than evaluated by changing its name-space. Values of variables are not bound to variable names and placed in the updated name-space as a function is applied; instead, the values, in case of string reduction, or pointers to these values, in case of graph reduction, are written directly into a new copy of the function being applied. Applying that

function involves reducing the new function body. Execution terminates when the overall expression is in an irreducible form.

The unusual aspect of the reduction model is that an an expression *is being changed as it is being evaluated.* This is in contrast to an environment evaluator where the name-space is being changed as an expression is being evaluated; the expression itself is left intact (it is read-only). In the environment model it is the name-space that reflects the progress of the computation, while in the reduction model it is the expression itself that does that.

Another difference between the two models is in the way they treat names. Names in the environment model are all treated equally, whether they were defined statically (using a `define` expression) or whether they came to life dynamically as variables in a function application or as variables in let or letrec block invocation. Their bindings are all found in the name-space. On the other hand, in the reduction model the dynamic and static names are treated differently. The static names are found in the name-space, while the dynamic names are rewritten with the corresponding arguments in a function application or the corresponding init expressions in a let or letrec block invocation. As a result, the name-space in the reduction model is a static, read-only structure as an expression is being evaluated, and in the environment model it is a dynamic, read-write structure. Because it is responsible for maintaining the dynamic bindings, the environment name-space is more complex than the reduction name-space. In the reduction model that complexity is transferred to the expression itself during the reduction process through rewriting names with corresponding values. It is interesting to observe that the dynamic name-value associations are distributed over the expression, where they are needed, incurring the access cost at the time of function application rather than at the time at which the name is referenced.

The following abstract reduction rules for fScheme expressions are presented in the same format as the abstract environment evaluation rules on page 43 of Chapter 4.

- **self-evaluating expression**
  ⟨boolean⟩ or ⟨integer⟩
  Return the expression as is (booleans and integers evaluate to themselves).

- **variable reference**
  ⟨variable⟩

Rewrite the expression with the value of the variable in the name-space.

- **literal expression**

  `(quote` ⟨datum⟩`)` or `'`⟨datum⟩

  Rewrite the expression with the literal part of the expression, i.e. ⟨datum⟩.

- **lambda expression**

  `(lambda` ⟨formals⟩ ⟨body⟩`)`

  Rewrite the expression with the corresponding functional value.

- **function application**[2]

  (⟨function⟩ ⟨argument$_1$⟩ ...)

  1. Reduce the arguments.
  2. Copy out the function body expression.
  3. Rewrite all free occurrences of the function's variables in the new function body with pointers to the corresponding arguments.

     All free occurrences of the variables includes all the free occurrences in all the subexpressions, including lambda, let and letrec subexpression. This is the step that ensures that static scoping is maintained. Contrast this with the environment model which ensures static scoping by (1) extending the function definition name-space with the variable-argument associations, (2) and using that name-space when evaluating the function body. This is the central difference between the two models: one rewrites the expression, and the other the name-space.
  4. Rewrite the function application expression with this new function body.
  5. Reduce this new version of the expression.

- **conditional expression**

  `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`

  1. Reduce the test-part of the expression.

[2]The fScheme environment evaluation rule for a function application is on page 40. in Chapter 4

2. If the the test expression reduced to a true value, rewrite the if expression with the consequent-part, else rewrite with the alternate part.
3. Reduce this new expression.

- **let expression**
  `(let` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ have the form

  `((`⟨variable$_1$⟩ ⟨init$_1$⟩`)` ...`)`

  The rule for evaluating a let expression is similar to that of evaluating a function application, the only difference being that there is no need to copy the let body.

  1. Reduce the init expressions.
  2. Rewrite all free occurrences of the let variables in the let body expression with pointers to the corresponding init expressions.
     The meaning of *all free occurrences* of the variables is the same as in the case of the function application.
  3. Rewrite the let expression with this new let body expression.
  4. Reduce this new version of the let expression.

- **letrec expression**
  `(letrec` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ are as in the let expression
  Recall that a letrec expression is different from the let expression in that it permits mutually recursive functions to be defined.

  1. Reduce the init expressions.
  2. In each of the init expression, rewrite all the free occurrences of all the letrec variables with the corresponding init expressions.
     This step is the only difference between the let and letrec reduction rules. It ensures that mutually recursive functions can be defined. Again, the meaning of *all free occurrences* of the variables is the same as in the case of the function application reduction rule.
  3. Rewrite all free occurrences of the letrec variables in the letrec body expression with pointers to the corresponding init expressions.

4. Rewrite the letrec expression with this new letrec body expression.
5. Reduce this new version of the letrec expression.

The define expression is different from the other expression types in that it does not have a reduction rule of its own. It is evaluated for the effect of entering a name-value association in the name-space. Only the names defined through the define expressions are in the name-space.

- **define expression**
  `(define` ⟨name⟩ ⟨expression⟩)

  1. Reduce the value part of the define expression, ⟨expression⟩.
  2. Enter the association of the name and the reduced expression into the name-space.

The Reduction Evaluator does not directly reduce the fScheme expression. It first translates them to a graph form, which just involves adding an an extra cons cell. By choosing cons cells as graph nodes in the Reduction Evaluator, fScheme expressions were almost in graph form. An extra cell was required to be added as the root cell of the graph; the cell that is rewritten with the value of the reduction.

5.2.2 Organization

The Reduction Evaluator has the same general organization as environment evaluator: it consists of five objects that communicate using message passing. The organization of the Reduction Evaluator is shown in Figure 5.2, and the code for each of the objects is given in Appendix B.

Each object in the evaluator is responsible for a significant function in reduction of an fScheme expression. For example, `*red-mgr*` manages the reduction process according to the reduction rules specified in previous section, `*name-space-mgr*` manages the name-space, `*graph-mgr*` manages all the operations directly dealing with the graph representation, and `*primitives*` deals with application of primitive functions. Here is a more detailed description of the function of each object:

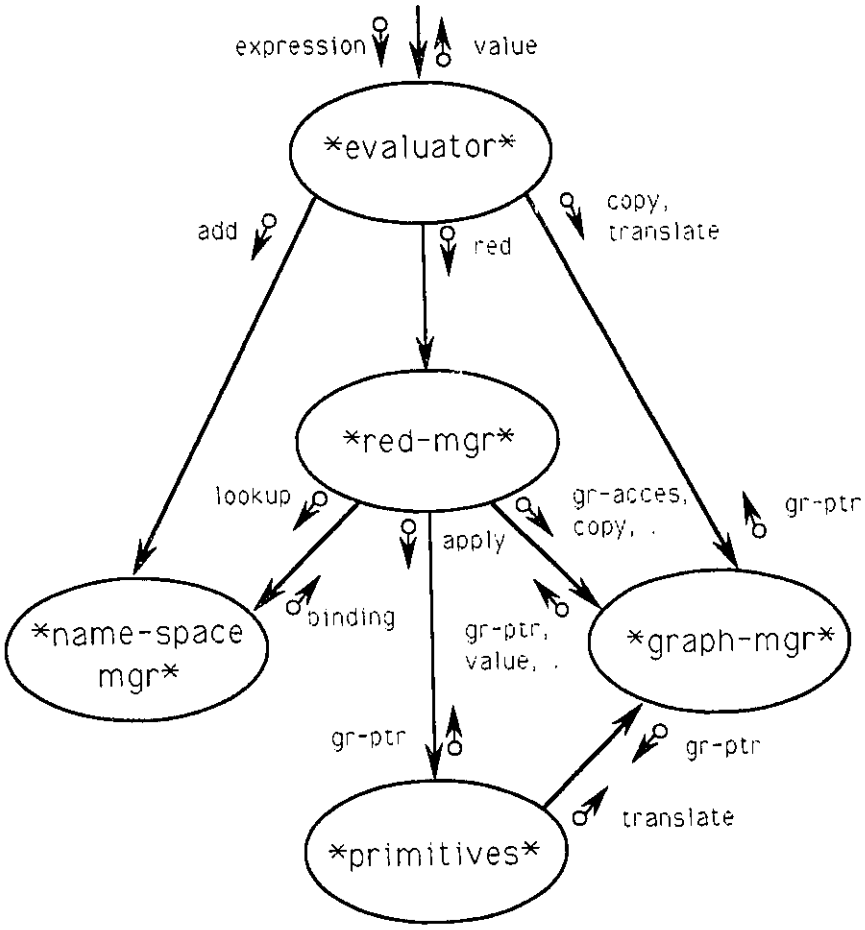


Figure 5.2: Reduction Evaluator

`*evaluator*` as in the Environment Evaluator, it accepts a request to evaluate an fScheme expression, and returns a value or displays a message that a binding was entered into the name-space.

`*evaluator*` first sends a translate and a copy request to the `*graph-mgr*` to transform the fScheme expression into a fresh graph. It then sends a reduce request to the `*red-mgr*`. When the `*red-mgr*` finishes, the graph is in a reduced form. `*evaluator*` then either passes the value in the reduced graph back to the sender of the original evaluate request, or (in case of a define expression) sends a request to add a name-value binding to `*name-space-mgr*`.

`*red-mgr*` manages the reduction process. It contains the knowledge of reduction rules of fScheme in the form of a collection of internal procedures.

`*red-mgr*` accepts a request to reduce a graph (pointed to by `gr-ptr`). First, `*red-mgr*` obtains the type of the expression by sending a `get-type` message to `*graph-mgr*`, and then dispatches on the type to an appropriate procedure. Each graph type has a corresponding procedure.

The reduction routines in `*red-mgr*` embody the reduction rules. Any changes to the reduction rules of fScheme can all be made in `*red-mgr*`.

`*name-space-mgr*` handles all name-space updates and accesses. It contains the knowledge of name-space representation.

Name-space is an internal state of `*name-space-mgr*`. This name-space is a simpler structure than the name-space in the Environment Evaluator because it needs to maintain only the static bindings. During graph reduction, the `name-space-mgr` only deals with lookup requests.

`*name-space-mgr*` accepts the following requests: add (add a binding to the name-space), lookup (lookup a binding in the name-space).

`*primitives*` as in the Environment Evaluator, handles application of primitive functions, and their instantiation in the name-space. `*primitives*` contains a table of all the primitive functions, `table-of-primitives`.

`*primitives*` accepts the following requests: apply (apply a given primitive

to given arguments), enter-primitives (enter the primitive names in the name-space).

`*graph-mgr*` handles all graph manipulation. It contains all the knowledge of graph representation, so that any changes to graph representation can all be made in `*graph-mgr*`.

`*graph-mgr*` accepts the following requests: translate (translate an expression to an equivalent graph), gr-access (make a given type of access to a given graph), copy (copy a given graph), rewrite! (rewrite one graph with the value of another graph), mark! (mark a graph as reduced), rewrite-and-mark! (rewrite one graph with the value of another graph and mark as reduced), get-typ (get the type of a given graph), and find-repl (find and replace all the free occurrences of a given variable in a given graph).

As pointed out for the Environment Evaluator, a consequence of the object-based design is that changing the reduction rules, the set of primitives, graph and expression representation or name-space representation can be done in a systematic way.

5.2.3 Execution Behavior

This section describes the behavior of the Reduction Evaluator in terms of actions that are performed for each fScheme expression type.

To evaluate an expression `eval` request is sent to the `*evaluator*` object, which (1) sends a `translate` request to `*graph-mgr*` to translate the expression into a graph form, (2) sends a `red` request to `*red-mgr*` to reduce the graph, (3) sends an access request to `*graph-mgr*` to obtain the value from the reduced graph, which it then passes to the original sender of the `eval` request. In the case of the define expression `*evaluator*`, after performing the steps (1)-(3) above, sends an `add` request to `*name-space-mgr*` to enter a name-value association in the name-space, and then displays a message that an association was entered in the name-space.

Step 2, reduction of the graph, (managed by `*red-mgr*`) is detailed in the format of the evaluation rules for fScheme expressions on page 43 of Chapter 4.

- **self-evaluating expression**

⟨boolean⟩ or ⟨integer⟩

Mark the graph as reduced by sending a `mark!` request to `*graph-mgr*`.

- **variable reference**

  ⟨variable⟩

  Lookup the value of the variable in the name-space by sending a `lookup` request to the `*name-space-mgr*`. The `*name-space-mgr*` returns either an association of the variable to its value or a flag indicating that the variable has no association. In case there is no association an error is signaled with a message "`unbound variable`", else the variable graph is rewritten with the value part of the association by sending a `rewrite!` request to `graph-mgr`.

- **literal expression**

  `(quote` ⟨datum⟩`)` or `'`⟨datum⟩

  First, access the literal part of the graph by sending a `gr-access` request to `*graph-mgr*`. Then, `rewrite!` with the literal part.

- **lambda expression**

  `(lambda` ⟨formals⟩ ⟨body⟩`)`

  Simply mark the graph as reduced.

  In the Reduction Evaluator lambda graphs are the way functions are represented.

- **function application**

  (⟨function⟩ ⟨argument$_1$⟩ ...)

  1. Send requests to `*graph-mgr*` to access ⟨function⟩ and ⟨argument$_1$⟩ subgraphs.
  2. Recursively invoke the `*red-mgr*`, by sending a `red` request for the function subgraph and for each of the argument subgraphs.
  3. If the function graph is a primitive function, apply the primitive and `rewrite-and-mark!` the application graph with the result of the primitive application. Otherwise, the function is a lambda graph. Proceed to the next step.

4. Copy the function-body graph by sending a `copy` request to the `*graph-mgr*`.
5. Replace the variables in the new function-body graph with the corresponding argument graphs by sending a `find-repl` request for each variable to the `*graph-mgr*`.
6. `Rewrite!` the application graph with this new function body graph.
7. Reduce this new version of the application graph.

- **conditional expression**

  `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`

  1. Send a request to `*graph-mgr*` to access the test subgraph.
  2. Recursively invoke `*red-mgr*` by sending a reduction request for the test subgraph.
  3. If the the test subgraph reduced to a true value, rewrite the if graph with the consequent subgraph, else rewrite with the alternate subgraph.
  4. Reduce this new graph.

- **let expression**

  `(let` ⟨bindings⟩ ⟨body⟩`)` where ⟨bindings⟩ have the form

  `((`⟨variable$_1$⟩ ⟨init$_1$⟩`) ...)`

  1. Send requests to `*graph-mgr*` to access ⟨init⟩'s, ⟨variables⟩, and ⟨body⟩ subgraphs.
  2. Recursively invoke `*red-mgr*` to reduce each of the ⟨init⟩ subgraphs.
  3. Rewrite all free occurrences of the let variables in the let body graph with pointers to the corresponding init graphs by sending `find-repl` request to the `*graph-mgr*`.
  4. Rewrite the let graph with this new let body graph by sending a `rewrite!` request to the `*graph-mgr*`.
  5. Reduce this new graph.

- **letrec expression**
  `(letrec` ⟨bindings⟩ ⟨body⟩) where ⟨bindings⟩ are as in the let expression

  1. Send requests to `*graph-mgr*` to access ⟨init⟩'s, ⟨variables⟩, and ⟨body⟩ subgraphs.
  2. Recursively invoke `*red-mgr*` to reduce each of the ⟨init⟩ subgraphs.
  3. In each of the init subgraphs, rewrite all the free occurrences of all the letrec variables with the corresponding init subgraphs.
  4. Rewrite all free occurrences of the letrec variables in the letrec body graph with pointers to the corresponding init graphs.
  5. Rewrite the letrec expression with this new letrec body expression.
  6. Reduce this new version of the letrec expression.

5.2.4 Monitoring Execution Behavior

As in the case of the Environment Evaluator, all significant activity in each of the objects was monitored:

- `*red-mgr*`: exactly the same statistics were gathered as for the `*eval-mgr*` in the Environment Evaluator. Namely, maximal recursion depth, number of recursive calls (excluding tail-recursive calls), and total number of evaluations.
- `*name-space-mgr*`: number, average and maximal depth of lookups.
- `*primitives*`: number of primitive applies, and in particular number of cons operations. (same as in the Environment Evaluator)
- `*graph-mgr*`: a number of parameters were gathered relating to replacing variables during function application, and during let and letrec block invocation, total number of copy operations, total number of translate operations, total number of graph nodes used, total number of graph accesses, and average and maximal depth of graph accesses.

Chapter 6

Experiments

In this chapter we present the test programs used in generating the execution profiles, the execution profiles themselves, and an analysis of execution costs of one particular execution profile.

6.1 Test Programs

The test programs used in generating the execution profiles were functions `fact`, `fact-iter`, `mrec-even?`, `deriv`, and `tak`.

Functions `fact`, `fact-iter` were already seen before, in Chapter 2, in a discussion on tail-recursion. They are used because they demonstrate that the two evaluators are tail-recursive.

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))

(define fact-iter
  (lambda (product counter)
    (if (<= counter 0)
        product
        (fact-iter (* counter product)
```

```
                    (- counter 1)))))
```

Function `mrec-even?` was chosen to test for mutual recursion.

```
(define mrec-even?
  (lambda (x)
    (letrec ((even?
      (lambda (n)
(if (= n 0)
    #t
    (odd? (- n 1)))))
     (odd?
      (lambda (n)
(if (= n 0)
    #f
    (even? (- n 1))))))
      (even? x))))
```

Function `tak` is a small tree recursive function. It is used in testing function call and recursion. This version was taken from [9].

```
(define tak
  (lambda (x y z)
    (if (not (< y x))
z
(tak (tak (- x 1) y z)
     (tak (- y 1) z x)
     (tak (- z 1) x y)))))
```

Function `deriv` (given in Appendix C) is a medium sized function adapted from *Structure and Interpretation of Computer Programs* [1]. `Deriv` computes a symbolic derivative on its argument with respect to a variable specified by its second argument. It is typical of most functional programs in that it is heavily recursive, and that there are a many small functions called by the main function.

```
Exp: (deriv '(* x (+ y 1)) 'x)
Value: (+ y 1)

eval-mgr* statistics:
number of recursive calls: 224
maximum recursion depth: 7
number of evaluations: 291

name-space-mgr* statistics:
lookup stats: tot 160, avg depth range (10,11), max depth 31
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 44
number of frames added: 33

primitives* statistics:
number of primitive applies: 52
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 341, avg depth range (1,2)
                         max depth 4
number of type-ops: 291
number of cons cells used: 0
```

Table 6.1: Example Execution Profile

6.2 Execution Profiles

Execution profiles from fifteen test runs were generated. An example of an execution profile generated by environment `*evaluator*` is shown in Table 6.1.

The runs are given in Table 6.2. The test runs were made by first adding the appropriate function or functions into an *empty* name-space, and then submitting an eval request to `*evaluator*` object. The name-space was emptied before each run by reloading the `*name-space-mgr*` object into the Scheme interpreter. This was done to make sure that there are no bindings left in the name-space from previous runs that would affect the statistics for the current run.

The execution profiles were all gathered into a log file for each of the two evaluators by a special log function. The log files are given in their raw form in Appendix D .

Expression	Run
(fact 1)	1
(fact 5)	2
(fact 10)	3
(fact-iter 1 1)	4
(fact-iter 1 5)	5
(fact-iter 1 10)	6
(mrec-even? 1)	7
(mrec-even? 10)	8
(mrec-even? 81)	9
(deriv '(* (+ x y) (* x (+ y z))) 'x)	10
(deriv '(* x (+ y 1)) 'x)	11
(tak 1 2 3)	12
(tak 3 2 1)	13
(tak 2 4 8)	14
(tak 8 4 2)	15

Table 6.2: Test Runs

Data from the log files was collected into the following tables. Table 6.2 shows the runs made and gives a numeric key used to index into the other tables. Table 6.3 and Table 6.4 contain data collected from the environment evaluator. Table 6.5 and Table 6.6 contain data from the Reduction Evaluator.

6.3 Analysis of Execution Costs

This section demonstrates how execution profiles can be used to study execution costs. Execution profiles could also be used in fine-tuning the evaluators, deriving requirements for a language-oriented machine architecture, understanding the mechanisms used in evaluating fScheme, and so on. The specific execution profile analyzed for execution cost was generated from evaluating the following fScheme expression:

```
(deriv '(* (+ x y) (* x (+ y z))) 'x)
```

6.3.1 Environment Evaluator

Table 6.7 shows the execution profile analyzed in this section.

eval-mgr			
Run	a	b	c
1	6	2	9
2	54	6	65
3	114	11	135
4	20	2	25
5	72	2	85
6	137	2	160
7	19	2	26
8	100	2	125
9	739	2	906
10	472	9	608
11	224	7	291
12	10	3	13
13	59	4	70
14	10	3	13
15	1676	9	1951

a — number of recursive calls
b — maximum recursion depth
c — number of evaluations

name-space-mgr							
Run	a	b	c	d	e	f	g
1	3	[1,2]	4	[0,1]	1	1	1
2	31	[2,3]	8	[0,1]	1	5	5
3	66	[2,3]	8	[0,1]	1	10	10
4	12	[2,3]	9	[0,1]	1	4	2
5	44	[2,3]	9	[0,1]	1	12	6
6	84	[2,3]	9	[0,1]	1	22	11
7	10	[2,3]	11	[1,2]	3	5	4
8	55	[3,4]	11	[1,2]	3	14	13
9	410	[3,4]	11	[1,2]	3	85	84
10	338	[10,11]	31	[0,1]	1	90	68
11	160	[10,11]	31	[0,1]	1	44	33
12	6	[3,4]	15	[0,1]	1	3	1
13	41	[4,5]	15	[0,1]	1	15	5
14	6	[3,4]	15	[0,1]	1	3	1
15	1196	[4,5]	15	[0,1]	1	411	137

a — total number of lookups
b — average lookup depth range
c — maximal lookup depth
d — average frame depth range
e — maximal frame depth
f — total number of bindings added
g — total number of frames added

Table 6.3: Environment Evaluator Execution Data (Part 1)

primitives		
Run	a	b
1	1	0
2	13	0
3	28	0
4	4	0
5	16	0
6	31	0
7	3	0
8	21	0
9	163	0
10	111	6
11	52	0
12	2	0
13	13	0
14	2	0
15	376	0

a — number of primitive applies
b — number of cons operations

exp-mgr					
Run	a	b	c	d	e
1	10	[1,2]	4	9	0
2	62	[1,2]	4	65	0
3	127	[1,2]	4	135	0
4	23	[1,2]	4	25	0
5	75	[1,2]	4	85	0
6	140	[1,2]	4	160	0
7	29	[2,3]	6	26	8
8	128	[1,2]	6	125	8
9	909	[1,2]	6	906	8
10	709	[1,2]	4	608	0
11	341	[1,2]	4	291	0
12	12	[1,2]	4	13	0
13	62	[1,2]	4	70	0
14	12	[1,2]	4	13	0
15	1712	[1,2]	4	1951	0

a — total number of accesses
b — average access depth range
c — maximal access depth
d — number of type operations
e — total number of cons cells used

Table 6.4: Environment Evaluator Execution Data (Part 2)

	red-mgr			*name-space-mgr*			*primitives*	
Run	a	b	c	d	e	f	g	h
1	6	2	9	2	[1,2]	3	1	0
2	54	6	65	18	[3,4]	7	13	0
3	114	11	135	38	[3,4]	7	28	0
4	20	2	25	6	[2,3]	7	4	0
5	72	2	85	22	[3,4]	7	16	0
6	137	2	160	42	[3,4]	7	31	0
7	19	2	26	4	[3,4]	7	3	0
8	100	2	125	22	[4,5]	7	21	0
9	739	2	906	164	[4,5]	7	163	0
10	472	9	608	179	[17,18]	30	111	6
11	224	7	291	85	[17,18]	30	52	0
12	10	3	13	3	[4,5]	12	2	0
13	59	4	70	18	[5,6]	12	13	0
14	10	3	13	3	[4,5]	12	2	0
15	1676	9	1951	513	[5,6]	12	376	0

a — number recursive calls
b — maximum recursion depth
c — number of reductions
d — total number of lookups
e — average lookup depth range
f — maximal lookup depth
g — number of primitive applies
h — number of cons operations

Table 6.5: Reduction Evaluator Execution Data (Part 1)

graph-mgr												
Run	a	b	c	d	e	f	g	h	i	j	k	l
1	3	1	[15,15]	15	[3,3]	3	40	2	25	18	[1,2]	4
2	15	5	[15,15]	15	[3,3]	3	172	14	121	152	[1,2]	4
3	30	10	[15,15]	15	[3,3]	3	337	29	241	583	[1,2]	4
4	10	4	[16,16]	16	[4,4]	4	79	5	52	630	[1,2]	4
5	30	12	[16,16]	16	[4,4]	4	219	17	152	793	[1,2]	4
6	55	22	[16,16]	16	[4,4]	4	394	32	277	1101	[1,2]	4
7	8	9	[12,13]	30	[5,5]	13	150	4	87	1150	[1,2]	7
8	26	18	[12,13]	30	[4,4]	13	393	22	258	1397	[1,2]	7
9	168	89	[12,13]	30	[3,4]	13	2310	164	1607	3206	[1,2]	7
10	344	90	[27,28]	65	[7,8]	16	3289	112	1857	4629	[1,2]	7
11	177	44	[29,30]	65	[8,9]	16	1700	53	951	5310	[1,2]	7
12	12	3	[34,34]	34	[4,4]	4	82	3	48	5334	[1,2]	7
13	60	15	[34,34]	34	[4,4]	4	366	14	213	5468	[1,2]	7
14	12	3	[34,34]	34	[4,4]	4	82	3	48	5492	[1,2]	7
15	1644	411	[34,34]	34	[4,4]	4	9738	377	5658	9256	[1,2]	7

a — total number of replaces
b — total number of searches
c — average search size range
d — maximum search size
e — average search depth range
f — maximum search depth
g — number of copy operations
h — number of translate operations
i — number of graph nodes (cons cells) used
j — total number of graph accesses
k — average graph access depth range
l — maximum graph access depth

Table 6.6: Reduction Evaluator Execution Data (Part 2)

```
Exp: (deriv '(* (+ x y) (* x (+ y z))) 'x)
Value: (+ (* (+ x y) (+ y z)) (* x (+ y z)))

eval-mgr* statistics:
number of recursive calls: 472
maximum recursion depth: 9
number of evaluations: 608

name-space-mgr* statistics:
lookup stats: tot 338, avg depth range (10,11), max depth 31
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 90
number of frames added: 68

primitives* statistics:
number of primitive applies: 111
number of cons operations: 6

exp-mgr* statistics:
expression access stats: tot 709, avg depth range (1,2)
                         max depth 4
number of type-ops: 608
number of cons cells used: 0
```

Table 6.7: Environment Evaluator Execution Profile for Run 10

Each evaluator object provides statistics related to use of space and time computational resources:

`*eval-mgr*` :

Time is a function of the `number of evaluations` (608) and `number of recursive calls` (472). Maximal space requirement at any one time is a function of `maximum recursion depth` (9). Total space cost is related to amount of stack activity, which in turn is related to `number of recursive calls` (472).

`*name-space-mgr*` :

Time is dominated by the product of `total number of lookups` (338) and `average depth of lookup` (in the range of `[10,11]`) for a total of `[3380,3718]`. `number of bindings added` and `number of frames added` also contribute to the time cost. Total amount of space (in terms of cons cells) is exactly the sum of `number of bindings added` (90) and `number of frames added` (68).

`*primitives*` :

Time is a function of `number of applies` (111), and space is a function of `number of cons operations` (6).

`*exp-mgr*` :

Time is a function of two factors. First, the product of `total number of expression accesses` (709) and `average access depth` (in the range of `[1,2]`) for a total in the range of `[709,1418]`, and second of `number of type operations` (608). Space is exactly a function of `number of cons cells used` (0).

Aggregate time and space costs can be obtained by adding the costs from each object. It is important to realize that such an analysis is highly dependent on *how* the costs are derived for each object. Therefore, in demonstrating how the execution profile in Table 6.7 can be used to obtain aggregate costs, we need to detail how the costs were computed.

The following equations were constructed using the statistics from the execution profile, the evaluator code in Appendix A, and assumptions about stack operation. The constants in the time cost equations for each object were taken directly from the

execution profile in Table 6.7, and the values of various variables were either derived from the code in Appendix A or from assumptions made about stack operation.

The aggregate time cost can be expressed as:

$$\text{Total time} = T_{eval-mgr} + T_{name-space-mgr} + T_{primitives} + T_{exp-mgr} \qquad (6.1)$$

Time cost equation for `*eval-mgr*` is

$$T_{eval-mgr} = 608t_{ev} + 472t_{stack}$$

where t_{ev} = average number of steps in each evaluation
t_{stack} = cost of a single stack operation

$$\begin{aligned} t_{stack} &\approx \text{(frame size)} \\ &\quad \times \text{cost of each push or pop} \\ &= 3 \times 2 \\ &= 6 \end{aligned}$$

The frame size is estimated at 3 because, in case of recursive calls, the `*eval-mgr*` would need to save and restore on the stack three elements: name-space pointer, argument list pointer, and return point. Cost of each push or pop is estimated to be 2 because two operations are performed in each case; for example, for pop, move top stack element and decrement stack pointer.

Average number of steps in each evaluation, t_{ev}, is estimated to be the average over each evaluation type. The time cost for evaluation type was calculated by counting the number of operations in the corresponding evaluation routine in `*eval-mgr*`. It was assumed that the number of arguments is 2 in case of a function apply, while for let and letrec it was assumed that they had 2 variables.

$$t_{ev} = \frac{1}{9}(t_{val} + t_{var} + t_{lit} + t_{lam} + t_{app} + t_{papp} + t_{con} + t_{let} + t_{letrec})$$

$$t_{val} = 0$$

$$
\begin{aligned}
t_{var} &= 3 \\
t_{lit} &= 1 \\
t_{lam} &= 1 \\
t_{app} &= 19 \\
t_{papp} &= 8 \\
t_{con} &= 6 \\
t_{let} &= 15 \\
t_{letrec} &= 15
\end{aligned}
$$

Therefore, $t_{ev} = \frac{68}{9}$, and the time cost for the `*eval-mgr*` is

$$
\begin{aligned}
T_{eval-mgr} &= 608 \times \frac{68}{9} + 472 \times 6 \\
&\approx 7426
\end{aligned}
$$

Time cost equation for `*name-space-mgr*` is

$$T_{name-space-mgr} = 338 \times [10, 11] \times t_{lookup} + 90 t_{add-bind} + 68 t_{add-frame}$$

where $t_{lookup} \approx 1$ (for lookup of depth 1)
$t_{add-bind} \approx 4$
$t_{add-frame} \approx 8$ (assuming 2 bindings per frame)

$$
\begin{aligned}
T_{name-space-mgr} &= 338 \times [10, 11] + 90 \times 4 + 68 \times 8 \\
&\approx 4453 \text{ with } [10, 11] \text{ averaged to } 10.5
\end{aligned}
$$

Time cost equation for `*primitives*` object is

$$T_{primitives} = 111 \times t_{apply}$$

where t_{apply} = time to dispatch + time to apply
$\approx 1 + 1$
$= 2$

$$
\begin{aligned}
T_{primitives} &\approx 111 \times 2 \\
&= 222
\end{aligned}
$$

Time cost equation for `*exp-mgr` object is

$$T_{exp-mgr} = 709 \times [1,2] \times t_{access} + 608 \times t_{type}$$

where $t_{access} = 1$ (access of depth 1)
$t_{type} = \frac{20}{8}$ (average number of steps in determining an expression type)

$$\begin{aligned} T_{exp-mgr} &= 709 \times [1,2] \times 1 + 608 \times \frac{20}{8} \\ &\approx 2584 \text{with} [1,2] \text{averaged to} 1.5 \end{aligned}$$

The total time is a sum of $T_{eval-mgr}$, $T_{name-space-mgr}$, $T_{primitives}$, and $T_{exp-mgr}$:

$$\text{Total time} = 7426 + 4453 + 222 + 2584 = 14685$$

The time cost distribution across the objects is

$$\begin{aligned} \%T_{eval-mgr} &\approx 51\% \\ \%T_{name-space-mgr} &\approx 30\% \\ \%T_{primitives} &\approx 1\% \\ \%T_{exp-mgr} &\approx 18\% \end{aligned}$$

A similar space cost analysis can be performed:

$$\text{Total space} = S_{eval-mgr} + S_{name-space-mgr} + S_{primitives} + S_{exp-mgr} \qquad (6.2)$$

Space cost equation for `*eval-mgr*` is

$$S_{eval-mgr} = 472 s_{stack-frame}$$

where $s_{stack-frame}$ = average stack frame size
$\approx$ 3 (as for $T_{eval-mgr}$ above)

$$\begin{aligned} S_{eval-mgr} &\approx 472 \times 3 \\ &= 1416 \end{aligned}$$

Space cost equation for `*name-space-mgr*`

$$S_{name-space-mgr} = 90 s_{binding} + 68 s_{frames}$$

where $s_{binding}$ = size of binding
= 1 (cons cell)
s_{frames} = size of frame
= 1 (cons cell)

$$\begin{aligned} S_{name-space-mgr} &= 90 + 68 \\ &= 158 \end{aligned}$$

Space cost equation for `*primitives*`

$$\begin{aligned} S_{primitives} &= \text{number of cons operations} \\ &= 6 \end{aligned}$$

Total space is 1580. The space cost distribution across the objects is

$$\begin{aligned} \%S_{eval-mgr} &\approx 90\% \\ \%S_{name-space-mgr} &\approx 10\% \end{aligned}$$

6.3.2 Reduction Evaluator

Table 6.8 shows the execution profile analyzed in this section.

As in the previous section, it is shown how the execution profile depicts the time and space computational costs:

`*red-mgr*` :

Time is a function of the `number of reductions` (608) and `number of recursive calls` (472). Maximal space requirement at any one time is a function of `maximum recursion depth` (9). Total space cost is related to amount of stack activity, which in turn is related to `number of recursive calls` (472).

```
Exp: (deriv '(* (+ x y) (* x (+ y z))) 'x)
Value: (+ (* (+ x y) (+ y z)) (* x (+ y z)))

red-mgr statistics:
number of recursive calls: 472
maximum recursion depth: 9
number of reductions: 608

name-space-mgr* statistics:
lookup stats: tot 179, avg depth range (17,18), max depth 30

primitives* statistics:
number of primitive applies: 111
number of cons operations: 6

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 344
         tot number of searches is 90
         avg search size range (27,28), max search size 65
         avg search depth range (7,8), max search depth 16
number of copy operations: 3289
number of translate operations: 112
number of graph nodes (cons cells) used: 1857
graph access stats: tot 4629, avg depth range (1,2)
                    max depth 7
```

Table 6.8: Reduction Evaluator Execution Profile for Run 10

name-space-mgr :

Time is dominated by the product of total number of lookups (179) and average depth of lookup (in the range of [17,18]) for a total of [3043,3222]. There is no space cost incurred.

primitives :

Time is a function of number of applies (111), space is a function of number of cons operations (6).

graph-mgr :

Time is a function of 5 factors. First, product of total number of searches (90) and average search size (in the range of (27,28)) for a total of (2430,2520). Second, number of copy operations (3289). Third, number of translate operations (112). Fourth, product of total number of graph accesses (4629) and average depth (in the range of (1,2)) for a total of (4629,9258). Space is a function of total number of graph nodes used (1857).

As for the Environment Evaluator, aggregate time and space costs were obtained for the Reduction Evaluator by combining the costs from each of its object. The following time and space cost equations were constructed using the statistics from the execution profile in Figure 6.8, the evaluator code in Appendix B, and assumptions about stack operation. The constants in the time cost equations for each object were taken directly from the execution profile in Table 6.8, and the values of various variables were either derived from the code in Appendix B or from assumptions made about stack operation.

The aggregate time cost is

$$\text{Total time} = T_{red-mgr} + T_{name-space-mgr} + T_{primitives} + T_{graph-mgr} \qquad (6.3)$$

$$T_{red-mgr} = 608t_{red} + 472t_{stack}$$

where t_{red} = average number of steps in each reduction

t_{stack} = cost of a single stack operation

$$
\begin{aligned}
t_{stack} &\approx \text{(frame size)} \\
&\quad \times \text{cost of each push or pop} \\
&= 2 \times 2 \\
&= 4
\end{aligned}
$$

The frame size is estimated at 2 because, in case of recursive calls, the `*red-mgr*` would need to save and restore on the stack the graph pointer, and the return point. Cost of each push or pop is estimated to be 2 as for the Environment Evaluator.

Average number of steps in each reduction, t_{red}, is estimated to be the average over each type of reduction. The time cost for type of reduction was calculated by counting the number of operations in the corresponding reduction rule routine in `*red-mgr*`. As for the Environment Evaluator, it was assumed that the number of arguments is 2 in case of a function apply, while for let and letrec it was assumed that they had 2 variables.

$$t_{red} = \frac{1}{9}(t_{val} + t_{var} + t_{lit} + t_{lam} + t_{app} + t_{papp} + t_{con} + t_{let} + t_{letrec})$$

$$
\begin{aligned}
t_{val} &= 1 \\
t_{var} &= 4 \\
t_{lit} &= 2 \\
t_{lam} &= 1 \\
t_{app} &= 16 \\
t_{papp} &= 6 \\
t_{con} &= 9 \\
t_{let} &= 13 \\
t_{letrec} &= 10
\end{aligned}
$$

Therefore, $t_{red} = \frac{62}{9}$, and the time cost for the `*red-mgr*` is

$$T_{red-mgr} = 608 \times \frac{62}{9} + 472 \times 4$$
$$\approx 6076$$

Time cost equation for `*name-space-mgr*` is

$$T_{name-space} = 179 \times [17,18] \times t_{lookup} = [3043, 3222]$$
$$\approx 3133 \text{ with } [17,18] \text{ averaged to } 17.5$$

where $t_{lookup} \approx 1$ (for lookup of depth 1)

Time cost equation for `*primitives*`

$$T_{primitives} = 111 \times t_{apply}$$

where t_{apply} = time to dispatch + time to apply
+ time to translate
≈ 3

$$T_{primitives} = 111 \times 3$$
$$\approx 333$$

Time cost equation for `*graph-mgr*`

$$T_{graph-mgr} = 90 \times [27,28] \times t_{search} + 3289 \times t_{copy}$$
$$+112 t_{translate} + 4629 \times [1,2] \times t_{access}$$
$$+344 \times t_{replace}$$

where t_{search} = average number of steps for each search type
$t_{translate} = 1$
$t_{access} = 1$ (access of depth 1)
$t_{copy} = 3$ (average number of steps for a copy operation)
$t_{replace} = 1$

$$t_{search} = \frac{1}{10}(t_{sval} + t_{slit} + t_{sreduced} + t_{svar} + t_{slam} + t_{slet} + t_{sletrec1} + t_{sletrec2} + t_{scon} + t_{sapp})$$

$$\begin{aligned}
t_{sval} &= 1 \\
t_{slit} &= 1 \\
t_{sreduced} &= 1 \\
t_{svar} &= 3 \\
t_{slam} &= 4 \\
t_{slet} &= 6 \\
t_{sletrec1} &= 3 \\
t_{sletrec2} &= 8 \\
t_{scon} &= 3 \\
t_{sapp} &= 3
\end{aligned}$$

Therefore, $t_{search} = \frac{33}{10}$, and the time cost for the `*graph-mgr*` is

$$\begin{aligned}
T_{graph-mgr} &= 90 \times [27, 28] \times \frac{33}{10} + 3289 \times 3 \\
&\quad +112 + 4629 \times [1, 2] + 344 \\
&= [22971, 27897] \\
&\approx 25434 \text{ average of } [22971, 27897]
\end{aligned}$$

The total time is a sum of $T_{red-mgr}$, $T_{graph-mgr}$, $T_{primitives}$, and $T_{name-space-mgr}$:

$$\text{Total time} = 6076 + 3133 + 333 + 25434 = 34976$$

The time cost distribution across the objects is

$$\begin{aligned}
\%T_{red-mgr} &= 17\% \\
\%T_{name-space} &= 9\% \\
\%T_{primitives} &= 1\% \\
\%T_{graph-mgr} &= 73\%
\end{aligned}$$

As for the Environment Evaluator, a similar cost analysis can be performed:

$$\text{Total space} = S_{red-mgr} + S_{name-space-mgr} + S_{primitives} + S_{graph-mgr} \tag{6.4}$$

$$S_{red-mgr} = 472 s_{stack-frame}$$

$$\begin{aligned} \text{where} \quad s_{stack-frame} &= \text{average stack frame size} \\ &\approx 2 \end{aligned}$$

$$\begin{aligned} S_{red-mgr} &\approx 472 \times 2 \\ &\approx 944 \end{aligned}$$

$$S_{name-space-mgr} = 0$$

$$\begin{aligned} S_{primitives} &= \text{number of cons operations} \\ &= 6 \end{aligned}$$

$$\begin{aligned} S_{graph-mgr} &= \text{number of graph nodes (cons cells) used} \\ &= 1857 \end{aligned}$$

The total space cost is is 2807. Only the `*graph-mgr*` and `red-mgr*` contribute significantly to the space cost

$$\begin{aligned} \%S_{graph-mgr} &\approx 66\% \\ \%S_{red-mgr} &\approx 34\% \end{aligned}$$

Environment	
T_{env}	14685
$\%T_{eval-mgr}$	51%
$\%T_{name-space-mgr}$	30%
$\%T_{primitives}$	1%
$\%T_{exp-mgr}$	18%
S_{env}	1580
$\%S_{eval-mgr}$	90%
$\%S_{primitives}$	10%

T_{env} — total environment time cost
S_{env} — total environment space cost

Reduction		
T_{red}	34976	$2.38T_{env}$
$\%T_{red-mgr}$	17%	
$\%T_{name-space-mgr}$	9%	
$\%T_{primitives}$	1%	
$\%T_{graph-mgr}$	73%	
S_{red}	2807	$1.78S_{env}$
$\%S_{red-mgr}$	34%	
$\%S_{graph-mgr}$	66%	

T_{red} — total reduction time cost
S_{red} — total reduction space cost

Table 6.9: Summary of Execution Costs for Run 10

6.4 Discussion of Experiments

The execution profile from Table 6.7 is similar to the other execution profiles (refer to Appendix D for details) in terms of distribution of activity among the different evaluator objects. Because of that, the above cost analysis would yield similar cost distribution for the other test cases. This applies to the Reduction Evaluator as well.

From Table 6.9, which summarizes the execution cost derived in the previous section, we can see that the dominant costs in both space and time in the Environment Evaluator are incurred in `*eval-mgr*` and `*name-space-mgr*`. Together they account for 81% of the time cost and nearly 100% of the space cost. If we were to improve the performance of the Environment Evaluator, the greatest payoff would be obtained from optimizing the `*eval-mgr*` and the `*name-space mgr*` objects.

In the Reduction Evaluator, the costs are even more unevenly distributed. `*red-mgr*` and `*graph-mgr*` account for 90% of the time cost and nearly 100% of the space cost. Clearly, these two objects are the key to performance of the Reduction Evaluator.

As expected the Environment Evaluator was more efficient than the reduction one. The reduction time cost was roughly 238% of the environment time cost, and 178% of the space cost. Considering that the Reduction Evaluator implemented a source level reduction of a statically-scoped language, the results are encouraging for the reduction model.

The requirement for static scoping was unfair towards the reduction model because static scoping is cheaper to implement under the environment model than under the

reduction model. In the Environment Evaluator all it required was an extra element in the functional value to hold the pointer to the definition environment of the function; essentially incurring only a small space cost. In the reduction model it required an expensive search and replace over a graph each time a function was applied or a let or letrec were invoked. And there appears to be no way around incurring this cost if static scoping is to be supported under a naive reduction model such as ours. The cost involved in this operation can be estimated to be 33% of the total time cost associated with the `*graph-mgr*` (33% figure is derived in the next paragraph). This cost would be largely eliminated if dynamic scoping was used because that would eliminate the need for searching through a graph to substitute for the variables, and a more efficient mechanism could be used for establishing a relationship between arguments and variables (such as an argument stack, i.e. spine stack [20]). This would make the reduction time cost closer to 200% of the environment time cost.

The percentage of the time cost attributable to find and replace operation (responsible for supporting static scoping) is derived from the time cost equation for `*graph-mgr*` as follows. Recall the time cost equation for `*graph-mgr*`,

$$\begin{aligned} T_{graph-mgr} &= 90 \times [27,28] \times t_{search} + 3289 \times t_{copy} \\ &\quad +112 t_{translate} + 4629 \times [1,2] \times t_{access} \\ &\quad +344 \times t_{replace} \end{aligned}$$

where $t_{search} = \frac{33}{10}$

$t_{translate} = 1$

$t_{access} = 1$

$t_{copy} = 3$

$t_{replace} = 1$

The cost of find and replace operation, $t_{find-repl}$ is

$$t_{find-repl} = 90 \times [27,28] \times t_{search} + 344 \times t_{replace}$$

Hence the percentage of the time attributable to find and replace is

$$\frac{t_{find-repl}}{T_{graph-mgr}} = 33\%$$

Ultimately, the reason for the relatively large costs of the Reduction Evaluator are that the reduction model is inherently more expensive than the environment model. Given equal degrees of optimization, the reduction model will always be more expensive, because it distributes the computation, and hence pays correspondingly higher management overheads. These costs could be decreased by restricting reduction only to the dynamic parts of the computations (as in the dataflow model). This would indicate that dataflow is a more promising approach to distributing computation because it achieves parallelism without paying the high costs of the purely reduction-based model.

Another approach to optimizing the reduction evaluation model is through reducing the complexity of functions. Called combinator reduction, this model requires that all free variables are compiled out from functions, so that the behavior of functions (combinators) is determined solely by their arguments and not by statically scoped free variables.

Here we intentionally followed a naive approach to implementing the two evaluation models. We were interested in first studying the the unoptimized versions of the two approaches, and then using the time and space costs (however poor they may be) to identify opportunities for optimization.

The reduction model is more expensive because it is a more general evaluation model than the environment model. The reduction model is less a function of the underlying architecture than the environment model, and more a function of the semantics of functional languages. In fact, the promise of the reduction model lies precisely in its semantic proximity to functional languages. It is because of that it can be mapped to a wide array of sequential and parallel machine architectures. However, it will always pay a higher total cost than the environment model. These costs may be distributed over time as is the case for a sequential architecture or they may be distributed over space as would be the case on a parallel architecture.

Chapter 7

Conclusions

This chapter first presents an overview of the thesis and its results. It then briefly describes related work, the scope of the methodology used in this thesis, some possible extensions, and future research.

7.1 Thesis Overview and Results

In this thesis a methodology was developed to analyse, compare and prototype evaluation models for functional programming. The first step was to identify the essential abstractions of functional programming and to embody them in a programming language. The programming language used was fScheme, a small functional subset of the Scheme dialect of Lisp. The next step was to analyse different evaluation models and to develop a framework for building corresponding evaluators. As a result, Environment and Reduction Evaluators were built and instrumented to provide statistics on various critical operations in the evaluation process. Statistics were gathered for a number of fScheme programs. These statistics were the basis for an execution cost analysis and comparison. They were also used to identify how the execution activity was distributed over different elements of the evaluators. The process of designing, building, and instrumenting the evaluators is essentially the process of prototyping functional programming language implementations.

Towards identifying the essential abstractions of functional programming an in-depth analysis of the functional programming paradigm was conducted. This analysis

served a twofold purpose: first, it established an approach to analyzing programming paradigms in general, and second, it provided a starting point for studying any prospective evaluation models for functional programming languages. The analysis hinged on the observation that a programming language is essentially a *system of abstractions* that is used to build programs that execute on some evaluator. Given that the first step in the design of any programming language evaluator is to identify the abstractions that constitute the language, the methodology that was developed here for specifically investigating functional programming evaluation models is equally well suited for investigating evaluation models for other types of programming languages.

The Scheme programming language was used both as a source of language elements for fScheme (the programming language designed here to embody the core system of functional abstractions), and as the implementation language for the evaluators of fScheme (MIT Scheme[1] was the specific dialect of Scheme used). It was natural to choose Scheme in both roles for three reasons:

1. the core of Scheme is a small lambda-calculus based functional programming language.

2. Scheme can accept its own expressions as data, and manipulate them just like any other data object.

3. Scheme, like all other Lisps, is an interpreted language, allowing for interactive and relatively fast program development.

After analyzing the functional programming paradigm, various evaluation models were surveyed and an object based approach was used in designing two instrumented fScheme evaluators embodying, respectively, the environment and reduction evaluation models. These evaluators were later used to generate execution profiles for a number of fScheme programs. The execution profiles gave an indication of the computational cost and its distribution among different objects of the evaluators. In this way a finer view of the execution cost was obtained than one given by such aggregate statistics as total time and total memory cost (such as obtained in [12] in a comparison of execution efficiency of an environment and reduction evaluators). Our

[1]MIT Scheme and other dialects of Scheme are available from **altdorf.ai.mit.edu** by anonymous ftp.

figures on cost distribution identified which objects in the evaluator needed to be optimized or which objects required support from an underlying machine to be feasible. Hence, the execution profiles were shown to be a way of exposing the consequences on computational cost of design decisions made at the evaluation model level.

The object-based organization of the Environment and Reduction Evaluator has the following advantages:

- flexible with respect to changes in a given programming language in that new expression types (constructs) or new primitives could be introduced by changing only the relevant objects and reusing all the other ones. For example, if fScheme was to be extended with a new expression type, the following changes would be made to the Environment Evaluator: new evaluation rule in `*eval-mgr*` object, and new expression access and type determination message handlers in `*exp-mgr*`. `*Primitives*` object would be unchanged, and so would `*name-space-mgr*` object if the new expression did not require a special type of name-space operation. Also, a new fScheme primitive can be added by simply changing the `table-of-primitives` structure in the `*primitives*` object.

- flexible with respect to programming languages in that similar types of objects can be used to evaluate languages other than fScheme.

- flexible with respect to each evaluation model in that each evaluator object can be at an arbitrarily low level, making it possible to provide an arbitrarily refined view of that part of the execution process for which the object is responsible. Also, different tactics for implementing the evaluation model can be investigated.

- versatile in that different evaluation models can be studied in a common object-based framework.

- amenable to instrumenting. Each evaluator object manages statistics gathering for the part of the execution process it is responsible. The statistics are part of an object's local state.

Use of the same functional programming language as a starting point for development of two different evaluators is unique to this thesis. There are quantitative studies

of functional programming systems, however, with the exception of [12] (discussed in more detail in the next section), they study one evaluation model at a time.

The Reduction Evaluator for fScheme is unique in the following ways:

- defines a reduction semantics for a lexically scoped language.
- defines an (almost) source level reduction interpreter for fScheme. It is "almost" a source level interpreter because the translation step from fScheme source form to graph form simply involves adding an extra cons cell to the fScheme expression representation.
- offers a basis for a reduction semantics for Scheme, a non-functional programming language.

Computation costs of simulations are the major cause of their limitations. The two object-based evaluators are no exception. Even though the execution process was monitored at a fairly high-level the time it took to evaluate even the smallest programs was significant (much greater than running the same programs directly in the Scheme interpreter itself). Monitoring the execution process at a lower level would have been even slower. However, when viewed as the first step, a prototype, a simulation serves important goals: to test design concepts, validate ideas and help structure the final implementation.

7.2 Related Work

The only other study that we are aware of that conducted a quantitative analysis of the environment and reduction based evaluators for functional programs is presented in *Comparing Two Functional Programming Systems* [12]. The authors investigated execution efficiency of complete functional programming systems, and therefore concentrated on obtaining aggregate performance figures. They compared execution efficiency of an environment-based evaluator for FP[4] to that of a graph reduction-based one for SRL (Simple Reduction Language). The evaluators were implemented using different languages, ran on different machines and accepted different functional languages. To isolate execution efficiency of the evaluation approaches a comparison was made between execution times of

"... a set of benchmarks written in the source languages to the same benchmarks written in the underlying implementation language of the interpreter."[12]

The ratio of execution times effectively factored out the underlying machine and implementation language. And so the comparison of the ratios for the two systems gave a measure of relative execution efficiencies of the two implementation approaches with respect to a set of typical applications.

The figures they obtained are useful more in choosing a specific functional programming system, than in understanding the evaluation mechanism for functional programs. In this thesis, we were interested more in understanding the reasons for a given execution behavior than in obtaining data about aggregate performance.

Performance and Evaluation of Lisp Systems [9] is another related study. However, its methodology was closer to the one used in this thesis because it was based on the premise that

> ... benchmarking is most effective when performed in conjunction with an analysis of the underlying Lisp implementation and computer architecture.

Consequently, the aggregate performance figures of the twelve Lisp systems that were considered were explained only after an analysis of general Lisp implementation requirements and of the way in which each system met these requirements. The results were useful both for choosing an appropriate system, and, with the accompanying analysis, for understanding the reasons for differences in performance.

7.3 Scope of the Methodology

The methodology was developed to aid in making the initial, high level decisions in language implementation: the type of evaluation model to use, identifying the major components in an implementation, defining high level behavior and architecture of the implementation. However, it can also be used in studying lower level evaluators. For example, the high-level `*eval-mgr*` of the Environment Evaluator, can be extended to explicitly implement control passing, rather than rely on underlying Scheme mechanisms for control passing. This could be done by including a `*stack*` object

on which `*eval-mgr*` could save the state of its computation as it passes control to another part of itself. To continue the suspended computation it would simply restore its state from the `*stack*` object (much as is done in the Explicit-Control Evaluator of Abelson and Sussman [1]). Conceivably, even such low level evaluators as processors could be studied this way. At some point it becomes computationally infeasible to study increasingly lower level evaluators, and one must use a combination of interacting tools: high level evaluators using the information either from lower level simulators or from measurements on actual prototype hardware to predict execution behavior.

With addition of a translator between the programming language and the evaluator, the implementation of compiled high level programming languages can be studied. In this case the interaction between the compiler and the evaluator can be analyzed: varying the compiler while keeping the evaluator constant to see the effects on execution behavior of different compilation strategies, and similarly varying the evaluator while keeping the compiler constant to see the effects of different implementation techniques for the same evaluation model.

While this thesis focused on studying existing evaluation models for existing functional programming languages, another way in which the methodology could be used is as way of prototyping new evaluation models and developing new tactics for implementing existing evaluation models. It could also be used for developing implementations of new language features and for quantifying their execution costs.

The object-based framework can be more than just a medium for experimenting with evaluation models. It can also be a tool for a language designer to experiment with new language features, as well as a medium for communicating application and language level requirements to the computer architect. As such, it has a potential in education, helping students get a quantitative perspective on interaction among programming language, its implementation, and underlying system.

7.4 Extensions

In the short term the following additions and improvements could be made:

- compare the evaluators more extensively.

- analyse and compare the suitability of the two evaluators as a basis for parallel evaluation of fScheme programs. Develop simulations for parallel evaluators and instrument them to provide parallelism profiles.

- study extensions to fScheme and their cost.

- extend to other programming paradigms and corresponding languages. In particular, extend to the object oriented paradigm and study the costs of supporting it.

- related to above, extend to study of implementations of multiparadigm programming languages. There is clearly promise in unifying different programming paradigms in a single programming language. One approach to implementing such a programming language is by developing a unified evaluation model that would incorporate the functionality of all the individual evaluation models for all the different paradigms supported in the language. By first developing that unified evaluation model and then studying its execution behavior over real applications we expose different implementation approaches. In presence of different paradigms and different options for the underlying computer architecture it is important to be able to formulate and quantitatively (even if only at a high level) evaluate different implementation approaches. Exposing alternatives for the unified evaluation model of multiparadigm programming languages may be the way toward true "general purpose" computers of the future.

- obtain a more refined view of the execution behavior of the two evaluators by introducing lower level objects that represent actual machine elements. Statistics could be gathered at this level. In view of the computation cost of the high-level simulation, particular attention needs to be paid to making the low level simulation feasible.

7.5 Future Research

In the long term the following research directions appear promising:

- develop an object-based framework for studying distributed High-Level Language evaluators, similar to the *The High-Level Language Architecture* proposed in [25].

- develop an object-oriented database of evaluators, like An Object-Oriented VLSI CAD Framework in [10]. The database could serve as a repository for different evaluation models and their evaluators.

- develop flexible executors that chose evaluators that best match the underlying machine organization. For example environment evaluator for von Neumann type architectures, reduction evaluator for a graph reduction machine, and so on. This flexibility could possibly be extended to applications, i.e. an executor that that chooses an evaluation model that matches the application to the underlying architectures.

- use the object-based evaluators as a basis for developing a system for visualizing the execution behavior of functional programs. This could be similar to the MulTVision system developed for visualizing programs written in Multilisp [13].

Bibliography

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

[2] Arvind and R.S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. Computation Structures Group Memo 271, MIT Laboratory for Computer Science, March 1987.

[3] Arvind, R.S. Nikhil, and K.K. Pingali. Id nouveau reference manual, part 2: Operational semantics. Computation structures group, MIT Laboratory for Computer Science, April 1987.

[4] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, vol. 21(8):613–641, August 1978.

[5] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[6] W. Clinger and J. Rees (Editors). Revised^4 Report on the Algorithmic Language Scheme, November 1991. Draft report obtained from an MIT public account (**altdorf.ai.mit.edu**).

[7] J. Darlington. Functional programming. In F.B. Chambers, D.A. Duce, and G.P. Jones, editors, *Distributed Computing*, chapter 5. Academic Press, Inc., 1984.

[8] A.J. Field and P.G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley, 1988.

[9] R.P. Gabriel. *Performance and Evaluation of Lisp Systems.* Computer Systems Series. MIT Press, 1985.

[10] R. Gupta, W.H. Cheng, R. Gupta, I. Hardonag, and M.A. Breuer. An Object-Oriented VLSI CAD Framework. *IEEE Computer*, vol. 22(5), May 1989.

[11] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, pages 34–52, January 1985.

[12] B. Hailpern, T. Huynh, and G. Revesz. Comparing Two Functional Programming Systems. *IEEE Transactions on Software Engineering*, vol. 15(5), May 1989.

[13] R.H. Halstead, Jr and D.A. Kranz. A Replay Mechanism for Mostly Functional Parallel Programs. Technical report, DEC Cambridge Research Lab, November 1990.

[14] P. Hancock. Polymorphic type-checking. In S.P. Jones, editor, *The Implementation of Functional Programming Languages*, chapter 8. Prentice-Hall International, 1987.

[15] P. Henderson. Functional Programming *Application and Implementation.* Series in Computer Science. Prentice-Hall International, 1980.

[16] C. Hewitt and H. Lieberman. Design issues for parallel architectures for artificial intelligence. A.I. Memo 750, MIT A.I. Laboratory, November 1983.

[17] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, vol. 21(3):359–411, September 1989.

[18] P. Hudak and B. Goldberg. Serial Combinators: "Optimal" Grains of Parallelism. In *Lecture Notes in Computer Science, Vol. 201.* Springer-Verlag, 1985.

[19] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Lecture Notes in Computer Science, Vol. 201.* Springer-Verlag, 1985.

[20] S.P. Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall International, 1987.

[21] P.M. Kogge. *The Architecture of Symbolic Computers.* Series in Supercomputing and Parallel Processing. McGraw-Hill, 1991.

[22] H. Ledgard and M. Marcotty. *The Programming Language Landscape.* The SRA Computer Science Series. SRA (Science Research Associates), 1981.

[23] G. Mago and D. Middleton. The FFP Machine—A Progress Report. In *Proceedings of the International Workshop on High- Level Language Computer Architectures*, 1984.

[24] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part 1. *Communications of the ACM*, pages 184–195, April 1960.

[25] G. Mousseau and M. Krieger. A Distributed High-Level Language Architecture. In *Proceedings of the the Seventeenth Annual Conference on Information Sciences and Systems*, 1983.

[26] D.A. Patterson and J.L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[27] T.W. Pratt. Programming Languages *Design and Implementation.* Prentice-Hall, 1984.

[28] G.L. Steele Jr. and G.J. Sussman. Design of a LISP-Based Microprocessor. *Communications of the ACM*, vol. 23(11):628–645, November 1980.

[29] S.R. Vegdahl. A Survey of Proposed Architectures for Functional Languages. *IEEE Transactions on Computers*, vol. 23(12), December 1984.

[30] P. Wadler. List comprehensions. In S.P. Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7. Prentice-Hall International, 1987.

Glossary

data abstraction (p 16): is the ability to name and manipulate compound data objects.

data composition (p 16): is the ability to build a data structure out of simpler data objects.

evaluation model (p 4): an abstract model of execution behavior of programming language components and their combinations.

evaluation rules (p 4): a set of rules for a programming language that define behavior of all the language abstractions.

functional abstraction (p 9): the ability to define and then apply a function.

function composition (p 9): the facility for combining simpler functions into complex expressions.

mutable state (p 13): is an abstraction for a read/write memory location. In imperative programming languages it is supported by the following: variable as a named location, assignment operation for setting a named location, and sequencing of operations. In functional programming there is no concept of mutable state.

programming language (p 3): a *medium for communicating* with a computer and a *system of abstractions* that is used to build programs that execute on that computer.

referentially transparent (p 14): a property of expressions of a programming language such that

...each expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of a program to share the expression [8].

Appendix A

Environment Evaluator Code

This appendix contains the Environment Evaluator code consisting of the `*evaluator*`, `*eval-mgr*`, `*name-space-mgr*`, `*primitives*`, and `*exp-mgr*` objects.

A.1

```
;;; ************************
;;; ************************
;;; *evaluator* object
;;; accepts: eval (fScheme expression), init-stats, and get-stats
;;; returns: value of the expression


(define (make-evaluator)

  (define (dispatch m)
    (case m
      ((eval) eval)
      (else (error "Error in *evaluator*: received unknown message " m))))

  ; msg handlers

  (define (eval exp)
        ; initilise statistics
      ((*eval-mgr* 'init-stats))
      ((*name-space-mgr* 'init-stats))
      ((*primitives* 'init-stats))
      ((*exp-mgr* 'init-stats))
         ; evaluate expression
      (let ((value
     ((*eval-mgr* 'eval)
       ((*exp-mgr* 'exp-access) 'value-part exp)
       (*name-space-mgr* 'get-global-ns-ptr))))
(if ((*exp-mgr* 'define?) exp)
     ; if define expression send message to *name-space-mgr*
     ; to enter the name-value association into the name space
  (let ((name ((*exp-mgr* 'exp-access) 'name-in exp)))
    ((*name-space-mgr* 'add) name value)
    (newline)
    (display "added variable ")
    (write name)
    (display " to the name-space"))
     ; else it is not a define expression, so simply return the
     ; value of the expression.
  (begin
    (newline)
    (display "Value: ")
    (write value)
    (newline) (newline)
    (display "*eval-mgr* statistics: ")
    (newline)
    ((*eval-mgr* 'get-stats))
    (newline) (newline)
    (display "*name-space-mgr* statistics: ")
    (newline)
    ((*name-space-mgr* 'get-stats))
              (newline) (newline)
              (display "*primitives* statistics: ")
              (newline)
              ((*primitives* 'get-stats))
    (newline) (newline)
              (display "*exp-mgr* statistics: ")
              (newline)
              ((*exp-mgr* 'get-stats))))))


  dispatch) ; end make-evaluator


; create *evaluator* object

(define *evaluator* (make-evaluator))
```

A.2

```
;;; **************************
;;; **************************
;;; "eval-mgr" object
;;; accepts: eval (exp, ns-ptr), init-stats, and get-stats
;;; eval: evaluate the expression exp in the name-space pointed to
;;;       by ns-ptr. Return the value.
;;; internal state: evaluation statistics


(define (make-eval-mgr)

  (define (dispatch m)
    (case m
      ((eval) (inc-num-of-evaluations) eval)
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in "red-mgr": received unknown message " m))))


  ; msg handlers


  (define (eval exp ns-ptr)

    ; evaluate exp in name-space pointed to by ns-ptr.
    ; First obtain the type of the expression, and then dispatch
    ; to the appropriate evaluation routines. Each evaluation rule,
    ; has a corresponding routine.

    (let ((exp-type
   (("exp-mgr" 'get-typ) exp)))
      (case exp-type
((val) (eval-val exp))
((var) (eval-var exp ns-ptr))
((lit) (eval-lit exp))
((lam) (eval-lam exp ns-ptr))
((app) (eval-app exp ns-ptr))
((con) (eval-con exp ns-ptr))
((let) (eval-let exp ns-ptr))
((letrec) (eval-letrec exp ns-ptr))
(else (error "Error in "eval-mgr": received
unknown exp type from "eval-mgr": " exp-type)))))


  (define (init-stats)
    (set! num-of-evals 0)
    (set! num-of-rec-calls 0)
    (set! rec-depth 0)
    (set! max-rec-depth 0))


  (define (get-state)
    (display "number of recursive calls: ")
    (write num-of-rec-calls)
    (newline)
    (display "maximum recursion depth: ")
    (write max-rec-depth)
    (newline)
    (display "number of evaluations: ")
    (write num-of-evals))

  ; evaluation rules


  ; value
     ; return the exp as is.
  (define (eval-val exp)
    exp)


  ; variable reference
     ; if var has an association in the name-space (ns-ptr), return
     ; the value part of the association, else signal an error.
  (define (eval-var exp ns-ptr)
    (let ((assoc (("name-space-mgr" 'lookup) exp ns-ptr)))
      (if
       (equal? assoc 'not-in-name-space)
       (error "Error in "eval-mgr": attempting to evaluate unbound variable" exp)
          ; else it is a bound variable. Return value in association.
       (cdr assoc))))
```

```
; literal
   ; return the literal part of the expression.
(define (eval-lit exp)
  ((*exp-mgr* 'exp-access) 'lit-part exp))


; lambda
   ; return a closure.
(define (eval-lam exp ns-ptr)
  ((*exp-mgr* 'make-closure) exp ns-ptr))


; application
   ; (1) evaluate function-part, and the arguments-part of the
   ;      application expression in the accompanying name-space.
   ; (2) if the function is a primitive, apply the primitive
   ;      and return that as the result of the application.
   ;      Otherwise, the function is a closure, i.e. a pair of
   ;      lambda expression and ns-ptr (a user function
   ;      and a pointer to its lexical name space).
   ; (3) extend the name-space of the closure with bindings of
   ;      variables to argument values.
   ; (4) evaluate the function body in this extended name-space.


(define (eval-app exp ns-ptr)

  (let ((f ((*exp-mgr* 'exp-access) 'fn-part exp))
(args ((*exp-mgr* 'exp-access) 'arg-part exp))
(arg-values '()))

       ; evaluate function part.
     (inc-rec-depth)
     (set! f ((*eval-mgr* 'eval) f ns-ptr))
     (dcr-rec-depth)

       ; evaluate arguments.
     (for-each
      (lambda (arg)
(inc-rec-depth)
(set! arg-values
      (cons ((*eval-mgr* 'eval) arg ns-ptr)
    arg-values))
(dcr-rec-depth))
       args)
       ; Note: arg-values list has arguments in reverse order.

       ; at this point f is either a closure or a primitive function name.
     (if (symbol? f)
    ; If f is a name, it is a primitive. Apply it and return
    ; that as the value of the application expression.
 ((*primitives* 'apply) f arg-values)

    ; else f is a closure.

  (let ((fbody ((*exp-mgr* 'exp-access) 'fbody f))
(fvars ((*exp-mgr* 'exp-access) 'fvars f))
(f-ns-ptr ((*exp-mgr* 'exp-access) 'f-ns-ptr f))
(ext-ns-ptr '()))

       ; extend the function name-space with the bindings of
       ; variables to corresponding argument values.

    (set! ext-ns-ptr
  ((*name-space-mgr* 'extend-ns)
   f-ns-ptr
   fvars
   (reverse arg-values))) ; reverse arg-values list to
                                          ; recover argument order.

       ; evaluate the function body in the extended name-space.
       ; Return that as the value of the application expression.

    ((*eval-mgr* 'eval) fbody ext-ns-ptr)))))


; conditional
   ; (1) evaluate the test-part in the current name-space.
   ; (2) if the result is true value evaluate the consequent part, else
   ;      evaluate the alternate part (in the current name-space).

(define (eval-con exp ns-ptr)

   (define condition #f)
   (inc-rec-depth) ; making a  recursive call
   (set! condition
 ((*eval-mgr* 'eval)
```

```
   ((*exp-mgr* 'exp-access) 'test-part exp)
   ns-ptr))
    (dcr-rec-depth)

    (if condition
   ; evaluate the consequent
((*eval-mgr* 'eval)
 ((*exp-mgr* 'exp-access) 'consequent exp)
 ns-ptr)

   ; evaluate the alternate
((*eval-mgr* 'eval)
 ((*exp-mgr* 'exp-access) 'alternate exp)
 ns-ptr)))


  ; let block
     ; (1) evaluate the inits in the current name-space.
     ; (2) extend the current name-space with the bindings of variables
     ;     to the respective inits.
     ; (3) evaluate the let body in the extended name-space.

  (define (eval-let exp ns-ptr)
    (let ((inits ((*exp-mgr* 'exp-access) 'inits exp))
     ; *exp-mgr* returns a list of individual init expressions
  (vars ((*exp-mgr* 'exp-access) 'let-vars exp))
              ; vars is a list of the let-variables
  (let-body ((*exp-mgr* 'exp-access) 'let-body exp))
     ; let-body is the expression in the let body
  (init-values '())
              ; init-values is a list of evaluated init expressions.
  (ext-ns-ptr '()))
              ; ext-ns-ptr is a pointer to the extended name-space.

         ; evaluate each of the init expressions.
       (for-each
        (lambda (init)
  (inc-rec-depth)
  (set! init-values
         (cons ((*eval-mgr* 'eval) init ns-ptr)
      init-values))
  (dcr-rec-depth))
        inits)
       (set! init-values (reverse init-values)) ; restore init order

          ; extend the current name-space with the bindings of
          ; variables to corresponding init values.

       (set! ext-ns-ptr
     ((*name-space-mgr* 'extend-ns)
      ns-ptr
      vars
      init-values))

          ; evaluate the let body in the extended name-space.
          ; Return that as the value of the let expression.

       ((*eval-mgr* 'eval) let-body ext-ns-ptr)))


  ; letrec block
     ; (1) extend the current name-space with an empty list of
     ;     bindings.
     ; (2) evaluate the init expressions in the extended name-space.
     ; (3) add the bindings of variables with corresponding init values
     ;     to the extended name-space.
     ; (4) evaluate the letrec-body in the extended name-space.

  (define (eval-letrec exp ns-ptr)
    (let ((inits ((*exp-mgr* 'exp-access) 'inits exp))
     ; *exp-mgr* returns a list of individual init expressions
  (vars ((*exp-mgr* 'exp-access) 'let-vars exp))
              ; vars is a list of the letrec variables
  (letrec-body ((*exp-mgr* 'exp-access) 'let-body exp))
     ; letrec-body is the expression in the letrec body
  (init-values '())
              ; init-values is a list of evaluated init expressions.
  (ext-ns-ptr '()))
              ; ext-ns-ptr is a pointer to the extended name-space.

         ; extend name-space with empty lists of names and values
       (set! ext-ns-ptr
     ((*name-space-mgr* 'extend-ns)
      ns-ptr
      '()
      '())))
```

```
        ; evaluate each of the init expressions in the extended name-space.
      (for-each
       (lambda (init)
(inc-rec-depth)
(set! init-values
       (cons ((*eval-mgr* 'eval) init ext-ns-ptr)
     init-values))
(dcr-rec-depth))
        inits)
      (set! init-values (reverse init-values)) ; recover order of inits.

         ; add to the extended name-space the bindings of
         ; variables to corresponding init values.

      ((*name-space-mgr* 'add-list)
       ext-ns-ptr
       vars
       init-values)

         ; evaluate the letrec body in the extended name-space.
         ; Return that as the value of the letrec expression.

      ((*eval-mgr* 'eval) letrec-body ext-ns-ptr)))


  ; end evaluation rules

  ; instrumentation functions and initialisations

  (define num-of-evals 0) ; number of evaluations

  (define inc-num-of-evaluations
     ; inc-num-of-evaluations aborts execution when it receives a "stop"
     ; argument and when num-of-evaluations
     ; exceeds the limit given in the argument list
    (lambda (#!rest x)
      (set! num-of-evals (+ 1 num-of-evals))
      (if (not (null? x))
  (if (and
        (equal? (car x) 'stop)
        (< num-of-evals (cadr x)))
       (error "Abort execution in *eval-mgr*: exceeded maximum number of evaluations," (- num-of-evals 1))))))

  (define rec-depth 0)

  (define max-rec-depth 0)
  (define num-of-rec-calls 0)

  (define (inc-rec-depth)
    (set! num-of-rec-calls (+ 1 num-of-rec-calls))
    (set! rec-depth (+ 1 rec-depth))
    (set! max-rec-depth (max rec-depth max-rec-depth)))

  (define (dcr-rec-depth)
    (set! rec-depth (- rec-depth 1)))

  ; end instrumentation functions and initialisation


  dispatch) ; end make-eval-mgr

; create *eval-mgr* object

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

A.3

```
;;; ************************
;;; ************************
;;; "name-space-mgr" object
;;; accepts: get-global-ns-ptr, add (name, value), lookup (name, ns-ptr),
;;;          add-list (ns-ptr, name-list, value-list),
;;;          extend-ns (ns-ptr, name-list, value-list),
;;;          print-ns (ns-ptr), init-stats, and get-stats
;;; returns: ns-ptr (get-get-global-ns-ptr), binding (lookup),
;;;          name-space (get-name-space)
;;; internal state: name-space


(define (make-name-space-mgr)

  (define (dispatch m)
    (case m
      ((get-global-ns-ptr) global-ns-ptr)
      ((add) add)
      ((lookup) (inc-n-lookups) (init-lookup-depth) lookup)
      ((add-list) add-list)
      ((extend-ns) extend-ns)
      ((print-ns) print-ns) ; print name-space
                ; NOTE: name-space is a circular list whenever
                ;       it contains bindings of names to closures.
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in "name-space-mgr": received unknown message " m))))

  ; msg handlers

  (define (add name value)
       ; add binding to global name-space by adding a binding to the
       ; global frame.
    (set-car!
     global-ns-ptr
     (cons (cons name value) (car global-ns-ptr))))


  (define (lookup name ns-ptr)
       ; start looking for the binding in the top frame.
    (let ((top-frame-ptr (car ns-ptr))
  (binding '#f))
          ; search sequentially through the top-frame
      (set! binding (search-frame name top-frame-ptr))
         ; binding is #f if name is not bound, and its a pair of
         ; name and value if it is bound.
      (if binding
  (begin
    (set! depth-list (cons (cons lookup-depth f-lookup-depth)
   depth-list))
    binding)
     ; binding is #f. Continue looking, up the name-space tree,
     ; until !top has been reached.
  (if
   (equal? (cdr ns-ptr) '!top)
   'not-in-name-space
   (begin
     (inc-f-lookup-depth) ; increment the frame depth of lookup counter
     (lookup name (cdr ns-ptr)))))))

    (define (search-frame name frame)
      (cond ((null? frame) #f)
    ((eq? name (caar frame)) (car frame))
    (else
     (inc-lookup-depth)
     (search-frame name (cdr frame)))))


  (define (add-list ns-ptr name-list value-list)
       ; make the list of bindings formed by name-list and value-list
       ; the top-most frame of the name-space.
    (let ((bindings '()))
          ; make a list of bindings. If name and value lists are empty,
          ; bindings will remain an empty list '().
      (for-each
       (lambda (name value)
  (inc-n-bindings)
  (set! bindings
        (cons (cons name value) bindings)))
        name-list value-list)

          ; make bindings the top-most frame.
      (set-car! ns-ptr bindings)))
```

```
(define (extend-ns ns-ptr name-list value-list)
      ; extend the name-space with a new frame consisting of
      ; a list of bindings made from name-list and value-list.
   (let ((bindings '()))
         ; make a list of bindings. If name and value lists are empty,
         ; bindings will remain an empty list '().
     (for-each
      (lambda (name value)
(inc-n-bindings)
(set! bindings
       (cons (cons name value) bindings)))
       name-list value-list)
         ; extend the name-space with a new frame.
     (inc-n-frames)
     (cons bindings ns-ptr)))


(define (print-ns ns-ptr)
      ; print the name-space by printing one frame at a time.
   (let ((top-frame-ptr (car ns-ptr))
(print-bindings '()) ; a non-circular list of bindings

         ; build print-bindings from the bindings in the top-most frame.
     (for-each

         ; make a safe binding from each individual binding.
       (lambda (binding)
(let ((name (car binding))
       (value (cdr binding)))
  (if (("exp-mgr" 'closure?) value)
          ; value is a closure, which may have a pointer back to
          ; its own name-space, making the name-space a circular
          ; list. In making a printable binding replace the ns-ptr
          ; with 'x-ns-ptr.
      (set! print-bindings
    (cons (cons (cons "name: " name)
   ; make a printable closure
(cons "value: "
       (cons '!closure
    (cons (cadr value) ; lambda
  'x-name-space-ptr))))
  print-bindings))
          ; else value is not a closure.
      (set! print-bindings
    (cons (cons (cons "name: " name)
(cons "value: " value))
  print-bindings)))))

          ; frame from which individual bindings are obtained.
top-frame-ptr)

        ; print a new frame.
     (newline)
     (display "frame is:")
     (newline)
     (if (null? print-bindings)
 (begin
   (display "empty")
   (newline))
 (for-each
  (lambda (print-binding)
    (let ((print-name (car print-binding))
  (print-value (cdr print-binding)))
      (write print-name)
      (newline)
      (write print-value)
      (newline)))
  print-bindings))

     ; stop when top of the name-space is reached.
     (if
      (equal? (cdr ns-ptr) '!top)
      (write '!top)
      (print-ns (cdr ns-ptr)))))


 (define (init-stats)
   (set! n-lookups 0)
   (set! depth-list '())
   (set! max-lookup-depth 0)
   (set! max-f-lookup-depth 0)
   (set! n-bindings 0)
   (set! n-frames 0))

 (define (get-stats)
   (display "lookup stats: tot ")
```

```
    (write n-lookups)
    (let ((d-pair (avg-lookup-depth)))
      (display ", avg depth range ")
      (let ((avg (car d-pair)))
(write-string (string-append
       (string #\()
       (number->string (floor avg))
       (string #\,)
       (number->string (ceiling avg))
       (string #\)))))
      (display ", max depth ")
      (write max-lookup-depth)
      (newline)
      (display "                    avg frame depth range ")
      (let ((avg (cdr d-pair)))
        (write-string (string-append
                       (string #\()
                       (number->string (floor avg))
                       (string #\,)
                       (number->string (ceiling avg))
                       (string #\))))))
    (display ", max frame depth ")
    (write max-f-lookup-depth)
    (newline)
    (display "number of bindings added: ")
    (write n-bindings)
    (newline)
    (display "number of frames added: ")
    (write n-frames))

  ; end message handlers

  ; local state

  (define global-ns-ptr (cons '() '!top))

  ; end local state

  ; instrumentation functions and initialisations

  (define n-lookups 0) ; number of lookups
  (define lookup-depth 0)
  (define f-lookup-depth 0)
  (define max-lookup-depth 0)
  (define max-f-lookup-depth 0)
  (define depth-list '())
  (define n-bindings 0)
  (define n-frames 0)

  (define (inc-n-lookups)
    (set! n-lookups (+ 1 n-lookups)))

  (define (inc-n-bindings)
    (set! n-bindings (+ 1 n-bindings)))

  (define (inc-n-frames)
    (set! n-frames (+ 1 n-frames)))

  (define (init-lookup-depth)
    (set! lookup-depth 0)
    (set! f-lookup-depth 0))
  (define (inc-lookup-depth)
    (set! lookup-depth (+ 1 lookup-depth))
    (set! max-lookup-depth (max lookup-depth max-lookup-depth)))
  (define (inc-f-lookup-depth)
    (set! f-lookup-depth (+ 1 f-lookup-depth))
    (set! max-f-lookup-depth (max f-lookup-depth max-f-lookup-depth)))

  (define (avg-lookup-depth)
    (if (zero? n-lookups)
'(zip . zip)
(let ((b-avg 0)
      (f-avg 0))
  (for-each
   (lambda (depth-pair)
     (set! b-avg (+ b-avg (car depth-pair)))
     (set! f-avg (+ f-avg (cdr depth-pair))))
   depth-list)
  (cons (/ b-avg n-lookups) (/ f-avg n-lookups)))))


  (define (avg-f-lookup-depth)
    (if (zero? n-lookups)
'zip
(/ (reduce + 0 depth-list) n-lookups)))

  ; end instrumentation functions and initialisation
```

```
  dispatch) ; end make-name-space-mgr

; create "name-space-mgr" object
(define "name-space-mgr" (make-name-space-mgr))

; enter primitives into the global name-space
(("primitives" 'enter-primitives))
```

A.4

```
;;; ************************
;;; ************************
;;; "primitives" object
;;; accepts: apply (f args), enter-primitives, get-primitives,
;;;          init-stats, and get-stats
;;; returns: result-expression (apply), table-of-primitives (get-primitives)
;;; internal state: table-of-primitives


(define (make-primitives)

  (define (dispatch m)
    (case m
      ((apply) apply)
      ((enter-primitives) enter-primitives)
      ((get-primitives) table-of-primitives)
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in "primitives": received unknown message " m))))

  ; msg handlers

  (define (apply f args)
    (let ((prim-entry
   (assq f table-of-primitives)))
          ; assq returns #f if it did not find an association
      (inc-n-p-apply-ops)
      (if prim-entry
     ; f is a primitive. Apply it to the args.
  ((cdr prim-entry) args)
     ; f is not a primitive. Signal error.
  (error "Error in "primitives": attempting to
apply non-existent primitive" f))))

  (define (enter-primitives)
       ; enter the primitives into the global name-space.
       ; NOTE: this means that primitive names are treated like
       ;       any other names, i.e. they can be used as variable
       ;       names.
    (for-each
     (lambda (entry)
       (("name-space-mgr" 'add) (car entry) (car entry)))
      table-of-primitives))


  (define (init-stats)
    (set! n-p-apply-ops 0)
    (set! n-cons-ops 0))

  (define (get-stats)
    (display "number of primitive applies: ")
    (write n-p-apply-ops)
    (newline)
    (display "number of cons operations: ")
    (write n-cons-ops))

  ; end message handlers

  ; local state

     ; NOTE: arguments passed to primitives are in reverse order.

  (define table-of-primitives
    (list
     (cons 'symbol? (lambda (arg) (symbol? (car arg))))
     (cons 'boolean? (lambda (arg) (boolean? (car arg))))
     (cons 'pair? (lambda (arg) (pair? (car arg))))
     (cons 'number? (lambda (arg) (integer? (car arg))))
        ; arg graph is a function if it is a primitive name,
        ; or if it is a lambda graph.
     (cons 'function? (lambda (arg)
(or
 (primitive? (car arg))
 (("exp-mgr" 'closure?) (car arg)))))
     (cons 'eq? (lambda (arg)
  (eqv? (cadr arg) (car arg))))
     (cons 'null? (lambda (arg)
    (null? (car arg))))
     (cons 'not (lambda (arg)
   (not (car arg))))
     (cons 'cons (lambda (arg)
    (inc-n-cons-ops)
    (cons (cadr arg) (car arg))))
     (cons 'car (lambda (arg)
```

```
  (car (car arg))))
     (cons 'cdr (lambda (arg)
  (cdr (car arg))))
     (cons '+ (lambda (arg)
(+ (cadr arg) (car arg))))
     (cons '- (lambda (arg)
(- (cadr arg) (car arg))))
     (cons '* (lambda (arg)
(* (cadr arg) (car arg))))
     (cons 'quotient (lambda (arg)
(quotient (cadr arg) (car arg))))
     (cons 'remainder (lambda (arg)
(remainder (cadr arg) (car arg))))
     (cons '= (lambda (arg)
(= (cadr arg) (car arg))))
     (cons '¡ (lambda (arg)
(¡ (cadr arg) (car arg))))
     (cons '¡= (lambda (arg)
(¡= (cadr arg) (car arg))))
     ))


  ; end local state

  ; instrumentation functions and initialisations

  (define n-p-apply-ops 0)
  (define n-cons-ops 0)
  (define (inc-n-p-apply-ops)
    (set! n-p-apply-ops (+ 1 n-p-apply-ops)))
  (define (inc-n-cons-ops)
    (set! n-cons-ops (+ 1 n-cons-ops)))

  ; end instrumentation functions and initialisations

  dispatch) ; end make-primitives


; create "primitives" object

(define *primitives* (make-primitives))
```

A.5

```
;;; ************************
;;; ************************
;;; "exp-mgr" object
;;; All expression manipulation is handled by the "exp-mgr" object.
;;; Expressions themselves, however, are not local to the "exp-mgr":
;;; whoever has an expression pointer (exp) can manipulate the expression.
;;; accepts: exp-access (access-typ, exp), get-typ (exp),
;;;          make-closure (exp, ns-ptr), closure? (exp),
;;;          define?(exp), get-stats, init-stats
;;; returns: exp (exp-access), exp-type (get-typ), closure (make-closure),
;;;          #t or #f (closure?, define?)
;;; internal state: expression manipulation statistics


(define (make-exp-mgr)

  (define (dispatch m)
    (case m
      ((exp-access) (inc-n-e-accesses) exp-access)
      ((get-typ) (inc-n-e-type-ops) get-typ)
      ((make-closure) make-closure)
      ((closure?) closure?)
      ((define?) define?)
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in "exp-mgr": received unknown message " m))))


  ; msg handlers

  (define (exp-access access-typ exp)
      ; invoke the appropriate expression-access procedure. The
      ; value of the variable "access-typ" is the name of the
      ; procedure to be invoked.
    ((eval access-typ (the-environment)) exp))
        ; Warning: this works in MIT Scheme, and I suspect is more
        ;          efficient than a dispatch using a case. However,
        ;          Scheme itself does not
        ;          support environments as first-class objects.
        ;          For compatibility with Scheme as defined
        ;          in the Revised^4 Report, this can be recoded as
        ;          a case on access-typ:
        ;    (case access-typ
        ;      ((value) (value exp))
        ;      ((lit-part) (lit-part exp)
        ;           ....
        ;      (else (error "Wrong expression access type"))))



  (define (get-typ exp)
    (cond

        ; val: expression is either a boolean or an integer
      ((or (boolean? exp)
    (integer? exp))
       'val)

        ; var: expression is a variable
      ((symbol? exp) 'var)

        ; check for invalid expression
      ((and (not (pair? exp)) (not (symbol? (car exp))))
       (error "Error in "exp-mgr": received get-typ request for " exp))

        ; lit: exp form is (quote literal-part)
      ((equal? (car exp) 'quote) 'lit)

        ; lam: exp form is (lambda (var1 ... varN) body)
      ((equal? (car exp) 'lambda) 'lam)

        ; let: expression form is (let ((var1 init1) ... (varN initN)) body)
      ((equal? (car exp) 'let) 'let)

        ; similarly for letrec
      ((equal? (car exp) 'letrec) 'letrec)

        ; con: expression form is (if test-part consequent alternate)
      ((equal? (car exp) 'if) 'con)

        ; app: else exp is an application
      (else 'app)))
```

```
(define (make-closure lambda-exp ns-ptr)
      ; a closure is a list of !closure, lambda-exp and ns-ptr.
  (inc-n-cons) (inc-n-cons)
  (cons '!closure (cons lambda-exp ns-ptr)))

(define (closure? pair)
  (and
   (pair? pair)
   (equal? (car pair) '!closure)))

(define (define? exp)
  (if (pair? exp)
      (equal? (car exp) 'define)
      '()))

(define (init-stats)
  (set! n-e-accesses 0)
  (set! n-e-type-ops 0)
  (set! n-cons 0)
  (set! access-depth-list '()))

(define (get-stats)
  (display "expression access stats: tot ")
  (write n-e-accesses)
  (display ", avg depth range ")
  (let ((avg 0) (maxd 0))
    (for-each
     (lambda (depth)
(set! avg (+ depth avg))
(set! maxd (max maxd depth)))
      access-depth-list)
     (set! avg (/ avg (length access-depth-list)))
     (write-string (string-append
     (string #()
     (number-¿string (floor avg))
     (string #,)
     (number-¿string (ceiling avg))
     (string #))))
      (newline) (display "                                ")
      (display "max depth ")
      (write maxd))
   (newline)
   (display "number of type-ops: ")
   (write n-e-type-ops)
   (newline)
   (display "number of cons cells used: ")
   (write n-cons))

 ; end message handlers


 ; expression access routines

    ; value part: expression form is (define name value-part)
 (define (value-part exp)
   (if (define? exp)
(begin
 (cons-access-depth-list 3)
 (caddr exp)) ; extract value part of expression
(begin
 (cons-access-depth-list 0)
 exp)))

 (define (name-in exp)
   (cons-access-depth-list 2)
   (cadr exp))

    ; literal part: expression form is (quote literal-part)
 (define (lit-part exp)
   (cons-access-depth-list 2)
   (cadr exp))

    ; let: expression form is (let ((var1 init1) ... (varN initN)) body)
 (define (inits exp)
   (let ((inits '())
 (bindings (let-bindings exp))
 (tot-access-depth 2))
        ; for each binding in the list of bindings extract the
        ; init expression and add to the inits expression list.
        ; Return inits.
     (for-each
      (lambda (binding)
(set! tot-access-depth (+ 2 tot-access-depth))
(inc-n-cons)
(set! inits (cons (cadr binding) inits)))
       bindings)
     (cons-access-depth-list tot-access-depth)
```

```
      inits)) ; return inits

  (define (let-vars exp)
    (let ((vars '())
  (bindings (let-bindings exp))
           (tot-access-depth 2))
          ; for each binding in the list of bindings extract the
          ; var part and add to the vars list. Return vars list.
      (for-each
       (lambda (binding)
         (set! tot-access-depth (+ 2 tot-access-depth))
         (inc-n-cons)
 (set! vars (cons (car binding) vars)))
        bindings)
      (cons-access-depth-list tot-access-depth)
      vars)) ; return vars

  (define (let-body exp)
    (cons-access-depth-list 3)
    (caddr exp))

  (define (let-bindings exp)
    (cons-access-depth-list 2)
    (cadr exp))

     ; conditional: expression form is (if test-part consequent alternate)
  (define (test-part exp)
    (cons-access-depth-list 2)
    (cadr exp))

  (define (consequent exp)
    (cons-access-depth-list 3)
    (caddr exp))

  (define (alternate exp)
    (cons-access-depth-list 4)
    (cadddr exp))

     ; application: expression form is (fn arg1 ... argN)
  (define (fn-part exp)
    (cons-access-depth-list 1)
    (car exp))

  (define (arg-part exp)
    (cons-access-depth-list 1)
    (cdr exp))

     ; by the time requests for fbody and fvars are received
     ; f is a closure, which has the form:
     ; (!closure . ((lambda (var1 ... varN) body) . ns-ptr))

  (define (fbody f)
    (cons-access-depth-list 4)
    (caddar (cdr f))) ; body

  (define (fvars f)
    (cons-access-depth-list 3)
    (cadar (cdr f))) ; (var1 ... varN)

  (define (f-ns-ptr f)
    (cons-access-depth-list 2)
    (cddr f))

  ; end expression exp-access routines


  ; instrumentation functions and initialisations

  (define n-e-accesses 0)
  (define n-e-type-ops 0)
  (define n-cons 0)
  (define access-depth-list '())

  (define (inc-n-e-accesses)
    (set! n-e-accesses (+ 1 n-e-accesses)))
  (define (inc-n-e-type-ops)
    (set! n-e-type-ops (+ 1 n-e-type-ops)))
  (define (inc-n-cons)
    (set! n-cons (+ 1 n-cons)))

  (define (cons-access-depth-list depth)
    (set! access-depth-list (cons depth access-depth-list)))

  ; end instrumentation functions and initialisation

  dispatch) ; end make-exp-mgr
```

```
; create "exp-mgr" object
(define "exp-mgr" (make-exp-mgr))
```

Appendix B

Reduction Evaluator Code

This appendix contains the Reduction Evaluator code consisting of the `*evaluator*`, `*red-mgr*`, `*name-space-mgr*`, `*primitives*`, and `*graph-mgr*` objects.

B.1

```
;;; ************************
;;; ************************
;;; *evaluator* object
;;; accepts: eval (fScheme expression), init-stats, and get-stats
;;; returns: value of the expression
;;; internal state: evaluation statistics


(define (make-evaluator)

  (define (dispatch m)
    (case m
      ((eval) eval)
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in *evaluator*: received unknown message " m))))

  ; msg handlers

  (define (eval exp)

      ; initilise the statistics
  ((*red-mgr* 'init-stats))
  ((*name-space-mgr* 'init-stats))
  ((*graph-mgr* 'init-stats))
  ((*primitives* 'init-stats))
     ; reduce the expression
    (let ((gr-ptr
       ; first translate exp to graph, and let gr-ptr be
       ; the pointer to that graph
     ((*graph-mgr* 'copy)  ; need a fresh copy so as not to
                                        ; reduce (mangle) the incoming expression.
      ((*graph-mgr* 'translate) (value-part exp)))))
          ; reduce the graph
        ((*red-mgr* 'red) gr-ptr)
           ; check if gr-ptr has been reduced. Signal error if not.
        (if
         (not (equal? 'reduced ((*graph-mgr* 'get-typ) gr-ptr)))
      (error "Error: *evaluator* received an unreduced
graph from *red-mgr*" gr-ptr))
        (if (define? exp)
       ; if define expression send message to *name-space-mgr*
       ; to enter the reduce graph into the name space. Entries
       ; in the name space are (name, gr-ptr).
    (begin
      ((*name-space-mgr* 'add) (name-in exp) gr-ptr)
      (newline)
      (display "added variable ")
      (write (name-in exp))
      (display " to the name-space"))
       ; else it is not a define expression, so simply return the value
       ; in the reduced graph.
    (begin
      (newline)
      (display "Value: ")
      (write ((*graph-mgr* 'gr-access) 'value gr-ptr))
      (newline) (newline)
      (display "*red-mgr statistics: ")
      (newline)
      ((*red-mgr* 'get-stats))
               (newline) (newline)
               (display "*name-space-mgr* statistics: ")
               (newline)
               ((*name-space-mgr* 'get-stats))
      (newline) (newline)
               (display "*primitives* statistics: ")
               (newline)
               ((*primitives* 'get-stats))
               (newline) (newline)
      (display "*graph-mgr statistics: ")
      (newline)
      ((*graph-mgr* 'get-stats))))))

  (define (init-stats)
    (newline)
    (display "doing init-stats"))

  (define (get-stats)
    (newline)
    (display "doing get-stats"))

  ; auxillary functions

  (define (value-part exp)
```

```
    (if (define? exp)
(caddr exp) ; extract value part of expression
exp))

  (define (name-in exp)
    (cadr exp))   ;  what defines are allowed?

  (define (define? exp)
    (if (pair? exp)
(equal? (car exp) 'define)
'()))

  ; end auxillary functions


  dispatch) ; end make-evaluator


; create "evaluator" object

(define "evaluator" (make-evaluator))
```

B.2

```
;;; ************************
;;; ************************
;;; *red-mgr* object
;;; accepts: red (gr-ptr), init-stats, and get-stats
;;; red: reduce graph gr-ptr


(define (make-red-mgr)

  (define (dispatch m)
    (case m
      ((red) (inc-num-of-red) red) ; increment reduction counter
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in *red-mgr*: received unknown message " m))))


  ; msg handlers


  (define (red gr-ptr)

    ; reduce graph (pointed to by gr-ptr).
    ; First obtain the type of the graph, and then dispatch
    ; to the appropriate reduction routines. Each reduction rule,
    ; has a corresponding reduction routine.

    (let ((graph-type
   ((*graph-mgr* 'get-typ) gr-ptr)))
      (case graph-type
((reduced) '()) ; in reduced form already
((val) (red-val gr-ptr))
((var) (red-var gr-ptr))
((lit) (red-lit gr-ptr))
((lam) (red-lam gr-ptr))
((app) (red-app gr-ptr))
((con) (red-con gr-ptr))
((let) (red-let gr-ptr))
((letrec) (red-letrec gr-ptr))
(else (error "Error in *red-mgr*: received unknown graph
type from *graph-mgr*: " graph-type)))))


  (define (init-stats)
    (set! num-of-red 0)
    (set! num-of-rec-calls 0)
    (set! rec-depth 0)
    (set! max-rec-depth 0))


  (define (get-stats)
    (display "number of recursive calls: ")
    (write num-of-rec-calls)
    (newline)
    (display "maximum recursion depth: ")
    (write max-rec-depth)
    (newline)
    (display "number of reductions: ")
    (write num-of-red))


  ; reduction rules

  ; value
     ; simply mark! the value graph, to indicate that it has been reduced.

  (define (red-val gr-ptr)
    ((*graph-mgr* 'mark!) gr-ptr))


  ; variable reference
     ; if var has an association in the name-space, rewrite! with the
     ; corresponding graph, else signal an error.

  (define (red-var gr-ptr)
    (let ((var-name ((*graph-mgr* 'gr-access) 'var-in gr-ptr)))
      (let ((assoc ((*name-space-mgr* 'lookup) var-name)))
(if
 (equal? assoc 'not-in-name-space)
    ; var-name is not a bound variable. Signal error.
 (else (error "Error in *red-mgr*: attempting to reduce
unbound variable" var-name))
    ; else it is a bound variable. Rewrite! with the graph (assoc).
```

```
((*graph-mgr* 'rewrite!) gr-ptr assoc)))))


 ; literal
    ; rewrite and mark! literal graph with the subgraph containing the
    ; literal-part.

 (define (red-lit gr-ptr)
   ((*graph-mgr* 'rewrite-and-mark!)
    gr-ptr
    ((*graph-mgr* 'gr-access) 'lit-part gr-ptr)))


 ; lambda
    ; simply mark! the lambda graph as reduced.

 (define (red-lam gr-ptr)
   ((*graph-mgr* 'mark!) gr-ptr))


 ; application
    ; 1) reduce function graph.
    ; 2) reduce argument graphs.
    ; 3) if the function graph is a primitive, apply the primitive
    ;    and rewrite and mark! the application graph, then exit.
    ; 4) else the function graph is a lambda graph,
    ;    i.e. a user function, so copy the function body.
    ; 5) replace the variables in the function-body graph with the
    ;    corresponding argument graphs.
    ; 6) rewrite! the application graph with this new function body
    ;    graph.
    ; 7) reduce this new version of the application graph.

 (define (red-app gr-ptr)
   (let ((f ((*graph-mgr* 'gr-access) 'fn-part gr-ptr))
  (arg-gr-list (make-gr-list
 ((*graph-mgr* 'gr-access) 'arg-part gr-ptr)))
  (args ((*graph-mgr* 'gr-access) 'arg-part gr-ptr)))

        ; reduce function f
     (inc-rec-depth)
     ((*red-mgr* 'red) f)
     (dcr-rec-depth)

        ; reduce arguments
     (for-each
      (lambda (arg-gr-ptr)
 (inc-rec-depth)
 ((*red-mgr* 'red) arg-gr-ptr)
 (dcr-rec-depth))
      arg-gr-list)

        ; at this point f is a lambda graph or a primitive function name.
     (if (symbol? ((*graph-mgr* 'gr-access) 'value f))
      ; If f is a name apply the primitive.
  ((*graph-mgr* 'rewrite-and-mark!)
   gr-ptr
   ((*primitives* 'apply)
      ((*graph-mgr* 'gr-access) 'value f) ; get primitive name
      args))
     ; else f is a lambda.
  (let ((fbody ((*graph-mgr* 'gr-access) 'fbody f)))
    (let ((new-fbody ; copy function body
   ((*graph-mgr* 'copy)
    fbody))
  (vars ((*graph-mgr* 'gr-access) 'fvars f)))

        ; replace vars in the copy of the body
     (for-each
      (lambda (var arg-gr-ptr)
 ((*graph-mgr* 'find-repl) new-fbody var arg-gr-ptr))
      vars arg-gr-list)

        ; rewrite! the graph with this new graph
     ((*graph-mgr* 'rewrite!) gr-ptr new-fbody)
        ; reduce the new version of the graph

     ((*red-mgr* 'red) gr-ptr))))))


  (define (make-gr-list linked-grs)

    ; convert a linked list of graphs into a list of graphs

    (if (null? linked-grs)
'()          ; in case of empty gr-list return empty list.
(cons linked-grs (make-gr-list (cdr linked-grs)))))
```

```
; conditional
    ; rewrite! the if graph with either the result of reducing the
    ; consequent part, or the result of reducing the alternate
    ; part, depending on whether the test part reduced to a true value
    ; or not.

(define (red-con gr-ptr)

      ; reduce the test-part.
   (inc-rec-depth)
   (("red-mgr" 'red)
    (("graph-mgr" 'gr-access) 'test-part gr-ptr))
   (dcr-rec-depth)

      ; reduce either the consequent or alternate parts.
   (let ((branch-gr-ptr ; depends on result of test-part reduction
  (if (("graph-mgr" 'gr-access)
'value
(("graph-mgr" 'gr-access) 'test-part gr-ptr))
       (("graph-mgr" 'gr-access) 'consequent gr-ptr)
       (("graph-mgr" 'gr-access) 'alternate gr-ptr))))
      (("graph-mgr" 'rewrite!) gr-ptr branch-gr-ptr))
   (("red-mgr" 'red) gr-ptr))


; let block
    ; 1) reduce the init graphs.
    ; 2) replace the variables in the let body with the corresponding
    ;    init graphs.
    ; 3) rewrite! the let graph with this changed let-body graph.
    ; 4) reduce this new version of the let graph.

(define (red-let gr-ptr)
  (let ((inits (("graph-mgr" 'gr-access) 'inits gr-ptr))
   ; "graph-mgr" returns a list of individual init graphs
(vars (("graph-mgr" 'gr-access) 'let-vars gr-ptr))
            ; vars is a list of the let-variables
(let-body (("graph-mgr" 'gr-access) 'let-body gr-ptr)))
    ; let-body is the graph of the let body

        ; first reduce each of the inits
     (for-each
      (lambda (init-gr-ptr)
(inc-rec-depth)
(("red-mgr" 'red) init-gr-ptr)
(dcr-rec-depth))
       inits)
        ; replace vars with the corresponding init graphs in the let-body
     (for-each
      (lambda (var init-gr-ptr)
(("graph-mgr" 'find-repl) let-body var init-gr-ptr))
       vars inits)
        ; rewrite the let graph with the let body graph
     (("graph-mgr" 'rewrite!) gr-ptr let-body)
        ; reduce this new graph
     (("red-mgr" 'red) gr-ptr)))


; letrec block
    ; 1) reduce the init graphs.
    ; 2) replace each letrec-variable with the corresponding init
    ;    in each of the init graphs. (this is the only difference
    ;    between the let and the letrec reduction rules)
    ; 3) replace the variables in the letrec body with the corresponding
    ;    init graphs.
    ; 4) rewrite! the letrec graph with this changed letrec-body graph.
    ; 5) reduce this new version of the letrec graph.

(define (red-letrec gr-ptr)
  (let ((inits (("graph-mgr" 'gr-access) 'inits gr-ptr))
(vars (("graph-mgr" 'gr-access) 'let-vars gr-ptr))
(letrec-body (("graph-mgr" 'gr-access) 'let-body gr-ptr)))
        ; reduce each of the init graphs
     (for-each
      (lambda (init-gr-ptr)
(inc-rec-depth)
(("red-mgr" 'red) init-gr-ptr)
(dcr-rec-depth))
       inits)

        ; replace each variable with the corresponding init
        ; in each of the init graphs
     (for-each
      (lambda (gr-ptr)
(for-each
```

```
  (lambda (var init-gr-ptr)
    (("graph-mgr" 'find-repl) (cons (cdar gr-ptr) '())
      var init-gr-ptr))
  vars inits))
       inits)

          ; replace vars with the corresponding init graphs in the letrec-body
      (for-each
       (lambda (var init-gr-ptr)
 (("graph-mgr" 'find-repl) letrec-body var init-gr-ptr))
       vars inits)
          ; rewrite the letrec graph with the letrec-body graph
      (("graph-mgr" 'rewrite!) gr-ptr letrec-body)
          ; reduce this new graph
      (("red-mgr" 'red) gr-ptr)))

  ; end reduction rules

  ; instrumentation functions and initialisations

  (define num-of-red 0) ; number of reductions

  (define inc-num-of-red
     ; inc-num-of-red aborts execution when it receives a "stop"
     ; argument and when num-of-reductions
     ; exceeds the limit given in the argument list
    (lambda (#!rest x)
      (set! num-of-red (+ 1 num-of-red))
      (if (not (null? x))
  (if (and
       (equal? (car x) 'stop)
       (¿ num-of-red (cadr x)))
      (error "Abort execution in "red-mgr": exceeded maximum number of reductions," (- num-of-red 1))))))

  (define rec-depth 0)

  (define max-rec-depth 0)
  (define num-of-rec-calls 0)
  (define (inc-rec-depth)
    (set! num-of-rec-calls (+ 1 num-of-rec-calls))
    (set! rec-depth (+ 1 rec-depth))
    (set! max-rec-depth (max rec-depth max-rec-depth)))

  (define (dcr-rec-depth)
    (set! rec-depth (- rec-depth 1)))

  ; end instrumentation functions and initialisation

  dispatch) ; end make-red-mgr

; create "red-mgr" object

(define "red-mgr" (make-red-mgr))
```

B.3

```
;;; ************************
;;; ************************
;;; "name-space-mgr" object
;;; accepts: add (name gr-ptr), lookup (name), init-stats, and get-stats
;;; returns: gr-ptr (lookup)
;;; internal state: name-space


(define (make-name-space-mgr)

  (define (dispatch m)
    (case m
      ((add) add)
      ((lookup) (inc-n-lookups) (init-lookup-depth) lookup)
      ((get-name-space) name-space) ; return current value of name-space
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in "name-space-mgr": received unknown message " m))))

  ; msg handlers

  (define (add name gr-ptr)
    (set! name-space
  (cons (cons name gr-ptr) name-space)))


  (define (lookup name)
    (let ((binding '#f))
          ; binding is #f if name is not bound, and its a pair of
          ; name and gr-ptr if its bound.
      (set! binding (search name name-space))
      (if binding
  (begin
            (set! depth-list (cons lookup-depth depth-list))
    (cdr binding))
  'not-in-name-space)))

    (define (search name bindings-list)
      (cond ((null? bindings-list) #f)
            ((eq? name (caar bindings-list)) (car bindings-list))
            (else
             (inc-lookup-depth)
             (search name (cdr bindings-list)))))

  (define (init-stats)
    (set! n-lookups 0)
    (set! depth-list '())
    (set! max-lookup-depth 0))

  (define (get-stats)
    (display "lookup stats: tot ")
    (write n-lookups)
    (display ", avg depth range ")
    (let ((avg (avg-lookup-depth)))
      (write-string (string-append
     (string #()
     (number-¿string (floor avg))
     (string #,)
     (number-¿string (ceiling avg))
     (string #)))))
    (display ", max depth ")
    (write max-lookup-depth))


  ; end message handlers

  ; local state

  (define name-space '())

  ; end local state

  ; instrumentation functions and initialisations

  (define n-lookups 0) ; number of lookups
  (define lookup-depth 0)
  (define depth-list '())
  (define max-lookup-depth 0)

  (define (inc-n-lookups)
    (set! n-lookups (+ 1 n-lookups)))

  (define (init-lookup-depth)
    (set! lookup-depth 0))
```

```
  (define (inc-lookup-depth)
    (set! lookup-depth (+ 1 lookup-depth))
    (set! max-lookup-depth (max lookup-depth max-lookup-depth)))

  (define (avg-lookup-depth)
    (if (zero? n-lookups)
        'zip
(/ (reduce + 0 depth-list) n-lookups)))

  ; end instrumentation functions and initialisation


  dispatch) ; end make-name-space-mgr

; create "name-space-mgr" object

(define "name-space-mgr" (make-name-space-mgr))

; enter primitives into the name-space

(("primitives" 'enter-primitives))
```

B.4

```
;;; ************************
;;; ************************
;;; *primitives* object
;;; accepts: apply (f args), enter-primitives, get-primitives,
;;;          init-stats, and get-stats
;;; returns: result-graph (apply),
;;;          table-of-primitives (get-primitives)
;;; internal state: table-of-primitives


(define (make-primitives)

  (define (dispatch m)
    (case m
      ((apply) apply)
      ((enter-primitives) enter-primitives)
      ((get-primitives) table-of-primitives)
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in *primitives*: received unknown message " m))))

  ; msg handlers

  (define (apply f args)
    (let ((prim-entry
   (assq f table-of-primitives)))
          ; assq returns #f if it did not find an association
      (inc-n-p-apply-ops)
      (if prim-entry
     ; f is a primitive. Apply it to the args.
  ((*graph-mgr* 'translate)
    (let ((arg-values '()))
         ; arg-values is a list of arguments without the !r nodes.
         ; NOTE: arguments in arg-values list appear in reverse order.
      (for-each
       (lambda (arg-gr)
(set! arg-values (cons (cdr arg-gr) arg-values)))
       args)
         ; apply the primitive to the list of arguments.
      ((cdr prim-entry) arg-values)))
     ; f is not a primitive. Signal error.
   (error "Error in *primitives*: attempting to apply
non-existent primitive" f))))

  (define (enter-primitives)
       ; enter the primitives into the name-space.
       ; NOTE: this means that primitive names are treated like
       ;       any other names, i.e. they can be used as variable
       ;       names.
    (for-each
     (lambda (entry)
       ((*name-space-mgr* 'add) (car entry)
(cons (cons '!r (car entry)) '())))
      table-of-primitives))


  (define (init-stats)
    (set! n-p-apply-ops 0)
    (set! n-cons-ops 0))

  (define (get-stats)
    (display "number of primitive applies: ")
    (write n-p-apply-ops)
    (newline)
    (display "number of cons operations: ")
    (write n-cons-ops))

  ; end message handlers

  ; local state

     ; NOTE: arguments passed to primitives are in reverse order.

  (define table-of-primitives
    (list
     (cons 'symbol? (lambda (arg) (symbol? (car arg))))
     (cons 'boolean? (lambda (arg) (boolean? (car arg))))
     (cons 'pair? (lambda (arg) (pair? (car arg))))
     (cons 'number? (lambda (arg) (integer? (car arg))))
        ; arg graph is a function if it is a primitive name,
        ; or if it is a lambda graph.
     (cons 'function? (lambda (arg)
(or
 (primitive? (car arg))
```

```
 (equal? ((*graph-mgr* 'get-typ) arg)
 'lam))))
      (cons 'eq? (lambda (arg)
  (eqv? (cadr arg) (car arg))))
      (cons 'null? (lambda (arg)
     (null? (car arg))))
      (cons 'not (lambda (arg)
   (not (car arg))))
      (cons 'cons (lambda (arg)
                    (inc-n-cons-ops)
   (cons (cadr arg) (car arg))))
      (cons 'car (lambda (arg)
  (car (car arg))))
      (cons 'cdr (lambda (arg)
  (cdr (car arg))))
      (cons '+ (lambda (arg)
(+ (cadr arg) (car arg))))
      (cons '- (lambda (arg)
(- (cadr arg) (car arg))))
      (cons '* (lambda (arg)
(* (cadr arg) (car arg))))
      (cons 'quotient (lambda (arg)
(quotient (cadr arg) (car arg))))
      (cons 'remainder (lambda (arg)
(remainder (cadr arg) (car arg))))
      (cons '= (lambda (arg)
(= (cadr arg) (car arg))))
      (cons '< (lambda (arg)
(< (cadr arg) (car arg))))
      (cons '<= (lambda (arg)
(<= (cadr arg) (car arg))))
      ))


  ; end local state


  dispatch) ; end make-primitives


  ; instrumentation functions and initialisations

  (define n-p-apply-ops 0)
  (define n-cons-ops 0)
  (define (inc-n-p-apply-ops)
    (set! n-p-apply-ops (+ 1 n-p-apply-ops)))
  (define (inc-n-cons-ops)
    (set! n-cons-ops (+ 1 n-cons-ops)))

  ; end instrumentation functions and initialisations

; create "primitives" object

(define *primitives* (make-primitives))
```

B.5

```
;;; ************************
;;; ************************
;;; "graph-mgr" object
;;; All graph manipulation is handled by the "graph-mgr" object. Graphs
;;; themselves, however, are not local to the "graph-mgr": whoever has
;;; a graph pointer (gr-ptr) can manipulate the graph.
;;; accepts: translate (exp), gr-access (access-typ, gr-ptr), copy (gr-ptr),
;;;          rewrite! (gr-ptr, gr-ptr), mark! (gr-ptr),
;;;          rewrite-and-mark! (gr-ptr, gr-ptr),
;;;          get-typ (gr-ptr), find-repl (gr-ptr, var, gr-ptr), get-stats,
;;;          init-stats
;;; returns: gr-ptr except: name (gr-access var-in), value (gr-access value),
;;;          graph-typ (get-typ), and 'do-nothing (find-repl in some cases)
;;; internal state: graph manipulation statistics


(define (make-graph-mgr)

  (define (dispatch m)
    (case m
      ((translate) translate)
      ((gr-access) (inc-n-g-accesses) gr-access)
      ((copy) (inc-n-copy-ops) copy)
      ((rewrite!) rewrite!)
      ((mark!) mark!)
      ((rewrite-and-mark!) rewrite-and-mark!)
      ((get-typ) get-typ)
      ((find-repl) find-repl)
      ((init-stats) init-stats)
      ((get-stats) get-stats)
      (else (error "Error in "graph-mgr": received unknown message " m))))

  ; msg handlers

  (define (translate exp)
    ; a graph is an exp plus a cons cell
    (inc-n-translates)
    (inc-n-nodes-used)
    (cons exp '()))

  (define (gr-access access-typ gr-ptr)
      ; invoke the appropriate graph-access procedure. The
      ; value of the variable "access-typ" is the name of the
      ; procedure to be invoked.
    ((eval access-typ (the-environment)) gr-ptr))
        ; Warning: this works in MIT Scheme, and I suspect is more
        ;          efficient than a dispatch using a case. However,
        ;          Scheme itself does not
        ;          support environments as first-class objects.
        ;          For compatibility with Scheme as defined
        ;          in the Revised'4 Report, this can be recoded as
        ;          a case on access-typ:
        ;    (case access-typ
        ;      ((value) (value gr-ptr))
        ;      ((lit-part) (lit-part gr-ptr)
        ;                  ....
        ;      (else (error "Wrong graph access type"))))

  (define (copy ptr)
      ; copy graph or subgraph by obtaining fresh nodes for all
      ; the unreduced parts, while reusing all the reduced parts.
    (if (pair? ptr)
(if (equal? (car ptr) '!r)
      ; reuse the reduced graph
    ptr
      ; else create a new node
    (begin
      (inc-n-nodes-used)
      (add2-n-copy-ops)
      (cons (copy (car ptr))
    (copy (cdr ptr)))))
   ; ptr is a value, simply return it
ptr))

  (define (rewrite! to-gr-ptr from-gr-ptr)
    (set-car! to-gr-ptr (car from-gr-ptr)))

  (define (mark! gr-ptr)
    (inc-n-nodes-used)
```

```
  (set-car! gr-ptr (cons '!r (car gr-ptr))))

(define (rewrite-and-mark! to-gr-ptr from-gr-ptr)
  (inc-n-nodes-used)
  (set-car! to-gr-ptr (car from-gr-ptr))
  (set-car! to-gr-ptr (cons '!r (car to-gr-ptr))))

(define (get-typ gr-ptr)
  (cond

      ; trap non-graphs
   ((not (pair? gr-ptr))
    (error "Error in "graph-mgr": received get-typ request for " gr-ptr))
   ((null? gr-ptr)
    (error "Error in "graph-mgr": received get-typ request for " gr-ptr))

      ; val: graph form is (boolean . ") or (integer . ")
   ((or (boolean? (car gr-ptr))
(integer? (car gr-ptr)))
    'val)

      ; var: graph form is (variable . ")
   ((and (not (equal? (car gr-ptr) '!r))
 (symbol? (car gr-ptr))) 'var)

      ; check for invalid graph of the form (invalid-object . ")
   ((not (pair? (car gr-ptr)))
    (error "Error in "graph-mgr": received get-typ request for " gr-ptr))

      ; reduced: graph form is ((!r . "))
   ((equal? (caar gr-ptr) '!r)
    'reduced)

      ; lit: graph form is ((quote literal-part) . ")
   ((equal? (caar gr-ptr) 'quote) 'lit)

      ; lam: graph form is ((lambda (var1 ... varN) body) . ")
   ((equal? (caar gr-ptr) 'lambda) 'lam)

      ; let: graph form is ((let ((var1 init1) ... (varN initN)) body) . ")
   ((equal? (caar gr-ptr) 'let) 'let)

      ; similarly for letrec
   ((equal? (caar gr-ptr) 'letrec) 'letrec)

     ; con: graph form is ((if test-part consequent alternate) . ")
   ((equal? (caar gr-ptr) 'if) 'con)

     ; app: else the graph must be an application
   (else 'app)))

(define (find-repl gr-ptr var arg-gr-ptr)
     ; find each var-type subgraph in gr-ptr that contains a free
     ; occurrence of var and replace it with arg-gr-ptr.
  (init-search)
  (search-repl gr-ptr var arg-gr-ptr)
  (set! search-list (cons (cons search-size max-search-depth) search-list))
  (set-2max-search-depth))


(define (init-stats)
  (set! n-var-replaces 0)
  (set! max-search-depth 0)
  (set! 2max-search-depth 0)
  (set! max-search-size 0)
  (set! search-list '())
  (set! n-copy-ops 0)
  (set! n-translates 0)
  (set! n-nodes-used 0))

(define (get-stats)
  (display "variable find-and-replace stats: tot number of replaces is ")
  (write n-var-replaces)
  (newline)
  (display "          tot number of searches is ")
  (write (length search-list))
  (newline)
  (let ((s-pair (avg-search)))
    (if (not (eq? s-pair 'no-searches))
(begin
  (display "          avg search size range ")
  (let ((avg (car s-pair)))
    (write-string (string-append
   (string #()
   (number->string (floor avg))
   (string #,)
   (number->string (ceiling avg))
```

```
    (string #)))))
   (display ", max search size ")
   (write max-search-size)
   (newline)
   (display "          avg search depth range ")
   (let ((avg (cdr s-pair)))
     (write-string (string-append
    (string #()
    (number-¿string (floor avg))
    (string #,)
    (number-¿string (ceiling avg))
    (string #)))))
   (display ", max search depth ")
   (write 2max-search-depth))))
   (newline)
   (display "number of copy operations: ")
   (write n-copy-ops)
   (newline)
   (display "number of translate operations: ")
   (write n-translates)
   (newline)
   (display "number of graph nodes (cons cells) used: ")
   (write n-nodes-used)
   (newline)
   (display "graph access stats: tot ")
   (write n-g-accesses)
   (display ", avg depth range ")
   (let ((avg 0) (maxd 0))
     (for-each
      (lambda (depth)
        (set! avg (+ depth avg))
        (set! maxd (max maxd depth)))
      access-depth-list)
     (set! avg (/ avg (length access-depth-list)))
     (write-string (string-append
                    (string #()
                    (number-¿string (floor avg))
                    (string #,)
                    (number-¿string (ceiling avg))
                    (string #))))
     (newline) (display "                    ")
     (display "max depth ")
     (write maxd)))


  ; end message handlers


  ; graph access routines

     ; value part: graph form is (value . ") or ((!r . value) . ")
  (define (value gr-ptr)
    (if (and (pair? (car gr-ptr)) (equal? (caar gr-ptr) '!r))
(begin
  (cons-access-depth-list 2)
  (cdar gr-ptr))
(begin
  (cons-access-depth-list 1)
  (car gr-ptr))))

     ; literal part: graph form is ((quote literal-part) . ")
  (define (lit-part gr-ptr)
    (cons-access-depth-list 2)
    (cdar gr-ptr))

     ; variable part: graph form is (variable . ") or ((!r . variable) . ")
  (define (var-in gr-ptr)
    (if (and (pair? (car gr-ptr)) (equal? (caar gr-ptr) '!r))
(begin
  (cons-access-depth-list 2)
  (cdar gr-ptr))
(begin
  (cons-access-depth-list 1)
  (car gr-ptr))))

     ; let: graph form is ((let ((var1 init1) ... (varN initN)) body) . ")
  (define (inits gr-ptr)
    (let ((inits '())
  (bindings (let-bindings gr-ptr))
  (tot-access-depth 3))
        ; for each binding in the list of bindings extract the
        ; init graph and add to the inits graph list. Return inits.
      (for-each
       (lambda (binding)
         (set! tot-access-depth (+ 2 tot-access-depth))
 (set! inits (cons (cdr binding) inits)))
       bindings)
```

```
      (cons-access-depth-list tot-access-depth)
      inits)) ; return inits

  (define (let-vars gr-ptr)
    (let ((vars '())
  (bindings (let-bindings gr-ptr))
           (tot-access-depth 3))
          ; for each binding in the list of bindings extract the
          ; var part and add to the vars list. Return vars list.
      (for-each
       (lambda (binding)
         (set! tot-access-depth (+ 2 tot-access-depth))
(set! vars (cons (car binding) vars)))
        bindings)
      (cons-access-depth-list tot-access-depth)
      vars)) ; return vars

  (define (let-body gr-ptr)
    (cons-access-depth-list 3)
    (cddar gr-ptr))

  (define (let-bindings gr-ptr)
    (cons-access-depth-list 3)
    (cadar gr-ptr))

     ; conditional: graph form is ((if test-part consequent alternate) . *)
  (define (test-part gr-ptr)
    (cons-access-depth-list 2)
    (cdar gr-ptr))

  (define (consequent gr-ptr)
    (cons-access-depth-list 3)
    (cddar gr-ptr))

  (define (alternate gr-ptr)
    (cons-access-depth-list 4)
    (cdddar gr-ptr))

     ; application: graph form is ((fn arg1 ... argN) . *)
  (define (fn-part gr-ptr)
    (cons-access-depth-list 1)
    (car gr-ptr))

  (define (arg-part gr-ptr)
    (cons-access-depth-list 2)
    (cdar gr-ptr))

     ; by the time requests for fbody and fvars are received
     ; f is either a reduced lambda graph (request coming from *red-mgr*
     ; red-app routine) or a lambda graph (request coming from the search-repl
     ; routine):
     ; ((!r . (lambda (var1 ... varN) body))) or
     ; ((lambda (var1 ... varN) body))

  (define (fbody f)
    (if (equal? (caar f) '!r)
(begin
  (cons-access-depth-list 4)
  (cdddar f))
(begin
  (cons-access-depth-list 3)
  (cddar f))))  ; (body)

  (define (fvars f)
    (if (equal? (caar f) '!r)
(begin
  (cons-access-depth-list 4)
  (caddar f))
(begin
  (cons-access-depth-list 3)
  (cadar f)))) ; (var1 ... varN)

  ; end graph gr-access routines


  ; replace-variable-in-graph functions

  (define (search-repl gr-ptr var arg-gr-ptr)
       ; search for a free occurence of var in gr-ptr, and replace
       ; it with arg-gr-ptr.
    (inc-search-size)
    (case ((*graph-mgr* 'get-typ) gr-ptr)
      ((val lit reduced) 'do-nothing)

      ((var) ; rewrite! if the variable in gr-ptr is var
       (if (eq? (var-in gr-ptr) var)
   (begin
```

```
    (inc-n-var-replaces)
    (("graph-mgr" 'rewrite!) gr-ptr arg-gr-ptr))
  'do-nothing))

    ((lam) ; replace only if var does not belong to the list of
           ; bound variables.
     (if (memq var (fvars gr-ptr))
  'do-nothing
  (begin
    (add-search-depth 3)
    (search-repl (fbody gr-ptr) var arg-gr-ptr)
    (sub-search-depth 3))))

    ((let) ; replace var in each init graph, and then replace var in
           ; the let-body only if var does not belong to the list
           ; of bound variables.
     (add-search-depth 3)
     (for-each
(lambda (binding)
  (add-search-depth 1)
  (search-repl (cdr binding) var arg-gr-ptr)
  (sub-search-depth 1))
(let-bindings gr-ptr))
     (sub-search-depth 3)
     (if (memq var (let-vars gr-ptr))
  'do-nothing
  (begin
    (add-search-depth 3)
    (search-repl (let-body gr-ptr) var arg-gr-ptr)
    (sub-search-depth 3))))

    ((letrec) ; if var belongs to the list of bound variables do nothing,
              ; otherwise replace var in each init and in the letrec body.
     (if (memq var (let-vars gr-ptr))
  'do-nothing
  (begin
    (add-search-depth 3)
    (for-each
     (lambda (binding)
(search-repl (cdr binding) var arg-gr-ptr))
     (let-bindings gr-ptr))
    (search-repl (let-body gr-ptr) var arg-gr-ptr)
    (sub-search-depth 3))))

    ((con) ; replace var in test, consequent and alternate parts.
           ; There is opportunity for optimisation, since only one branch
           ; is ever taken. I suspect, however, that it is more expensive
           ; to do the optimisation than to do the replacement.
     (add-search-depth 2)
     (search-repl (test-part gr-ptr) var arg-gr-ptr)
     (add-search-depth 1)
     (search-repl (consequent gr-ptr) var arg-gr-ptr)
     (add-search-depth 1)
     (search-repl (alternate gr-ptr) var arg-gr-ptr)
     (sub-search-depth 4))

    ((app) ; replace var in fn-part and in each arg.
           ; This works because the fn-part is either a variable
           ; or a lambda graph.
     (add-search-depth 1)
     (search-repl (fn-part gr-ptr) var arg-gr-ptr)
     (sub-search-depth 1)
     (search-repl-on-list (arg-part gr-ptr) var arg-gr-ptr))))

(define (search-repl-on-list list-of-grs var arg-gr-ptr)
  (if (null? list-of-grs)
'()
(begin
  (add-search-depth 1)
  (search-repl list-of-grs var arg-gr-ptr)
  (sub-search-depth 1)
  (search-repl-on-list (cdr list-of-grs) var arg-gr-ptr))))

; end variable replace functions

; instrumentation functions and initialisations

(define n-var-replaces 0) ; number of times a variable is replaced
                          ; in a graph.
(define (inc-n-var-replaces)
  (set! n-var-replaces (+ 1 n-var-replaces)))


(define search-size 0)
(define max-search-size 0)
(define search-depth 0)
```

```
  (define max-search-depth 0)
  (define 2max-search-depth 0)
  (define (inc-search-size)
    (set! search-size (+ 1 search-size))
    (set! max-search-size (max search-size max-search-size)))
  (define (init-search) (set! search-depth 0)
    (set! search-size 0) (set! max-search-depth 0))
  (define (add-search-depth n)
    (set! search-depth (+ n search-depth))
    (set! max-search-depth (max search-depth max-search-depth)))
  (define (sub-search-depth n)
    (set! search-depth (- n search-depth)))
  (define (set-2max-search-depth)
    (set! 2max-search-depth (max max-search-depth 2max-search-depth)))

  (define search-list '())

  (define (avg-search)
    (let ((n-searches (length search-list)))
      (if (zero? n-searches)
  'no-searches
  (let ((size-avg 0)
(depth-avg 0))
    (for-each
     (lambda (search-pair)
       (set! size-avg (+ size-avg (car search-pair)))
       (set! depth-avg (+ depth-avg (cdr search-pair))))
     search-list)
    (cons (/ size-avg n-searches) (/ depth-avg n-searches))))))


(define n-copy-ops 0)
(define n-translates 0)
(define n-nodes-used 0)
(define n-g-accesses 0)
(define access-depth-list '())
(define (inc-n-copy-ops) (set! n-copy-ops (+ 1 n-copy-ops)))
(define (add2-n-copy-ops) (set! n-copy-ops (+ 2 n-copy-ops)))
(define (inc-n-translates) (set! n-translates (+ 1 n-translates)))
(define (inc-n-nodes-used) (set! n-nodes-used (+ 1 n-nodes-used)))
(define (inc-n-g-accesses)
  (set! n-g-accesses (+ 1 n-g-accesses)))
(define (cons-access-depth-list depth)
  (set! access-depth-list (cons depth access-depth-list)))

  ; end instrumentation functions and initialisation


  dispatch) ; end make-graph-mgr

; create "graph-mgr" object

(define "graph-mgr" (make-graph-mgr))
```

Appendix C

Symbolic Derivative

This appendix contains the symbolic derivative program:

```
(define deriv
  (lambda (exp var)
    (if (constant? exp) 0
        (if (variable? exp) (if (same-variable? exp var) 1 0)
            (if (sum? exp)
                (make-sum (deriv (addend exp) var)
                          (deriv (augend exp) var))
                (if (product? exp)
                    (make-sum
                     (make-product (multiplier exp)
                                   (deriv (multiplicand exp) var))
                     (make-product (deriv (multiplier exp) var)
                                   (multiplicand exp)))
                    'invalid-expression))))))

(define constant? (lambda (x) (number? x)))

(define variable? (lambda (x) (symbol? x)))

(define same-variable?
  (lambda (v1 v2)
    (if (variable? v1)
        (if (variable? v2)
            (eq? v1 v2)
            #f)
        #f)))

(define sum?
```

```
  (lambda (x)
    (if (pair? x) (eq? (car x) '+) '())))

(define addend (lambda (s) (car (cdr s))))

(define augend (lambda (s) (car (cdr (cdr s)))))

(define product?
  (lambda (x)
    (if (pair? x) (eq? (car x) '*) '())))

(define multiplier (lambda (p) (car (cdr p))))

(define multiplicand (lambda (p) (car (cdr (cdr p)))))

(define make-sum
  (lambda (a1 a2)
    (if (number? a1)
        (if (number? a2)
            (+ a1 a2)
            (if (= a1 0) a2 (cons '+ (cons a1 (cons a2 '())))))
        (if (number? a2)
            (if (= a2 0) a1 (cons '+ (cons a1 (cons a2 '()))))
            (cons '+ (cons a1 (cons a2 '())))))))

(define make-product
  (lambda (m1 m2)
    (if (number? m1)
        (if (number? m2) (* m1 m2)
            (if (= m1 0) 0
                (if (= m1 1) m2
                    (cons '* m1 m2))))
        (if (number? m2)
            (if (= m2 0) 0
                (if (= m2 1) m1
                    (cons '* (cons m1 (cons m2 '())))))
            (cons '* (cons m1 (cons m2 '())))))))
```

Appendix D

Log Files

This appendix contains the log files as they are generated by the Environment and Reduction Evaluators.

D.1 Environment Evaluator Log File

```
==============================================

Exp: (fact 1)
Value: 1

eval-mgr* statistics:
number of recursive calls: 6
maximum recursion depth: 2
number of evaluations: 9

name-space-mgr* statistics:
lookup stats: tot 3, avg depth range (1,2), max depth 4
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 1
number of frames added: 1

primitives* statistics:
number of primitive applies: 1
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 10, avg depth range (1,2)
                         max depth 4
number of type-ops: 9
number of cons cells used: 0
==============================================

Exp: (fact 5)
Value: 120

eval-mgr* statistics:
number of recursive calls: 54
maximum recursion depth: 6
number of evaluations: 65

name-space-mgr* statistics:
lookup stats: tot 31, avg depth range (2,3), max depth 8
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 5
number of frames added: 5

primitives* statistics:
number of primitive applies: 13
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 62, avg depth range (1,2)
                         max depth 4
number of type-ops: 65
number of cons cells used: 0
==============================================

Exp: (fact 10)
Value: 3628800

eval-mgr* statistics:
number of recursive calls: 114
maximum recursion depth: 11
number of evaluations: 135

name-space-mgr* statistics:
lookup stats: tot 66, avg depth range (2,3), max depth 8
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 10
number of frames added: 10

primitives* statistics:
number of primitive applies: 28
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 127, avg depth range (1,2)
                         max depth 4
number of type-ops: 135
number of cons cells used: 0
==============================================

Exp: (fact-iter 1 1)
Value: 1

eval-mgr* statistics:
number of recursive calls: 20
```

```
maximum recursion depth: 2
number of evaluations: 25

name-space-mgr* statistics:
lookup stats: tot 12, avg depth range (2,3), max depth 9
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 4
number of frames added: 2

primitives* statistics:
number of primitive applies: 4
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 23, avg depth range (1,2)
                         max depth 4
number of type-ops: 25
number of cons cells used: 0
==========================================

Exp: (fact-iter 1 5)
Value: 120

eval-mgr* statistics:
number of recursive calls: 72
maximum recursion depth: 2
number of evaluations: 85

name-space-mgr* statistics:
lookup stats: tot 44, avg depth range (2,3), max depth 9
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 12
number of frames added: 6

primitives* statistics:
number of primitive applies: 16
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 75, avg depth range (1,2)
                         max depth 4
number of type-ops: 85
number of cons cells used: 0
==========================================

Exp: (fact-iter 1 10)
Value: 3628800

eval-mgr* statistics:
number of recursive calls: 137
maximum recursion depth: 2
number of evaluations: 160

name-space-mgr* statistics:
lookup stats: tot 84, avg depth range (2,3), max depth 9
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 22
number of frames added: 11

primitives* statistics:
number of primitive applies: 31
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 140, avg depth range (1,2)
                         max depth 4
number of type-ops: 160
number of cons cells used: 0
==========================================

Exp: (mrec-even? 1)
Value: ()

eval-mgr* statistics:
number of recursive calls: 19
maximum recursion depth: 2
number of evaluations: 26

name-space-mgr* statistics:
lookup stats: tot 10, avg depth range (2,3), max depth 11
              avg frame depth range (1,2), max frame depth 3
number of bindings added: 5
number of frames added: 4

primitives* statistics:
number of primitive applies: 3
number of cons operations: 0
```

```
exp-mgr* statistics:
expression access stats: tot 29, avg depth range (2,3)
                         max depth 6
number of type-ops: 26
number of cons cells used: 8
==============================================

Exp: (mrec-even? 10)
Value: #T

eval-mgr* statistics:
number of recursive calls: 100
maximum recursion depth: 2
number of evaluations: 125

name-space-mgr* statistics:
lookup stats: tot 55, avg depth range (3,4), max depth 11
              avg frame depth range (1,2), max frame depth 3
number of bindings added: 14
number of frames added: 13

primitives* statistics:
number of primitive applies: 21
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 128, avg depth range (1,2)
                         max depth 6
number of type-ops: 125
number of cons cells used: 8
==============================================

Exp: (mrec-even? 81)
Value: ()

eval-mgr* statistics:
number of recursive calls: 739
maximum recursion depth: 2
number of evaluations: 906

name-space-mgr* statistics:
lookup stats: tot 410, avg depth range (3,4), max depth 11
              avg frame depth range (1,2), max frame depth 3
number of bindings added: 85
number of frames added: 84

primitives* statistics:
number of primitive applies: 163
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 909, avg depth range (1,2)
                         max depth 6
number of type-ops: 906
number of cons cells used: 8
==============================================

Exp: (deriv '(* (+ x y) (* x (+ y z))) 'x)
Value: (+ (* (+ x y) (+ y z)) (* x (+ y z)))

eval-mgr* statistics:
number of recursive calls: 472
maximum recursion depth: 9
number of evaluations: 608

name-space-mgr* statistics:
lookup stats: tot 338, avg depth range (10,11), max depth 31
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 90
number of frames added: 68

primitives* statistics:
number of primitive applies: 111
number of cons operations: 6

exp-mgr* statistics:
expression access stats: tot 709, avg depth range (1,2)
                         max depth 4
number of type-ops: 608
number of cons cells used: 0
==============================================

Exp: (deriv '(* x (+ y 1)) 'x)
Value: (+ y 1)

eval-mgr* statistics:
```

```
number of recursive calls: 224
maximum recursion depth: 7
number of evaluations: 291

name-space-mgr* statistics:
lookup stats: tot 160, avg depth range (10,11), max depth 31
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 44
number of frames added: 33

primitives* statistics:
number of primitive applies: 52
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 341, avg depth range (1,2)
                         max depth 4
number of type-ops: 291
number of cons cells used: 0
==========================================

Exp: (tak 1 2 3)
Value: 3

eval-mgr* statistics:
number of recursive calls: 10
maximum recursion depth: 3
number of evaluations: 13

name-space-mgr* statistics:
lookup stats: tot 6, avg depth range (3,4), max depth 15
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 3
number of frames added: 1

primitives* statistics:
number of primitive applies: 2
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 12, avg depth range (1,2)
                         max depth 4
number of type-ops: 13
number of cons cells used: 0
==========================================

Exp: (tak 3 2 1)
Value: 2

eval-mgr* statistics:
number of recursive calls: 59
maximum recursion depth: 4
number of evaluations: 70

name-space-mgr* statistics:
lookup stats: tot 41, avg depth range (4,5), max depth 15
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 15
number of frames added: 5

primitives* statistics:
number of primitive applies: 13
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 62, avg depth range (1,2)
                         max depth 4
number of type-ops: 70
number of cons cells used: 0
==========================================

Exp: (tak 2 4 8)
Value: 8

eval-mgr* statistics:
number of recursive calls: 10
maximum recursion depth: 3
number of evaluations: 13

name-space-mgr* statistics:
lookup stats: tot 6, avg depth range (3,4), max depth 15
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 3
number of frames added: 1

primitives* statistics:
number of primitive applies: 2
```

```
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 12, avg depth range (1,2)
                         max depth 4
number of type-ops: 13
number of cons cells used: 0
==========================================

Exp: (tak 8 4 2)
Value: 3

eval-mgr* statistics:
number of recursive calls: 1676
maximum recursion depth: 9
number of evaluations: 1951

name-space-mgr* statistics:
lookup stats: tot 1196, avg depth range (4,5), max depth 15
              avg frame depth range (0,1), max frame depth 1
number of bindings added: 411
number of frames added: 137

primitives* statistics:
number of primitive applies: 376
number of cons operations: 0

exp-mgr* statistics:
expression access stats: tot 1712, avg depth range (1,2)
                         max depth 4
number of type-ops: 1951
number of cons cells used: 0
```

D.2 Reduction Evaluator Log File

```
==============================================

Exp: (fact 1)
Value: 1

red-mgr statistics:
number of recursive calls: 6
maximum recursion depth: 2
number of reductions: 9

name-space-mgr* statistics:
lookup stats: tot 2, avg depth range (1,2), max depth 3

primitives* statistics:
number of primitive applies: 1
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 3
          tot number of searches is 1
          avg search size range (15,15), max search size 15
          avg search depth range (3,3), max search depth 3
number of copy operations: 40
number of translate operations: 2
number of graph nodes (cons cells) used: 25
graph access stats: tot 18, avg depth range (1,2)
                    max depth 4
==============================================

Exp: (fact 5)
Value: 120

red-mgr statistics:
number of recursive calls: 54
maximum recursion depth: 6
number of reductions: 65

name-space-mgr* statistics:
lookup stats: tot 18, avg depth range (3,4), max depth 7

primitives* statistics:
number of primitive applies: 13
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 15
          tot number of searches is 5
          avg search size range (15,15), max search size 15
          avg search depth range (3,3), max search depth 3
number of copy operations: 172
number of translate operations: 14
number of graph nodes (cons cells) used: 121
graph access stats: tot 152, avg depth range (1,2)
                    max depth 4
==============================================

Exp: (fact 10)
Value: 3628800

red-mgr statistics:
number of recursive calls: 114
maximum recursion depth: 11
number of reductions: 135

name-space-mgr* statistics:
lookup stats: tot 38, avg depth range (3,4), max depth 7

primitives* statistics:
number of primitive applies: 28
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 30
          tot number of searches is 10
          avg search size range (15,15), max search size 15
          avg search depth range (3,3), max search depth 3
number of copy operations: 337
number of translate operations: 29
number of graph nodes (cons cells) used: 241
graph access stats: tot 583, avg depth range (1,2)
                    max depth 4
==============================================
```

```
Exp: (fact-iter 1 1)
Value: 1

red-mgr statistics:
number of recursive calls: 20
maximum recursion depth: 2
number of reductions: 25

name-space-mgr* statistics:
lookup stats: tot 6, avg depth range (2,3), max depth 7

primitives* statistics:
number of primitive applies: 4
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 10
         tot number of searches is 4
         avg search size range (16,16), max search size 16
         avg search depth range (4,4), max search depth 4
number of copy operations: 79
number of translate operations: 5
number of graph nodes (cons cells) used: 52
graph access stats: tot 630, avg depth range (1,2)
                    max depth 4
================================================

Exp: (fact-iter 1 5)
Value: 120

red-mgr statistics:
number of recursive calls: 72
maximum recursion depth: 2
number of reductions: 85

name-space-mgr* statistics:
lookup stats: tot 22, avg depth range (3,4), max depth 7

primitives* statistics:
number of primitive applies: 16
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 30
         tot number of searches is 12
         avg search size range (16,16), max search size 16
         avg search depth range (4,4), max search depth 4
number of copy operations: 219
number of translate operations: 17
number of graph nodes (cons cells) used: 152
graph access stats: tot 793, avg depth range (1,2)
                    max depth 4
================================================

Exp: (fact-iter 1 10)
Value: 3628800

red-mgr statistics:
number of recursive calls: 137
maximum recursion depth: 2
number of reductions: 160

name-space-mgr* statistics:
lookup stats: tot 42, avg depth range (3,4), max depth 7

primitives* statistics:
number of primitive applies: 31
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 55
         tot number of searches is 22
         avg search size range (16,16), max search size 16
         avg search depth range (4,4), max search depth 4
number of copy operations: 394
number of translate operations: 32
number of graph nodes (cons cells) used: 277
graph access stats: tot 1101, avg depth range (1,2)
                    max depth 4
================================================

Exp: (mrec-even? 1)
Value: ()

red-mgr statistics:
number of recursive calls: 19
```

```
maximum recursion depth: 2
number of reductions: 26

name-space-mgr* statistics:
lookup stats: tot 4, avg depth range (3,4), max depth 7

primitives* statistics:
number of primitive applies: 3
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 8
          tot number of searches is 9
          avg search size range (12,13), max search size 30
          avg search depth range (5,5), max search depth 13
number of copy operations: 150
number of translate operations: 4
number of graph nodes (cons cells) used: 87
graph access stats: tot 1150, avg depth range (1,2)
                    max depth 7
==============================================

Exp: (mrec-even? 10)
Value: #T

red-mgr statistics:
number of recursive calls: 100
maximum recursion depth: 2
number of reductions: 125

name-space-mgr* statistics:
lookup stats: tot 22, avg depth range (4,5), max depth 7

primitives* statistics:
number of primitive applies: 21
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 26
          tot number of searches is 18
          avg search size range (12,13), max search size 30
          avg search depth range (4,4), max search depth 13
number of copy operations: 393
number of translate operations: 22
number of graph nodes (cons cells) used: 258
graph access stats: tot 1397, avg depth range (1,2)
                    max depth 7
==============================================

Exp: (mrec-even? 81)
Value: ()

red-mgr statistics:
number of recursive calls: 739
maximum recursion depth: 2
number of reductions: 906

name-space-mgr* statistics:
lookup stats: tot 164, avg depth range (4,5), max depth 7

primitives* statistics:
number of primitive applies: 163
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 168
          tot number of searches is 89
          avg search size range (12,13), max search size 30
          avg search depth range (3,4), max search depth 13
number of copy operations: 2310
number of translate operations: 164
number of graph nodes (cons cells) used: 1607
graph access stats: tot 3206, avg depth range (1,2)
                    max depth 7
==============================================

Exp: (deriv '(* (+ x y) (* x (+ y z))) 'x)
Value: (+ (* (+ x y) (+ y z)) (* x (+ y z)))

red-mgr statistics:
number of recursive calls: 472
maximum recursion depth: 9
number of reductions: 608

name-space-mgr* statistics:
lookup stats: tot 179, avg depth range (17,18), max depth 30
```

```
primitives* statistics:
number of primitive applies: 111
number of cons operations: 6

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 344
         tot number of searches is 90
         avg search size range (27,28), max search size 65
         avg search depth range (7,8), max search depth 16
number of copy operations: 3289
number of translate operations: 112
number of graph nodes (cons cells) used: 1857
graph access stats: tot 4629, avg depth range (1,2)
                    max depth 7
================================================

Exp: (deriv '(* x (+ y 1)) 'x)
Value: (+ y 1)

red-mgr statistics:
number of recursive calls: 224
maximum recursion depth: 7
number of reductions: 291

name-space-mgr* statistics:
lookup stats: tot 85, avg depth range (17,18), max depth 30

primitives* statistics:
number of primitive applies: 52
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 177
         tot number of searches is 44
         avg search size range (29,30), max search size 65
         avg search depth range (8,9), max search depth 16
number of copy operations: 1700
number of translate operations: 53
number of graph nodes (cons cells) used: 951
graph access stats: tot 5310, avg depth range (1,2)
                    max depth 7
================================================

Exp: (tak 1 2 3)
Value: 3

red-mgr statistics:
number of recursive calls: 10
maximum recursion depth: 3
number of reductions: 13

name-space-mgr* statistics:
lookup stats: tot 3, avg depth range (4,5), max depth 12

primitives* statistics:
number of primitive applies: 2
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 12
         tot number of searches is 3
         avg search size range (34,34), max search size 34
         avg search depth range (4,4), max search depth 4
number of copy operations: 82
number of translate operations: 3
number of graph nodes (cons cells) used: 48
graph access stats: tot 5334, avg depth range (1,2)
                    max depth 7
================================================

Exp: (tak 3 2 1)
Value: 2

red-mgr statistics:
number of recursive calls: 59
maximum recursion depth: 4
number of reductions: 70

name-space-mgr* statistics:
lookup stats: tot 18, avg depth range (5,6), max depth 12

primitives* statistics:
number of primitive applies: 13
number of cons operations: 0

graph-mgr statistics:
```

```
variable find-and-replace stats: tot number of replaces is 60
         tot number of searches is 15
         avg search size range (34,34), max search size 34
         avg search depth range (4,4), max search depth 4
number of copy operations: 366
number of translate operations: 14
number of graph nodes (cons cells) used: 213
graph access stats: tot 5468, avg depth range (1,2)
                    max depth 7
==========================================

Exp: (tak 2 1 8)
Value: 8

red-mgr statistics:
number of recursive calls: 10
maximum recursion depth: 3
number of reductions: 13

name-space-mgr* statistics:
lookup stats: tot 3, avg depth range (4,5), max depth 12

primitives* statistics:
number of primitive applies: 2
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 12
         tot number of searches is 3
         avg search size range (34,34), max search size 34
         avg search depth range (4,4), max search depth 4
number of copy operations: 82
number of translate operations: 3
number of graph nodes (cons cells) used: 48
graph access stats: tot 5492, avg depth range (1,2)
                    max depth 7
==========================================

Exp: (tak 8 4 2)
Value: 3

red-mgr statistics:
number of recursive calls: 1676
maximum recursion depth: 9
number of reductions: 1951

name-space-mgr* statistics:
lookup stats: tot 513, avg depth range (5,6), max depth 12

primitives* statistics:
number of primitive applies: 376
number of cons operations: 0

graph-mgr statistics:
variable find-and-replace stats: tot number of replaces is 1644
         tot number of searches is 411
         avg search size range (34,34), max search size 34
         avg search depth range (4,4), max search depth 4
number of copy operations: 9738
number of translate operations: 377
number of graph nodes (cons cells) used: 5658
graph access stats: tot 9256, avg depth range (1,2)
                    max depth 7
```