Representing Control in Parallel Applicative Programming

Chi Yao

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 1994

Approved:

Benjamin Goldberg, Research Advisor

©Copyright by Chi Yao, 1994

All rights Reserved

To My Parents

Acknowledgments

First of all, I would like to express my deepest gratitude to my advisor Benjamin Goldberg for his guidance, supports, and encouragements over these years. He has helped a lot to realize this research idea since its formation. I am very glad that we have walked through this together.

I would like to thank Suresh Jagannathan and Edmond Schonberg for providing helpful comments on my work in the early stage and giving me suggestions in the implementation. My thanks also go to Malcolm Harrison and Ernest Davis, who, as my committee members, provide many interesting ideas regarding my thesis research.

A lot of effort of this research is spent in the implementation. I am glad that I have a chance to develop the system on NYU's Ultracomputer. Thank Allan Gottlieb and the Ultra Lab. They have been of great help during the passed year. I have benefited a lot especially from my discussion with Jan Edler and Eric Freudenthal. Thank you both.

I had many inspirational discussions with my fellow students in Courant, especially with Tyng-Ruey Chuang, Karpjoo Jeong, Yaw-Tai Lee and Shih-Chen Huang. They, together with many friends, have made my student life here memorable.

My cousin and the Fang's family have also given me a wonderful time in New York. My girl friend Jean has always been supportive and understanding. I couldn't imagine how I would have survived without them. But now, I am so glad to share my happiness with them.

Finally, I want to sincerely thank my parents. They always have confidence in me, and always accompany me to go through tough moments even they're on the other side of the earth. I dedicate all my accomplishments to them. Thank you, Mom and Dad.

This research was supported in part by a grant from ARPA/ONR (contract #N00014-92-J-1719).

Abstract

This research is an attempt to reason about the control of parallel computation in the world of applicative programming languages.

Applicative languages, in which computation is performed through function application and in which functions are treated as first-class objects, have the benefits of elegance, expressiveness and having clean semantics. Parallel computation and real-world concurrent activities are much harder to reason about than the sequential counterparts. Many parallel applicative languages have thus hidden most control details with their declarative programming styles, but they are not expressive enough to characterize many real world concurrent activities that can be easily explained with concepts such as message passing, pipelining and so on.

Ease of programming should not come at the expense of expressiveness. Therefore, we design a parallel applicative language **Pscheme** such that programmers can express explicitly the control of parallel computation while maintaining the clean semantics and the ease of programming of applicative languages. In Pscheme, we propose the concept of *ports* to model the general control in parallel computation. Through program examples, we show how Pscheme and ports support various parallel programming paradigms. We have also built libraries for higher level control facilities with ports so that programming in Pscheme becomes easier.

We provide an operational semantics for Pscheme, and develop a compiler and a run time system on NYU's Ultracomputer. Our experiments with parallel programs have shown satisfactory speedup. We claim that ports are the natural parallel extensions of *continuations* in sequential computation, and thus conclude that representing general control in parallel applicative programming is feasible.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Approach and Background 3

1.2.1 Continuations 3

1.2.2 Extensions of Continuations 4

1.3 Dissertation Outline 6

2 Pscheme – A Parallel Applicative Language Providing a General Control Mechanism **9**

2.1 Main Features 10

2.2 Definitions of Pscheme Parallel Constructs 11

2.2.1 `pcall`: A Simple Construct for Expressing Parallelism 11

2.2.2 Ports: Parallel Extensions of Continuations 11

2.2.3 Multi vs. Single Ports 13

2.2.4 Pscheme Thread Creation and Termination 13

2.2.5 Exclusive Functions 14

2.2.6 Further Notes about Definitions 17

2.3 The Relationship Between Ports and Continuations 18

3 Programming in Pscheme **21**

3.1 Divide and Conquer – Horizontal Parallelism 22

3.2 Using `pcall` and Ports as a Synchronizing Mechanism 22

3.3 Blocking and Locking 26

3.4 Shared Mutable Objects and Atomic Assignments 28

3.5 Shared FIFO Queues 31

3.6 Futures 33

3.7 Stream-based Programming and Vertical Parallelism 35

3.7.1 A Small Example – Stream of Fibonacci Numbers 36

3.7.2 Filters 37

3.8 Applications 41

3.8.1 Quicksort 41

3.8.2 Merge Sort 43

3.9 Remarks 46

4 Operational Semantics **47**

4.1 The Abstract Machine 47

4.2 Meanings of Semantic Notations 48

4.3 Operational Semantics of Pscheme 50

4.3.1 Basic Reductions Regarding Sequential Computation 51

4.3.2 Exclusive Functions 52

4.3.3 Function Application 53

4.3.4 Blocking 55

4.3.5 Environment Recovery 57

4.3.6 First Class Ports 57

4.3.7 Parallel Control 58

4.4 An Example 60

5 The Compiler **63**

5.1 Intermediate Representation 65

5.2 Syntactic Preprocessing, Scope analysis and Assignment Conversion . . . 69

5.2.1 Syntactic Preprocessing 69

5.2.2 Scope Analysis 70

5.2.3 Assignment Conversion 70

5.3 The CPS Transform Process 72

5.3.1 Transforming Sequential Expressions 72

5.3.2 Ports and Parallel Constructs 74

5.3.3 Mutual Recursion and `letrec` 81

5.4 Free Variable Set Computing and Hoisting 83

5.5 Closure Conversion 86

5.6 Thread Abstraction 89

5.7 Heap-based Code Generation 91

6 The Run Time System **95**

6.1 The Execution Model and The Current Platform 95

6.2 Implementation of The Run Time System 98

6.3 Storage Management 99

6.4 Garbage Collection of The Shared Heap 100

6.5 The Ordering Mechanism 101

6.5.1 Implementation of Exclusive Functions 102

7 Performance **105**

7.1 Sample Parallel Programs 105

7.2 Stream Based Programs 110
7.3 Sequential Programs 113

8 Related Work **117**

8.1 Multilisp and Futures 117
8.2 Qlisp 119
8.3 Actors 120
8.4 Lucid 121
8.5 Other Models of Parallel Continuations 122
8.6 Pscheme's Approach to Parallelism 123
8.7 Comparison in Terms of Expressiveness 124

9 Future Work and Conclusion **127**

9.1 Future Work 127
9.1.1 Compiler Optimization 127
9.1.2 Run Time System Improvement 128
9.1.3 I/O and System Interface 128
9.2 Conclusion 128

A Transforming The Parallel Sum Program **131**

A.1 Input Program 131
A.2 Continuation Passing Style: (hoisted) 131
A.3 Closure Passing Style: (hoisted) 133
A.4 Thread Abstraction 136

B Internal Representation of Pscheme Objects **141**

Bibliography **146**

List of Figures

2.1 A simple data-flow diagram 12
2.2 Finding an element in a tree in parallel 15
2.3 Checking if two trees have the same fringe in parallel. In this program, `nil` is not considered as a leaf but a null child link. 16
2.4 A sequential representation of function application using CPS 19

3.1 Parallel sum 22
3.2 Quicksort 23
3.3 Implementation of Ada-like rendezvous 25
3.4 Implementation of semaphores 27
3.5 Two implementations of stores supporting atomic access 30
3.6 Implementation of shared FIFO queues 32
3.7 Implementation of Multilisp futures 34
3.8 A data-flow diagram for computing the Fibonacci stream 36
3.9 Implementation of the Fibonacci stream 37
3.10 Implementing the Fibonacci stream without using `set!` 38
3.11 Conversion between lists and streams 39
3.12 Generating the stream of prime numbers 40
3.13 Stream-based quicksort 42

3.14 Stream-based merge sort 44
3.15 Implementing the cell merging two sorted streams 45

4.1 Notation definitions in Pscheme's operational semantics 49

5.1 The CPS transform algorithm for sequential constructs 75
5.2 The CPS transformed Factorial function 76
5.3 The CPS transform algorithm for parallel constructs 78
5.4 An example of hoisting 85
5.5 An example of closure conversion 88
5.6 Closure conversion with restricted sharing 90

6.1 Pscheme's execution model 96

7.1 Three sample programs for parallel performance 106
7.2 Sample program for the N-queens problem 107
7.3 Performance of the four parallel sample programs 109
7.4 Speedup curve for all sample programs 111
7.5 Performance for two stream based programs 112
7.6 Two sequential sample programs 114

List of Tables

7.1 Execution time (ms) of sequential programs on different platforms 114

7.2 Execution time (ms) of sequential programs through different implementations 114

8.1 Paradigms for various parallel languages 125

Chapter 1

Introduction

This research is an attempt to reason about parallel computation in the world of applicative programming by exposing control details in the language level to obtain more expressive power in a language.

In the dissertation, we propose a parallel applicative language Pscheme to illustrate our idea. We provide a formal definition of Pscheme, describe its implementation, and perform experiments with various examples.

1.1 Motivation

Applicative languages, in which computation is performed through function application and in which functions are treated as first-class objects, have the benefits of elegance, expressiveness and having clean semantics.

With the increasing need of parallel computation, there have been proposals for various parallel applicative languages. Since parallel computation and real-world concurrent activities are much harder to reason about than the sequential counterparts, many parallel languages have hidden most control details and support a particular programming paradigm which represents a certain aspect of parallel computation. In the case of ap-

plicative languages, the aspect that most parallel dialects illustrate is *to apply functions in parallel.*

Clearly, not all parallel behaviors can be modeled by function application, but the declarative style of applicative programming hides all other control details. Ease of programming should not come at the expense of expressiveness. Therefore, the goal of this research is to design a parallel applicative language such that programmers can express explicitly the control of parallel computation while maintaining the clean semantics and the ease of programming of applicative languages.

Specifically, we are proposing a language that

- supports more expressive parallel applicative programming,
- provides a general control mechanism in parallel computation,
- is compatible with different parallel programming paradigms, and
- can characterize various parallel behaviors in the real world.

We also believe that parallel computing means more than high performance computing. From a language designer's view point, a language should executes well on various computer architectures, so should it model well on various issues and activities. Through our proposed language, we hope to answer the following questions:

- How expressive can a parallel language be?
- How hard is it for a programmer to reason about or model parallel computation?
- How efficient can it be for a parallel language to support multiple programming paradigms?

1.2 Approach and Background

To provide a general control mechanism in a parallel language, it is not enough to have just a set of process control primitives. It would be better to have an orthogonal construct that can represent a particular control point during the computation. This approach is adopted by many sequential languages.

In the world of lambda-calculus based languages, each expression has a return value, and the returning of this value corresponds to a state in the course of the whole computation based on a particular evaluation order. This state can be made available to programmers as a language-level construct so as to model control in many applicative languages, and is called a *continuation*. Since we are focusing on parallel applicative programming, we would like to model as well the state that an expression represents during the parallel execution of a program This motivates our proposing the concept of *ports*, which we will explain in following chapters.

1.2.1 Continuations

The concept of continuations traces back to denotational models of the call-by-value lambda calculus [53,54]. Traditional semantics of a lambda expression is based on a meaning function of a domain expression and an environment. The meaning function gives a denotation of the expression in the environment. An alternative semantic model is the continuation model, in which, the meaning function takes one more argument – a continuation function, and the denotation of the expression in the environment is obtained by applying the continuation function with the result of interpreting the expression in the environment. This is called the *continuation passing style* (CPS) semantics.

The CPS semantics makes the expression evaluation order explicit by specifying each sub-expression's meaning and its continuation in turn so as to get the meaning of the whole

expression. This denotational model helps provide language-level continuations because it becomes easier to develop a CPS denotational semantics for a language providing first-class continuations, and it is also easier to implement it.

In most applicative languages, continuations appear in the form of one-argument functions. At the interpretation meta-level, however, applying such a one-argument function causes computation to go on towards the completion of the whole evaluation. In other words, application of continuation function never returns even if it happens within an expression block. In fact, it serves as a control point in the course of the whole computation. Such a control point, when provided in the language level, becomes a very useful tool to programmers.

Therefore, with first-class continuations provided, programs become more expressive because any control point can be *captured* in the form of a continuation. Various control facilities such as non-local escape, J-operator [48], label values [56], coroutines [32], and engines [14] can hence be implemented.

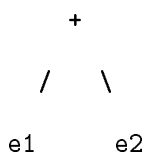
It is known that continuations provide a general control mechanism in applicative languages with acceptable performances (close to high order languages without continuations). However, continuations also decrease the readability of programs written in applicative languages. Solving the trade-off between readability and expressiveness relies on building higher level control operators so that the language has enough expressive power while it is still easy to program with.

1.2.2 Extensions of Continuations

In a parallel applicative language, it is natural to represent control with an extension of continuations. There have been attempts to propose *parallel continuations* [33]. Since continuations represent control points in sequential threads, most proposed parallel con-

tinuations represent *the rest of the computation* for a particular process. Invoking a parallel continuation still behaves like a transfer of control, but the concept of *the rest of computation* does not apply across processes.

Our approach views *parallel continuations* as the links in the data-flow diagram representing the expression being evaluated in parallel. For example, in a Scheme expression `(+ e1 e2)`, `e1` and `e2` may be evaluated in parallel before `+` is applied. The two links in the data-flow diagram shown below can be reified as language-level constructs which represents *ports* to the `+` operator. The `+` operation will be triggered when data elements flow to both ports.



In any given expression of an applicative language, there is a corresponding data-flow diagram. If those links in the data-flow diagram become first-class language-level constructs, it is then possible to make any link accessible to any expression node, and hence enables us to provide a general control mechanism with them.

Such intuition motivates our designing the language Pscheme, a parallel dialect of Scheme, and its primary new construct, the *port*, a concrete representation for those conceptual links. Ports distinguish continuations conceptually in the following aspects:

1. A port no longer represents the rest of the computation. In the above example, each port only represents some information about the rest of computation.

2. A port is no longer a procedural abstraction or a label for control to be transferred to. Data elements passed to a port sometimes have to synchronize with those in other ports before following computation starts.

Formal and detailed definitions regarding ports are provided in subsequent chapters. In our approach to model control in parallel applicative programming, we believe that ports are the natural parallel extension of continuations not only in the language level, but also in the semantics and implementation aspects.

Ports are useful to explain the semantics of multi-threaded applicative languages. As continuations are used in compilers for applicative languages (the intermediate representation is in continuation passing style), ports can be also be used to implement synchronization among function arguments evaluated in parallel. We will give further details in chapters regarding formal semantics and the Pscheme compiler.

1.3 Dissertation Outline

The dissertation presents the work to represent general control in parallel applicative programming through the language Pscheme. In chapter 2, we define Pscheme, describe its features, and introduce the first-class language construct port.

Chapter 3 discusses programming in Pscheme and presents various programming styles. It demonstrates how ports can be used to represent different control facilities.

Chapter 4 gives a formal definition of Pscheme. We have an operational semantics which interprets Pscheme expressions in terms of transition rules on our abstract reduction machine.

The implementation work is shown in chapter 5 and 6. Chapter 5 describes our Pscheme-to-C translator, and chapter 6 discusses the run time system and our execution model.

We present our performance results in chapter 7, and discuss issues of scalability, granularity, slow down factors and so on.

Chapter 8 discusses related works in other languages including: Multilisp, Actors,

Lucid and Qlisp, and other parallel extensions of continuations. We will also make comparisons between Pscheme and those languages.

Chapter 9 describes our future work and concludes the dissertation.

Chapter 2

Pscheme – A Parallel Applicative Language Providing a General Control Mechanism

In this chapter, we describe Pscheme, a parallel dialect of Scheme [Cl91]. The primary construct for specifying parallelism, synchronization, and communication is a natural extension of first-class continuations which we call a *port*. The benefit of using Pscheme is that the user has complete control over the order of parallel evaluation, while at the same time benefiting from the use of a very high level language like Scheme. Other parallel variants of LISP and Scheme have been proposed, based on first-class continuations, but we feel that our ports provide the most natural parallel extension of sequential continuations of the methods suggested.

Like first-class continuations, ports are very powerful constructs that can lead to complicated programs that are difficult to read. We envision ports (and continuations) to be primarily the tool of advanced system programmers who provide libraries of higher level parallel constructs (futures, barriers, engines, etc.) for use by the general programming

community. We show how ports are sufficiently powerful to easily implement such high level parallel constructs in chapter 3.

We present the language Pscheme in this chapter through our design motivation and short program examples. Formal semantics will be given later in chapter 4.

2.1 Main Features

Pscheme is a parallel extension of the Lisp dialect Scheme. It can be viewed as the following formula shows:

```
Pscheme = Scheme + Parallel Control + First Class Ports
```

The main features of Pscheme are:

1. As in Scheme, Pscheme is lexically scoped, has call by value semantics, and supports first-class functions.

2. It is a multi-threaded language with a global name space.

3. It provides a first-class construct named port, which represents a particular control point and its related environmental information in the course of computation.

4. It provides an ordering mechanism through ports so that order preservation of data elements, messages or thread execution is all possible.

5. It provides exclusively accessed functions (similar to Hoare's monitor [35]) so that there can only be one activation of an *exclusive function* at any time during the execution. These functions are used to control the thread ordering as well.

6. It supports *horizontal parallelism*, in which the arguments in a function call are evaluated in parallel, as well as *vertical parallelism*, which is parallelism between a producer (e.g. of stream of values) and a consumer.

As the above formula shows, Scheme is a subset of Pscheme. A program that does not contain any parallel construct has exactly the same semantics as in Scheme. In other words, any sequential thread in Pscheme is actually the execution of a Scheme expression.

2.2 Definitions of Pscheme Parallel Constructs

Pscheme contains relatively few constructs for specifying parallel program behavior. These constructs, primarily `pcall`, `call/mp`, `call/sp`, `throw`, `die`, and `exclusive` are simple to describe and use, but are very powerful when used in conjunction with each other.

2.2.1 `pcall`: A Simple Construct for Expressing Parallelism

Before discussing ports and continuations, we introduce `pcall`, the basic construct for expressing parallelism in Pscheme. It is a simple fork-join construct seen in many languages. The innovation of the work described here results from the interaction of `pcall` with ports, as described later in this paper. The `pcall` construct takes the form

```
(pcall e0 e1 ... en)
```

and causes the evaluation of `e1, ... , en` in parallel after `e0` is evaluated. When each of the expressions `ei` has been evaluated to its value v_i, the procedure value v_0 (resulting from evaluating `e0`) is invoked with arguments $v_1 \cdots v_n$. It is important to note that the body of the procedure is not invoked until the parallel evaluation of the arguments has completed. Thus, pcall serves as a barrier synchronization for the evaluation of the expressions `e1 ... en`. This synchronization becomes very important in our later examples.

2.2.2 Ports: Parallel Extensions of Continuations

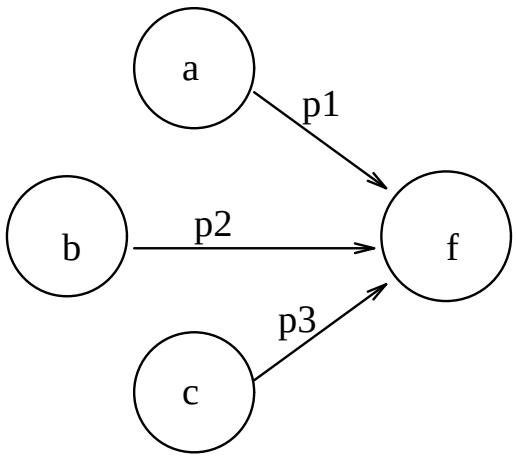


Figure 2.1: A simple data-flow diagram

The expression `(pcall f a b c)` can be thought of as being graphically represented by the picture in figure 2.1.

When a, b, and c have been evaluated, their values are sent along their corresponding arcs to f. In order to capture the arcs as first-class values, we introduce a construct called *call-with-current-multiport*, which is written `call/mp` (the term "multi-" will be addressed later). Like its sequential analog `call/cc`, `call/mp` takes a single parameter which is a function that takes a single parameter, the current port, that is, the arc.

Thus, the expression

```
(pcall f (call/mp (lambda (p1) a))
         (call/mp (lambda (p2) b))
         (call/mp (lambda (p3) c)))
```

binds `p1`, `p2`, and `p3` to the arcs as shown in figure 2.1

Looking at the illustration as a data-flow graph, it is clear that `f` is invoked whenever a trio of values are sent to ports `p1`, p2, and p3. It was mentioned above that one way to produce values along the arcs is for the evaluation of a, b, and c to complete. However,

now that `p1`, `p2`, and `p3` are tangible objects, values can also be sent down the arcs by explicitly *throwing* the values to the respective ports (using the `throw` construct described below). With the explicit use of ports, `f` can be invoked many times, once each time all of its input ports have a value available (due to a throw or a returned value). Thus, not only is parallelism generated by the use of pcall, but also by multiple invocations of `f` occurring in parallel due to multiple values being thrown to each of `f`'s input ports. The image of a single `f` being a consumer of the values is no longer accurate. A new activation of `f` is created for each trio of values thrown to its input ports (later on, we will introduce a new construct, `exclusive`, that specifies that only one activation of `f` can execute at any given time).

2.2.3 Multi vs. Single Ports

It might also be desirable for a given function to be invoked only once, but to be able to invoke it either by having its arguments return values, or by explicitly passing values to its input ports. That is, it might be desirable to restrict the flow of data to each input port to a single value. After a value is sent to the port, the port shuts down and refuses to accept any more values. Subsequent values thrown to that port have no effect.

This special kind of port is called a *single-port*, and is made into a first-class object by the construct `call/sp` (meaning *call-with-current-single-port*). The usual port, one to which many values can be sent, is therefore called a *multi-port*.

In general, each sub-expression in a Pscheme program corresponds to a port which is implicitly multi-entrant if it is not made single-entrant by `call/sp`.

2.2.4 Pscheme Thread Creation and Termination

In Pscheme, threads (processes) are created by the `pcall` construct, as described above. In addition, the `throw` construct, which sends a value to a port explicitly, sometimes also

creates a thread. Thread termination is specified by the `die` construct, which terminates the current thread's computation.

`(throw e1 e2)` evaluates expressions `e1` and `e2`, resulting in a port p and a value v respectively, sends v to p, and returns v. If p is a closed single-port, then the throw has no effect other than to return v. However, if p is a multi-port or an open single-port, a new thread is created if the receiver is not blocked waiting for values from other ports. Otherwise, v is queued within p. [1] In either case, the `throw` expression returns the value v, and the current thread continues. [2] `throw` is an atomic action in the sense that if a process throws one value after another to the same port, the arrival order is the same as the sending order.

`(die)` terminates the computation of the current thread and returns no value. A `(die)` in a parent process does not kill its child processes. Here is an example of the typical use of a single-port: to commit to the first value computed by competing parallel processes. For example, suppose a binary tree is encoded in the list form `(key left-child right-child)`. To find an element with a key value in this tree, a parallel search along all branches can be performed as shown figure 2.2. The first value passed to the answer port is the value that `find` returns.

2.2.5 Exclusive Functions

As described above, a new activation of a function is created each time there is a value available on each of its input ports. Thus, this is a method for creating multiple invocations of the same function in parallel.

[1] For example, a new thread cannot be spawned if the value is passed to a port associated with a function argument position, and it has to synchronize with other arguments in order to apply the function.

[2] Notice that our definition of `throw` is different from those in sequential languages supporting continuations. Our `throw` does not transfer control, but tries to spawn a new thread.

```
(define (find1 x tr port)
        (cond ((null? tr) #f)
              ((eq? x (car tr)) (throw port #t) (die))
              (else (pcall (lambda (a1 a2) #f)
                           (find1 x (cadr tr) port)
                           (find1 x (caddr tr) port)))))

(define (find elt tree)
  (call/sp (lambda (p) (find1 elt tree p))))
```

Figure 2.2: Finding an element in a tree in parallel

It might be the case, however, that it is desirable for there to be only one invocation of the function active at any given time. Such a function is said to be *exclusive* (and has essentially the same behavior as a Hoare's monitor). An *exclusive function* is desirable, for example, if it modifies some data structure that requires mutually exclusive access. Another (related) possibility is that the function produces output or results whose order is important.

The Pscheme construct exclusive is used to create exclusive functions. The expression

```
(exclusive e)
```

evaluates `e` to some function value f, and returns a function f' with the same behavior as f except that f' is exclusive. When f' is called, subsequent requests to call f' are blocked and queued until the current invocation of f' returns or a `(die)` is executed during the evaluation of the current invocation of f'.

Consider the program in figure 2.3 for comparing the fringes of two trees. The parallelism between the searching of the two trees is desirable. However, an exclusive version of compare is essential to avoid a race condition in which compare reports success because

```
(define (samefringe t1 t2)
  (call/sp (lambda(p)
             (pcall (exclusive compare)
                    (call/mp (lambda(p1) (search t1 p1) nil))
                    (call/mp (lambda(p2) (search t2 p2) nil))))))

(define (search tree outport)
  (cond ((null? tree) nil)
        ((atom? tree) (throw outport tree))
        (else (search (car tree) outport)
              (search (cdr tree) outport))))

(define (compare a b)
  (cond ((null? a) (null? b))
        ((eq? a b) (die))
        (else nil)))
```

Figure 2.3: Checking if two trees have the same fringe in parallel. In this program, `nil` is not considered as a leaf but a null child link.

the right most leaves of the two trees are identical, even though some of the interior leaves (which may be different in the two trees) have not yet been compared. The `call/sp` in samefringe avoids the problem of success being reported after failure has already been reported.

`(exclusive f)` returns a function `f'` which can have only one activation at any time. However, during the evaluation of `f'`, exclusiveness can be violated if pcall is invoked, or a multi-port is captured inside of `f'` and gets thrown a value. The language definition only enforces exclusiveness of the entry point of an exclusive function. The effect of creating parallel threads executing the body of an exclusive function is left unspecified by the language definition and should be avoided.

2.2.6 Further Notes about Definitions

We have introduced all constructs regarding parallelism in Pscheme. All these primitives have to appear in the forms as described. None of them is a first-class function. For example, `(pcall throw e1 e2)` and `(exclusive call/mp)` are both illegal expressions. In other words, `pcall`, `throw`, `call/mp`, `call/cp`, `exclusive` and `die` are all special terms.

The concept of threads needs further explanation. A thread is the execution of a Pscheme expression. When a throw happens, say `(throw p v)`, the *current thread* continues by taking the return value `v` of the throw expression. A *new thread*, if necessary, is created to execute the context that the port `p` represents. This is considered a side effect of the the throw expression. Therefore, in the body of an exclusive function `f`, there can be a throw of an object to a port outside of the body, and does not violate the exclusive property of `f`. There is still only one thread executing the function body.

With the existence of multiports, codes become re-entrant, and expressions could

return values more than once. However, at the top level, a program should give its answer only once. Top-level parallelism, where a program keeps returning values from multi-port outputs, is not allowed. In other words, any top-level expression (the expression that represents the whole program) can be viewed as implicitly associated with a single-port. Whenever a program returns its result, the whole computation stops even if there are still running threads. This is also guaranteed in our formal semantics.

On the other hand, a program might not return anything because a `die` *blocks* all threads at the top level. Such programs are considered *non-terminating*.

2.3 The Relationship Between Ports and Continuations

Although we introduced `call/mp` in the context of pcall, they are really orthogonal constructs. We saw above that pcall is useful by itself, and is actually a very common parallel construct. What is the effect of using `call/mp` without using `pcall`? Consider the expression

```
(f (call/mp (lambda (p1) a))
   (call/mp (lambda (p2) b))
   (call/mp (lambda (p3) c)))
```

Because the evaluation of expressions a, b, and c occurs sequentially, the illustration used in figure 2.1 would be misleading for this example. Assuming a left-to-right evaluation order among function arguments, the evaluation order is clearly seen when the expression is represented in continuation passing style (CPS). Assuming k is the continuation for the entire expression, the cps-converted expression would be

```
(a' (lambda (v1) (b' (lambda (v2) (c' (lambda (v3) (f' v1 v2 v3 k)))))))
```

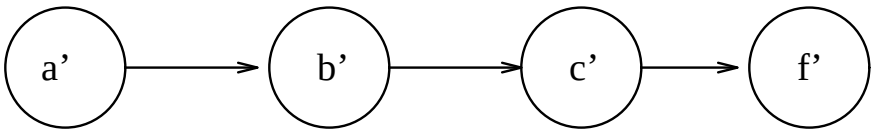


Figure 2.4: A sequential representation of function application using CPS

where `a'`, `b'`, `c'`, and `f'` are the cps-converted versions of `a`, `b`, `c`, and `f`, respectively.

This might be illustrated graphically as in figure 2.4, Each continuation is a procedure with a single input port. When a value is sent to the port, the continuation is invoked. This is exactly the behavior caused by the capture of the continuation using `call/cc` and the invocation of that continuation. This is yet another reason that we are convinced that ports are the logical extension of continuations in a parallel Scheme.

In a sequential computation, such as the one pictured above, each function has only one input multi-port. When a value is thrown to that port, a new thread computing the function call is created. This is not the case when `call/cc` is used in a sequential language. Thus, one way to simulate the effect of invoking a continuation by throwing to a port is to kill off the throwing process. Specifically, the expression

```
(call/cc f)
```

in ordinary (sequential) Scheme is equivalent to

```
(call/mp (lambda (p) (f (lambda (v) (throw p v) (die)))))
```

in Pscheme.

Pscheme provides all features in Scheme except `call/cc` and first class continuations. When a Pscheme expression contains none of the six parallel constructs described above, its semantics is the same as Scheme's. That is, expressions are evaluated in applicative order, and the relative evaluation order among the arguments in a function call is unspecified.

As mentioned above, Pscheme's view of computation is a dynamic data-flow diagram. Data streams may flow between nodes as thread execution is ordered through exclusive functions. In Scheme or other sequential applicative languages, the sequential execution is actually a depth first search with a post order traversal of the data-flow diagrams (this is what the CPS transform does). First class continuations enable us to jump anywhere in the traversal. On the other hand, in Pscheme, first class ports allow us to throw elements to any link, or disconnect a link (close a single port), which gives us more control of the data-flow diagram.

Chapter 3

Programming in Pscheme

In this chapter, we show how to program in Pscheme. Examples will explain how ports and exclusive functions can represent synchronization, blocking, pipelining style parallelism and ordering.

Through various programming styles, we have built higher level control facilities such as semaphores, Ada-like rendezvous and Multilisp futures. Parallel data structures such as shared atomic-accessed variables and FIFO queues have also been implemented. From theses examples, we can see that Pscheme gives us various aspects of parallelism, and enables us to apply different programming paradigms in one language.

The chapter is also an experiment to see the feasibility of, and limits to, representing control in a parallel applicative language. We claim that ports provide a control mechanism as general as continuations in sequential languages. Our experience also suggests that they are easily understood by a programmer familiar with call/cc and continuation-based codes.

```
(define (sum m n)
 (cond ((eq? m n) m)
       ((eq? n (+ 1 m)) (+ m n))
       (else (let ((x (/ (+ m n) 2)))
                (pcall + (sum m x) (sum (+ 1 x) n))))))
```

Figure 3.1: Parallel sum

3.1 Divide and Conquer – Horizontal Parallelism

A typical case of parallel execution is to evaluate argument expressions of a function call in parallel, that is, to exploit horizontal parallelism. Problems that can be solved with a recursive divide-and-conquer algorithm can usually be parallelized in this way. For example, computing the sum of all numbers from 1 to n (as in figure 3.1), or sorting a list of numbers with the quicksort algorithm (as in figure 3.2).

Horizontal parallelism is often seen in other parallel languages. Programs with horizontal parallelism are easy to understand and are usually used to measure the performance speedup.

3.2 Using `pcall` and Ports as a Synchronizing Mechanism

Divide and conquer programs, as shown above, use `pcall` to create parallel threads. But at the same time, `pcall` also creates *synchronous multiple ports* so that subsequent pairs of arguments can synchronize and reactivate the function body evaluation. The samefringe example in chapter 2 shows such use of `pcall`. This enables us to build synchronizing

```
(define (qs num-list)
  (if (null? num-list)
      nil
      (if (null? (cdr num-list))
          num-list
          (let ((pivot (car num-list)))
            (let ((left-right (partition pivot (cdr num-list) nil nil)))
              (pcall append
                     (qs (car left-right))
                     (cons pivot (qs (cdr left-right)))))))))


(define (partition p l l-buf r-buf)
  (if (null? l)
      (cons l-buf r-buf)
      (letrec ((x (car l)))
        (if (< x p)
            (partition p (cdr l) (cons x l-buf) r-buf)
            (partition p (cdr l) l-buf (cons x r-buf))))))
```

Figure 3.2: Quicksort

facilities using `pcall`.

Ada's rendezvous is a synchronous communication in that the sender and the receiver of a message have to synchronize before further execution. The rendezvous is itself a function call with the arguments passed by the sender. The return value of the function call is actually the return value of the rendezvous expression for both the sender (the entry call expression) and the receiver (the accept expression).

We can easily build an Ada-like rendezvous facility in Pscheme. The expression `(make-rendezvous f)` returns what we call a *rendezvous* object `r` to serve as an entry of a task in Ada. Tasks communicating through a rendezvous in Ada are just Pscheme sequential threads in which the rendezvous object is visible. A thread evaluating `((r 'send) argument-list)` causes `f` to be applied with `argument-list`. It synchronizes with the expression `(r 'accept)` in another thread. Both the send and the accept expression returns the result of applying `f` with `argument-list`.

Implementation of rendezvous is shown as in figure 3.3. The idea is for the message sender and receiver to capture their current single-ports and pass them to a pair of synchronizing input ports to a `pcall`'ed function, which evaluates `(apply f arguments-list)` and then passes the result back to the sender and receiver respectively.

The `(make-rendezvous f)` expression does everything. It defines the `send` and `accept` operations in the binding of `rendezvous`, and it creates the synchronizing mechanism by `pcall`'ing a function that takes the sender's thunk and receiver's current port as arguments. This function simply invokes the thunk and sends the result back to the receiver. What the thunk does is to apply f with arguments provided by the sender, and then pass the result back to sender. The `pcall`'ed function never returns because only the threads for the sender and the receiver need to be reactivated.

We also utilize the pcall mechanism to return the created rendezvous object to where

```
(define (make-rendezvous f)
  (let ((port1 nil) (port2 nil))
    (let ((rendezvous
            (lambda (op)
              (if (eq? op 'send)
                  (lambda (arg)
                    (call/sp (lambda(p)
                               (throw port1 (lambda() (throw p (f arg))))
                               (die))))
                  ;else op = 'accept
                  (call/sp (lambda(p) (throw port2 p) (die)))))))
    (call/sp
      (lambda (result)
        (pcall (lambda (sender-thunk receiver-port)
                 (throw receiver-port (sender-thunk))
                 (die))
               (call/mp (lambda(p) (set! port1 p) (lambda() rendezvous)))
               (call/mp (lambda(P) (set! port2 p) result)))))))



> (let ((r (make-rendezvous (lambda(x) x))))
    (pcall cons
           (r 'accept)
           (begin ...  ((r 'send) 1) ... 2)))

==> (1 . 2)
```

Figure 3.3: Implementation of Ada-like rendezvous

`make-rendezvous` is called by wrapping the rendezvous object in a thunk and passing the thunk to the current single port of the `make-rendezvous` expression.

The short example in figure 3.3 shows how to perform synchronous communication among threads in Pscheme.

3.3 Blocking and Locking

In Pscheme, the basic blocking mechanism is provided by exclusive functions. As has been stated in previous chapters, it helps control the ordering of threads or data elements. It also provide exclusive access to shared variables. To guarantee mutual exclusion, an alternative approach is to use semaphores to protect critical sections.

In Pscheme, binary semaphores can be implemented. That is, only one thread can execute the critical section. Subsequent access requests (that is, the "P" operations) will be queued up and satisfied later. To implement binary semaphores in Pscheme, the idea is also to utilize `pcall`, `call/sp` and `call/mp`.

In figure 3.4, `(make-semaphore)` returns a semaphore `s` such that a thread evaluating `(s 'p)` performs the "P" operation on `s`, and returns the symbol `'dontcare`. The call `(s 'v)` performs the "V" operation and returns `'dontcare` as well.

`s` is created by `pcall`'ing a function `f` of two parameters `next-process` and `release-signal`. What the function `f` does is simply to invoke the thunk `next-process`. Any thread evaluating `(s 'p)` captures its current single-port `p`, passes `p` to `entry-port` (the port associated with `next-process`), and then dies. Evaluating `(s 'v)` throws `'dontcare` to `exit-port` (the port associated with `release-signal`), and thus synchronizes with the requesting thread (the thread evaluating `(s 'p)`) through the `pcall`'ed function `f`. `s` is initialized by having the `'dontcare` symbol queued in `exit-port` so that the first thread calling `(s 'p)` will not block.

```
(define (make-semaphore)
 (let ((entry-port nil) (exit-port nil))
  (let ((semaphore
         (lambda(op)
           (if (eq? op 'p)
               (call/sp
                 (lambda(p)
                   (throw entry-port
                          (lambda() (throw p 'dontcare) (die)))
                   (die)))
               ;else op = 'v
               (throw exit-port 'dontcare)))))
   (pcall (lambda (next-process release-signal) (next-process))
          (call/mp (lambda(p) (set! entry-port p)
                              (lambda() semaphore)))
          (call/mp (lambda(p) (set! exit-port p)
                              (throw exit-port 'dontcare)))))))
```

Figure 3.4: Implementation of semaphores

Both semaphores and exclusive functions provide exclusive access. But, in our current implementation, exclusive functions cause busy-waiting blocking, while semaphores cause back-off blocking, since threads making access requests have to idle themselves before sending their current ports to the request handler. In the case of a large critical section and high contention, semaphores perform better than exclusive functions. On the other hand, exclusive functions should be light weight activations in order to perform well. Our current implementation puts each created thread in a global shared ready queue, and several virtual processors will execute them. If this model is not on a platform that supports fine grain parallelism, the back-off blocking will take some turn-around time, but the total running time should be close to the busy-waiting model. When we discuss our implementation in chapter 6, we will talk about this in more detail.

3.4 Shared Mutable Objects and Atomic Assignments

Scheme is not a purely functional language because any variable can be modified through the use of `set!`. In the parallel extension Pscheme, the non-atomic `set!` on a shared mutable variable may cause nondeterministic results. It is usually suggested that programmers are responsible to guarantee atomic access on shared variables in a parallel language (e.g. by using locks explicitly), but it is also usually desirable for a language to provide parallel data structures with atomic access operations. For the case of shared mutable variables, we need atomic reads and writes.

One way to implement shared mutable variables in Pscheme is to use exclusive functions to handle all update requests. The simplest approach is to devote an exclusive function for each shared mutable variable as in the first implementation shown in figure 3.5. Of course, if a shared variable is guaranteed to be updated only in a piece of

exclusive codes, `set!` can be used directly.

The other way is to use the pcall mechanism to serialize all update requests as is done in the semaphore implementation. The second implementation in figure 3.5 illustrates this idea. The current value is stored in the value port, and the access requests are queued within the request port. The request is handled by the function `monitor`. It either passes the current value v to the read requester and restores v back to the current value port, or stores a new value v' in the current value port. The identifications of the request port and the current value port are not stored globally. Instead, they are passed as parameters each time an access request is issued.

The interesting thing about doing this is that the implementation needs no `set!` or any assignment operator, and can be used to represent assignment in languages not supporting assignment. It demonstrates the fact that ports are expressive enough to describe assignment. However, it also shows the fact that a control mechanism based on ports and threads violates referential transparency in parallel programs as is the case will continuations in sequential programs.

As shown in the figure 3.5, `(make-cell val)` creates a cell containing a request port and a value port. A read request is of the pattern `('read requester-current-port value-port)`, and an assign request is of the pattern `('assign new-value value-port)`. On a read request, the cell passes the current value to the requester's current port as well as the value port. On an assign request, the cell simply passes the current value to the value port.

```
(define (read x) (x 'read))                ; x is a cell
(define (assign x v) ((x 'assign) v))

; first implementation of stores with atomic accesses

(define (cell v)
  (exclusive
    (lambda(op)
      (case op
       ((read) v)
       ((assign) (lambda (new-v) (set! v new-v)))))))


;second implementation of stores with atomic accesses

(define (make-cell val)
 (let ((monitor
         (lambda (p1 p2)
           (if (list? p1)                ;handle access requests
               (if (eq? (car p1) 'read)  ;p1= request, p2= current value
                   (begin (throw (cadr p1) p2)
                          (throw (caddr p1) p2)
                          (die))
                   ;else (car p1) = 'assign
                   (begin (throw (caddr p1) (cadr p1))
                          (die)))
               ;else create the monitor and return
               (lambda(op)
                 (case op            ; p1 = request port, p2 = value port
                  ((read)
                   (callsp (lambda(p) (throw p1 (list 'read p p2))
                                      (die))))
                  ((assign)
                   (lambda(v) (throw p1 (list 'assign v p2)) v))))))))
   (pcall monitor
          (callmp (lambda(p) p))                    ;create request port
          (callmp (lambda(p) (throw p p) val)))))  ;create value port and
                                                    ;initialize with val
```

Figure 3.5: Two implementations of stores supporting atomic access

3.5 Shared FIFO Queues

Pscheme supports order preservation of messages, data elements or thread execution [1] through first class ports. The ordering is guaranteed implicitly because of the nature of ports. But, sometimes it may also be desirable to provide explicit ordering, that is, FIFO queues with atomic access that can be shared by multiple threads. Scheme's management of data structures is mostly list-based, so are most applicative languages providing list constructors (such as ML [50], Haskell [37], etc.) because the LIFO feature fits well in recursive programming. However, if Pscheme is to support other parallel programming paradigms, lists or cons-cells may not be sufficient for a program that, for example, processes a stream of data and stores them in the order they arrive. This motivates our implementation of shared FIFO queues.

The idea is similar to the implementation of atomic assignments, that is, to use exclusive functions to serialize requests. The internal structure of a FIFO queue is a linked list with the links implemented with free variable pointers in a closure. (Pscheme does not provide `set-car!` and `set-cdr!`.) Enqueue and Dequeue are both constant time operations containing several function calls. Since this is only a sequential FIFO mechanism under the protection of `exclusive`, we may use it without `exclusive` as long as it is guaranteed to be accessed only by exclusive functions.

In fact, queues are often used in stream-based programs as a local data structure of a particular exclusive function which processes streams. Later on in following sections, we will see larger examples using the FIFO queues to store elements in FIFO order.

[1] For example, keeping throwing values to a particular multiport creates *waves* of threads executing the same code. These threads can be ordered through the use of exclusive functions.

```
(define (new-q)
 (let ((hd nil) (tl nil))
  (exclusive
    (lambda (op)
      (case op
        ((deq) (let ((elt (hd 'val)))
                 (set! hd (hd 'next))
                 elt))
        ((enq) (lambda (v)
                 (let ((new (new-q-elt v nil)))
                   (if (null? hd)               ;queue empty
                       (set! hd (set! tl new))  ;initialize with 1st elt
                       (begin
                         ((tl 'set-next) new)
                         (set! tl new))))))
        ((empty?) (null? hd))))))


(define (new-q-elt val next)
  (lambda(op)
    (case op
      ((val) val)
      ((next) next)
      ((set-next) (lambda(v) (set! next v))))))


(define (dequeue q) (q 'deq))         ; return fisrt element
(define (enqueue q v) ((q 'enq) v)    ; return v
(define (queue-empty? q) (q 'empty))  ; return #t or #f
```

Figure 3.6: Implementation of shared FIFO queues

3.6 Futures

Multilisp's future is a powerful construct that allows programmers to specify parallelism in very fine granularity. Programmers have sufficient control of what jobs are to spawn for parallel execution without worrying about how to get back the result of the spawned task.

Specifically, in multilisp, `(future exp)` spawns a new process evaluating `exp`, and returns a future object, which, when passed to a strict function, gives the evaluated value v of `exp`. If the evaluation of exp is not complete, the function waits for its completion. The future object will always have the value v once the evaluation of `exp` is complete (supposing no interaction with continuations).

In our Pscheme implementation of futures, the value of a future object needs to be explicitly retrieved by applying the function `future-val` to the future object. Figure 3.7 shows the implementation.

`(future exp)` initiates a new thread evaluating exp, and returns a future object f, which can be stored or passed to other functions. A strict operator needing `exp`'s value v can call `(future-val f)`, which returns v immediately if the evaluation of exp has completed, or blocks until v is available. Once the evaluation of `exp` is complete, the result value is committed as the value of this future object.

Similarly, a future object can be created by `pcall`'ing a function f of two parameters `val` and `req`, representing the future value and the value requester's current port, respectively. Before the future value v is available, there might be several requests for it, so f performs `(throw req val)`, and then throws `val` back to the value port to synchronize with subsequent requests. The short example in figure 3.7 shows how to speed up computation in call-by-value languages.

```
(def-syntax (future exp)
  (make-future (lambda() (call/sp (lambda(p) exp))))   ; not re-entrant

(define (future-val future-object) (future-object))

(define (make-future thunk)
 (call/sp
   (lambda (return)
    (letrec
      ((value nil) (touched? nil) (val-port nil) (req-port nil)
       (future-obj (lambda()
                     (if touched?
                         value
                         (call/sp (lambda(p) (throw req-port p) (die)))))))
      (pcall (lambda (val req) (throw req val) (throw val-port val) (die))
             (call/mp (lambda (p) (set! val-port p)
                                  (set! value (thunk))
                                  (set! touched? #t)
                                  value))
             (call/mp (lambda (p) (set! req-port p)
                                  (throw return future-obj)
                                  (die)))))))))



> (let ((x (future (largest-prime-less-than 1000))))
    ( .... (future-val x) ...)
```

Figure 3.7: Implementation of Multilisp futures

3.7 Stream-based Programming and Vertical Parallelism

Most parallel applicative languages exploit parallelism through a fork-join structure such as Pscheme's `pcall` demonstrates. It is sometimes necessary for a program to specify vertical parallelism, a parallelism between a producer and a consumer. For example, we may want to evaluate the function body and the function's argument expression in parallel if the argument value is not needed immediately by the body. Futures, as mentioned above, can satisfy our need. In Pscheme, we may say

```
((lambda(x) .... (future-val x) ...) (future exp))
```

to gain vertical parallelism.

Furthermore, we can generalize this idea to express pipelining style parallelism. This is usually hard to express in most parallel applicative languages. Even with futures provided as in Multilisp, we still need exclusive access facilities to control the order of racing threads, but exclusive functions are not supported in Multilisp.

On the other hand, pipelining is a kind of parallelism that is often seen in real world activities. Although there are attempts to model streams in applicative languages [19], most are only sequential simulation of data streams, and fail to really provide pipelining parallelism in a parallel language. The reason may be that current Von Neumann architectures force the implementation of pipelining parallelism to be *simulated* by light weight processes, which is not considered to have good enough performances. We think that if massive fine-grained parallelism or data-flow computing is supported by the underlying platform, it is worth trying to express pipelining at the language level. In this section, we show how to develop stream-based programs with pipelining-style parallelism in Pscheme, and demonstrate another dimension of its expressiveness.

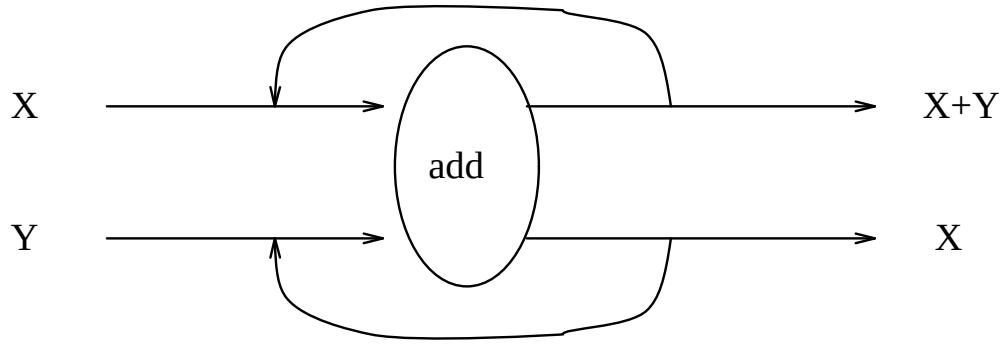


Figure 3.8: A data-flow diagram for computing the Fibonacci stream

3.7.1 A Small Example – Stream of Fibonacci Numbers

The idea of representing computation in terms of streams comes from the data-flow languages such as VAL [1], Id [7], Lucid [63] and so on.

For example, to calculate Fibonacci numbers in Lucid, we observe that the infinite Fibonacci stream `f` satisfies the recurrence equation:

```
f = 1 :: 1 :: (add_list f (cdr f))
```

The stream `f` can be generated by an adding cell with feedback wires to its two inputs as shown in Figure 3.8.

In Pscheme, the "stream" data type is not supported. In order to model streams in Pscheme, multiports are used. A stream is expressed by throwing data elements to a multiport. Figure 3.9 shows how to interpret the conceptual model of computing the Fibonacci stream in figure 3.8.

The input arcs are captured by multi-ports in Pscheme, and the adding cell is simply a function which also counts the cycle number n in order to compute up to $fib(n)$. Since the two input ports are not visible inside `add`, we use two global variables `next-x` and `next-y` to store the captured ports. The initial values (the two 1's) are sent to the ports and added together after `next-x` and `next-y` are initialized properly. `add` only returns

```
(define (fib n)
  (let ((counter 1) (next-x nil) (next-y nil))
    (let ((add (lambda (x y)
                 (if (eq? n counter)
                     x
                     (begin (set! counter (+ 1 counter))
                            (throw next-y x)
                            (throw next-x (+ x y))
                            (die))))))
      (pcall add
             (call/mp (lambda(p) (set! next-x p) 1))
             (call/mp (lambda(p) (set! next-y p) 1))))))
```

Figure 3.9: Implementation of the Fibonacci stream

the sum when the counter reaches n. Otherwise, it dies. Notice that `die` does not have to be exclusive because the barrier synchronization caused by `pcall` naturally enforces the exclusiveness of the application of `add`.

The use of `set!` in `fib` is awkward and unnecessary. The state of the computation that `counter` represents can be passed as parameters as are `x` and `y`. Therefore, we can have a *more functional* adding cell as shown in figure 3.10.

Streams are often processed through multiple stages. In the following sections, we will show larger examples that illustrate pipelining style parallelism.

3.7.2 Filters

Basic stream programming applies a few processing stages to an input stream and generates a result stream. These stages work concurrently in the pipelining model, and serve as filters of their inputs. We give examples in this section of how to implement filters and data streams to express pipelining style parallelism.

```
(define (fib n)
  (let ((next-x nil) (next-y nil) (next-c nil))
    (let ((add (lambda (x y counter)
                 (if (eq? n counter)
                     x
                     (begin (throw next-c (+ 1 counter))
                            (throw next-y x)
                            (throw next-x (+ x y))
                            (die))))))
      (pcall add
             (call/mp (lambda(p) (set! next-x p) 1))
             (call/mp (lambda(p) (set! next-y p) 1))
             (call/mp (lambda(p) (set! next-c p) 1))))))
```

Figure 3.10: Implementing the Fibonacci stream without using `set!`

Since Pscheme is a call-by-value language, a filter receiving an input stream will be represented as a one-argument function. Each activation of the function is the process an element from the stream. Filtering a stream requires reactivating the function for each stream element by throwing the element to the port associated with the function argument position.

The Relationship between Lists and Streams

The input stream is often generated from data structures such as lists, and the output stream is collected and stored into data structures as well. Since the filter function does not know when there is no element coming (it is decided by the caller or thrower), a special token is often passed to the function to signal the end of the stream.

Figure 3.11 shows how lists and streams of numbers can be transformed to each other via the function `list->stream` and `stream->list`. `(stream->list eos)` returns

```
(define (list->steam l receiving-port eos)
  (if (null? l)
      (begin (throw receiving-port eos) (die))
      (begin (throw receiving-port (car l))
             (list->stream (cdr l) receiving-port eos))))

(define (stream->list eos)
  (let ((buf nil))
    (exclusive
      (lambda(v) (if (eq? v eos)
                     buf
                     (begin (set! buf (cons v buf)) (die)))))))
```

Figure 3.11: Conversion between lists and streams

a stream-to-list converter, which is a one-argument exclusive function that takes stream elements as its argument and stores them in its internal buffer. (The stream ends with `eos`.) This converter finally returns a list of elements in the reverse order.

Examples – Generating Prime Numbers

We compute the prime numbers within a given range by creating an exclusive function which filters all numbers from the input stream that is a multiple of previous elements in the stream. Any application program that works on primes can take as input the stream generated by `prime-cell`.

If we just want to collect all primes less than or equal to n, we may use `stream->list` to return the prime list as shown in figure 3.12.

In traditional applicative call-by-value languages, the list of primes in some given range will only be available when all elements have been computed. Pscheme lifts the restriction. The first prime is available for use even when the second prime is still being

```
(define (num-stream a b port)
  (if (> a b)
      (begin (throw port 'eos) (die))
      (begin (throw port a) (num-stream (+ 1 a) b))))

(define (multiple? val num-list)         ; to see if val is a multiple of
  (if (null? num-list)                   ; a number in num-list
      #f
      (if (eq? (modulo val (car num-list)) 0)
          #t
          (multiple? val (cdr num-list)))))

(define (prime-cell out-port)
  (let ((prime-num-list nil))
    (exclusive
      (lambda(val)
        (if (eq? 'eos val)
            (begin (throw out-port 'eos) (die))
            (if (multiple? val prime-num-list)
                (die)
                (begin (throw out-port val)
                       (set! prime-num-list (cons val prime-num-list))
                       (die))))))))


(define (primes n)        ; return a list of primes <= n
  ((stream->list 'eos)
   (call/mp (lambda (out-port)
              ((prime-cell out-port)
               (call/mp (lambda (p) (num-stream 2 n p))))))))
```

Figure 3.12: Generating the stream of prime numbers

computed. We can achieve finer grained pipelining by splitting the testing of multiples into stages, each of which tests if the input number is a multiple of one number. If the underlying platform supports fine grain parallelism, this is worth trying.

3.8 Applications

We present in this section two larger examples of stream-based programming – quicksort and merge sort. The idea is to use any available result as early as possible.

3.8.1 Quicksort

Quicksort is a recursive function which partitions its input list in two, sorts them recursively, and appends them to get the result. We try to parallelize the partition and the two sub-sorts so that once a number less than the pivot is found, it is passed to the sorter for numbers less than the pivot. Numbers larger than the pivot are handled similarly.

The whole sorting process is a number of partitions that internally build up a binary search tree (BST). Our program views it as a binary tree of concurrent partitioning cells. Each cell passes the input from its parent to one of its two children depending on whether it is less or larger than the pivot. When the input stream comes to an end, we perform a depth first search with in order traversal of this BST based on the pivot value of each partitioning cell.

The partitioning cells in this scenario are exclusive functions in Pscheme. The *outgoing arcs* of a cell are actually ports visible in the exclusive function. Cells are created only when needed. When a cell is created, an input arc to the cell is also built as a multiport visible in its parent cell.

Figure 3.13 shows the code for our quicksort. As seen in the function `qs`, the sorting starts with one partitioning cell. The end-of-stream token is actually a port where the

```
(define (new-partition-cell pivot port1 port2)
  (exclusive
    (lambda (v)
      (if (number? v)
          (if (< v pivot)
              (if (null? port1)
                  ((new-partition-cell v nil nil)
                   (call/mp (lambda (p) (set! port1 p) (die))))
                  (throw port1 v))
              (if (null? port2)
                  ((new-partition-cell v nil nil)
                   (call/mp (lambda (p) (set! port2 p) (die))))
                  (throw port2 v)))
          ;else v is a port, so partition is done, now collect results
          (throw v (lambda ()  ; delay result collection
                     (pcall
                       (lambda (th1 th2)
                         (append (th1) (cons pivot (th2))))
                       (call/sp (lambda (p)
                                  (if (null? port1)
                                      (lambda () nil)
                                      (begin (throw port1 p) (die)))))
                       (call/sp (lambda (p)
                                  (if (null? port2)
                                      (lambda () nil)
                                      (begin (throw port2 p) (die))))))))))
      (die))))

(define (qs l)
  (if (null? l)
      nil
      (let ((partition-cell (new-partition-cell (car l) nil nil)))
        ((call/sp
           (lambda (result-port)
             (partition-cell
               (call/mp (lambda (p)
                          (list->stream (cdr l) p result-port))))))))))
```

Figure 3.13: Stream-based quicksort

result should go. When a cell receives a non-number element, it knows it is a result port. Then, it prepares to pass the sorted result (a list) to the result port by throwing two newly created receiving ports to its children respectively. After receiving two sorted lists from its children, the cell appends then and passes the whole sorted list to its parent cell.

We implement the result-collecting protocol a little differently from the above idea. That is, the result is collected lazily. A thunk is always passed to a result port, and calling it gives the real sorted list. The reason to do this is to keep exclusive functions light, and avoid parallelism or a "context switch" (that is, a `die`) in the middle of exclusive functions' bodies.

3.8.2 Merge Sort

Another application of pipelining-style parallelism is the merge sort. It is natural to think of a cell merging two sorted input streams and generating a sorted output stream. We use a recursive function `ms` to sort a vector of numbers as shown in figure 3.14.

When a merging cell is created in `ms`, it merges two streams whose lengths differ by at most one from each other so that `pcall` works. Since the `pcall` expression forces the merging cell to take numbers from both input ports, we buffer the larger one in one of the two local queues for subsequent comparisons.

Figure 3.15 shows the implementation of a merging cell. In a merging cell, there are two local queues to store elements from the two input streams respectively. At least one of them must be empty. The function call `(compare-deq value queue port all)` keeps comparing `value` with numbers in `queue`, throwing the smaller to `port` until `val` is thrown or `queue` is empty. If the flag `all` is true, everything will be thrown to `port`.

```
(define (merge-sort vct)
  ((stream->list 'eos)
   (call/mp (lambda(p) (ms vct 0 (- (vector-length vct) 1) p)))))

(define (ms num-vct i1 i2 out-port)
  (cond ((eq? i1 i2)
         (throw out-port (vector-ref num-vct i1))
         (throw out-port 'eos)
         (die))
        ((eq? i2 (+ 1 i1))
         (let ((n1 (vector-ref num-vct i1))
               (n2 (vector-ref num-vct i2)))
           (if (< n1 n2)
               (begin (throw out-port n1) (throw out-port n2))
               (begin (throw out-port n2) (throw out-port n1)))
           (throw out-port 'eos)
           (die)))
        (else
         (let ((i3 (/ (+ i1 i2) 2)))
           (pcall (new-ms-cell out-port)
                  (call/mp (lambda(p) (ms num-vct i1 i3 p)))
                  (call/mp (lambda(p) (ms num-vct (+ i3 1) i2 p))))))))


; auxiliary functions used by new-ms-cell

(define (deq-all q p)
  (if (q 'empty?)
      nil
      (begin (throw p (q 'deq)) (deq-all q p))))

(define (compare-deq val queue port all)
  (if (queue 'empty?)
      (if all (throw port val) nil)
      (if (< val (queue 'head))
          (begin (throw port val)
                 (if all (deq-all queue port) nil))
          (begin (throw port (queue 'deq))
                 (compare-deq val queue port all)))))
```

Figure 3.14: Stream-based merge sort

```
(define (new-ms-cell out-port q1 q2 state)
 (exclusive
  (lambda(n1 n2)
   (cond ((and (eq? n1 'eos) (eq? n2 'eos))
          (if (eq? state 1)
              (deq-all q1 out-port)
              (deq-all q2 out-port))
          (throw out-port 'eos))
         ((eq? n1 'eos)
          (if (eq? state 1)
              (compare-deq n2 q1 out-port #t)
              (begin (deq-all q2 out-port) (throw out-port n2)))
          (throw out-port 'eos))
         ((eq? n2 'eos)
          (if (eq? state 1)
              (begin (deq-all q1 out-port) (throw out-port n1))
              (compare-deq n1 q2 out-port #t))
          (throw out-port 'eos))
         ((null? state)            ; initial state
          (if (< n1 n2)
              (begin (throw out-port n1) ((q2 'enq) n2) (set! state 2))
              (begin (throw out-port n2) ((q1 'enq) n1) (set! state 1))))
         ((eq? state 1)
          ((q1 'enq) n1)
          (compare-deq n2 q1 out-port #f)
          (if (q1 'empty?)
              (begin ((q2 'enq) n2) (set! state 2))
              (die)))
         (else                     ; (eq? state 2)
          ((q2 'enq) n2)
          (compare-deq n1 q2 out-port #f)
          (if (q2 'empty?)
              (begin ((q1 'enq) n1) (set! state 1))
              (die))))
   (die))))
```

Figure 3.15: Implementing the cell merging two sorted streams

3.9 Remarks

What was the objective of this chapter? We have shown that Pscheme can model different kinds of parallel programming paradigms. For programmers familiar with continuation-based programs, understanding Pscheme programs is not harder because port operations (`call/sp`,`call/mp` and `throw`) are similar to continuation operations. As a matter of fact, it is easy to program in Pscheme with the help of our library of higher order control facilities and parallel data structures.

Our point, however, is not to develop programs that can beat those written in Multilisp, Lucid or Actors. Instead, we try to reason about control within the world of applicative programming, and illustrate the fact that ports are expressive enough to describe most concepts in parallel computation. Furthermore, the concept of ports can be used to explain the semantics of multi-threaded applicative languages. [2] We believe that ports do play the role that continuations do in sequential languages.

We would also like to point out that Pscheme does support the declarative programming style. Issues that are easily solved by, for example, functional programs, should be solved in that way. But, programming in Pscheme, we have more alternatives.

As for practical programming in Pscheme, we will give performance numbers for the two practical stream-based examples quicksort and merge sort to see how well they scale up on a Von Neumann machine.

[2] We use the concept of ports to develop Pscheme's operational semantics. This can be seen in chapter 4.

Chapter 4

Operational Semantics

In this chapter, we give a formal definition of Pscheme. The framework is an operational semantics that can be used to define any multi-threaded expression language with a global naming space. Our approach is similar to CML's operational semantics developed by Reppy [55] and Berry, Milner and Turner [8].

We first introduce an abstract reduction machine that illustrates our computation model. Then, we present Pscheme's operational semantics in terms of the reduction rules on our abstract machine. As CPS is used to define sequential languages, the reduction rules are presented in *port passing style*, which also constitutes the basis of Pscheme's compiler.

4.1 The Abstract Machine

Our abstract machine contains

- a *shared memory* which maps *addresses* to *values* in the language domain,
- a *shared queuing mechanism* which maps *queue addresses* to FIFO queues, and
- a *set of reduction operations* applicable to *threads* running on it.

At any time, the configuration of the abstract machine is represented by the triple:

(*set of running threads, shared memory, shared queuing mechanism*)

The semantics of a Pscheme program is defined by the initial and terminating configuration of the abstract machine with this input program.

The first component of the machine configuration is defined by a set of running threads. Each thread is expressed as a triple (expression, environment, port), where expression is a legal expression in Pscheme, environment is a mapping from identifiers in a Pscheme program to locations (addresses) in the shared memory, and port is a semantic abstraction to be defined later.

The instructions of the abstract machine are the reduction operations between machine configurations. At any time during the program execution, only one reduction step can occur, but the one to apply among all applicable reductions is selected nondeterministically.

4.2 Meanings of Semantic Notations

A thread [exp, env, port] has the meaning that a process is currently evaluating the expression exp under the environment env, and then passes the result to the port port. port is syntactically a series of operation tags separated by the symbol "::". A port can be reified [1] at the expression level, (and hence becomes an object in the language domain and can be manipulated by the programmer) and stored in the shared memory. The top-level port is denoted as RETURN, meaning the control point where a program is to return its final result. If a thread is denoted as $[\mathsf{val}, \mathsf{env}, \mathsf{port}]_{done}$, it means that it is about to *pass*

[1] The term *reify* comes from the concept of computational reflection [20,59,49] It means to make an invisible meta-level object (such as a construct used in the language interpreter or the semantic framework) available or tangible at the language level so that the programmer can manipulate it directly. For example, reifying a continuation in the semantics gives a one-argument function in the language.

K	$\in$	$Constant$
I	$\in$	$Identifier$
a, a_i	$\in$	$Address$
F, A, A_i, v, v_i	$\in$	$Expression$
ρ	$\in$	$Identifier \rightarrow Address$
M	$\in$	$Address \rightarrow Representation$
$P, < * >, < * >:: P$	$\in$	$Port$
Q	$\in$	$QueueAddress \rightarrow Queue$
q	$\in$	$Queue$
$[*], [*]_{done}$	$\in$	$Process$
S	$\in$	$powerset(Process)$
$S[*]$	$=$	$S \cup \{[*]\}$
$update$	$\in$	$((Address \rightarrow Representation) \times Address \times Representation)$ $\rightarrow (Address \rightarrow Representation)$
$update(M, a, v)$	$=$	$M - \{(a, Ma)\} + \{(a, v)\}$
$enqueue(Q, q, v)$	$=$	$Q - \{(q, Qq)\} + \{(q, enq(Qq, v))\}$
$dequeue(Q, qset)$	$=$	$Q - \{(q, Qq) \mid q \in qset\} + \{(q, deq(Qq)) \mid q \in qset\}$
$PRIM$	$=$	the set of all identifiers represting primitive functions

Figure 4.1: Notation definitions in Pscheme's operational semantics

the evaluated value val to the port port, and there is no need to evaluate val in env.

We use the notion $C1 \Longrightarrow C2$ to denote an atomic reduction step in which configuration $C1$ is transformed to configuration $C2$. As mentioned above, the system configuration will be changed through only one reduction rule at a time. If there are more than one reduction rules applicable to some system configuration, the rule to apply is chosen non-deterministically.

Figure 4.1 shows the notations (and their definitions) we will use in our reduction rules. We use the character "$*$" to match any content in the context. To simplify the configuration expression, we express the set of threads without the curly brackets "{}". Instead, for a thread set S, $S \cup \{[*]\}$ is expressed as $S[*]$. Also, representing the triple for the machine configuration is abbreviated without the parentheses "()". The definitions of

the function *enq*, *deq* and *powerset* in figure 4.1 are not provided for the obvious meanings of these names.

We define a series of reductions by the symbol $\Longrightarrow^*$ as follows.

$C \Longrightarrow^* C'$ iff $C = C'$ or $\exists C''\ \ C \Longrightarrow C''$ *and* $C'' \Longrightarrow^* C'$

Definition of a Pscheme Program

Let $eval(Exp)$ be the set of all possible results of evaluating the program expression Exp, and RETURN be the top-level output port. Then,

1. $v \in eval(Exp)$ iff

   $\exists \rho, S, M, Q,\ \ \{[Exp, \{\}, \mathsf{RETURN}]\}, \{\}, \{\} \Longrightarrow^*\ \ S[v, \rho, \mathsf{RETURN}]_{done}, M, Q$

2. $eval(Exp) = \perp$ iff

   $\neg\exists v, \rho, S, M, Q,\ \ \{[Exp, \{\}, \mathsf{RETURN}]\}, \{\}, \{\} \Longrightarrow^*\ \ S[v, \rho, \mathsf{RETURN}]_{done}, M, Q$

4.3 Operational Semantics of Pscheme

In this section, we present the reduction rules for Pscheme expressions on our abstract machine. This is a parallel extension of the sequential continuation-passing style semantics. The continuation abstraction in the sequential semantics is replaced by a port abstraction which is syntactically composed of a series of operation tags. We call it the *port passing style* semantics. As in the case of the continuation abstraction, the port abstraction also encapsulates the information regarding the environment and the computation that follows. Side effects caused by destructive assignments or parallel primitives are reflected in the shared memory or the shared queuing mechanism, which are part of the system configuration.

As stated in previous chapters, Pscheme views parallel computation as a dynamic data-flow diagram. Our port-passing-style semantics described here makes all links in

the data-flow diagram explicit. Evaluated values are passed explicitly along the links to receiving nodes just as a thread passes a value to its port abstraction in the operational semantics.

4.3.1 Basic Reductions Regarding Sequential Computation

Constants do not require evaluation and can be passed to ports directly. Evaluating an identifier requires looking it up in the environment to get the address where the value of the identifier is stored. Conditional expressions and assignment expressions have the same semantics as in CPS semantics. Primitive operators, if not bound to new values, are treated as first class closures as lambda forms are.

Constants and Identifiers

$$S[K, \rho, P]\ ,\ M\ ,\ Q \Longrightarrow S[K, \rho, P]_{done}\ ,\ M\ ,\ Q$$

$$S[I, \rho, P]\ ,\ M\ ,\ Q \Longrightarrow S[M(\rho I), \rho, P]_{done}\ ,\ M\ ,\ Q$$

Conditionals

$$S[(\texttt{if}\ e_1\ e_2\ e_3), \rho, P]\ ,\ M\ ,\ Q \Longrightarrow S[e_1, \rho, <\mathsf{branch}, e_2, e_3 >:: P]\ ,\ M\ ,\ Q$$

$$S[v, \rho, <\mathsf{branch}, e_1, e_2 >:: P]_{done}\ ,\ M\ ,\ Q$$

$$\Longrightarrow \begin{cases} S[e_1, \rho, P]\ ,\ M\ ,\ Q & \text{if } v = \texttt{nil} \\ S[e_2, \rho, P]\ ,\ M\ ,\ Q & \text{otherwise} \end{cases}$$

Shared Memory

$$S[(\texttt{set!}\ X\ Y), \rho, P]\ ,\ M\ ,\ Q \Longrightarrow S[Y, \rho, <\mathsf{assign}, X >:: P]\ ,\ M\ ,\ Q$$

$$S[v, \rho, <\mathsf{assign}, X >:: P]_{done}\ ,\ M\ ,\ Q \Longrightarrow S[v, \rho, P]\ ,\ update(M, \rho X, v)\ ,\ Q$$

Function Abstraction

$S[(\texttt{lambda}\ args\ body), \rho, P]\ ,\ M\ ,\ Q \Longrightarrow S[closure(args, body, \rho), \rho, P]_{done}\ ,\ M\ ,\ Q$

$S[F, \rho, P], M, Q \Longrightarrow S[primop(F), \rho, P]_{done}, M, Q \qquad$ if $F \in PRIM$, and ρF undefined

4.3.2 Exclusive Functions

A function can be made exclusive, that is, at most one activation of an exclusive function can exist at any time. When an exclusive function is called, a shared lock and a shared queue is created to guarantee the exclusiveness and to store the calling requests. At the same time, another thread is created to retrieve arguments (if any) in the queue and dispatch them to the function. The following rules show that the primitive `exclusive` can only take as its argument normal functions or primitive operators. No exclusive functions can be passed to `exclusive`.

$S[(\texttt{exclusive}\ exp), \rho, P],\ M,\ Q\ \Longrightarrow S[exp, \rho, <\mathsf{makeexclusive}>:: P],\ M,\ Q$

$S[closure(args, body, \rho), \rho', <\mathsf{makeexclusive}>:: P]_{done},\ M,\ Q\ \Longrightarrow$

$$\begin{cases} S[exclosure(args, body, \rho, q, a), \rho', P]_{done}[exclosure(args, body, \rho, q, a), \rho', <\mathsf{exgetval}, q>:: P']_{done}, \\ M \cup (a, \mathsf{free}), Q \cup (q, \mathsf{empty}) \qquad \text{if } P =<\mathsf{ApplyWith}, *>:: P' \text{ or } <\mathsf{ParaApplyWith}, *>:: P' \\ S[exclosure(args, body, \rho, q, a), \rho', P]_{done}, M \cup (a, \mathsf{free}), Q \cup (q, \mathsf{empty}) \ \text{ otherwise} \end{cases}$$

$S[primop(F), \rho, <\mathsf{makeexclusive}>:: P]_{done},\ M,\ Q\ \Longrightarrow$

$$\begin{cases} S[exprimop(F, q, a), \rho, P]_{done}[exprimop(F, q, a), \rho, <\mathsf{exgetval}, q>:: P']_{done}, \\ M \cup (a, \mathsf{free}), Q \cup (q, \mathsf{empty}) \qquad \text{if } P =<\mathsf{ApplyWith}, *>:: P' \text{ or } <\mathsf{ParaApplyWith}, *>:: P' \\ S[exprimop(F, q, a), \rho, P]_{done}, M \cup (a, \mathsf{free}), Q \cup (q, \mathsf{empty}) \qquad \text{otherwise} \end{cases}$$

4.3.3 Function Application

Function application expressions are handled by evaluating the function first, and then evaluating all argument expressions sequentially or in parallel depending on whether `pcall` is used or not.

The Sequential Case

In a sequential function application expression, arguments are evaluated left to right. When all arguments have been evaluated, the function is called if it is not exclusive. Otherwise, arguments are enqueued for exclusive calls.

$$F \notin \{\texttt{pcall}, \texttt{call/sp}, \texttt{call/mp}, \texttt{throw}, \texttt{exclusive}, \texttt{set!}\}$$
$$S[(FA_1 \cdots A_n), \rho, P]\ ,\ M\ ,\ Q \Longrightarrow S[F, \rho, <\mathsf{ApplyWith}, A_1, \cdots, A_n >:: P]\ ,\ M\ ,\ Q$$

$$S[f, \rho, <\mathsf{ApplyWith}, A_1, \cdots, A_n >:: P]_{done}\ ,\ M\ ,\ Q$$
$$\Longrightarrow S[A_1, \rho, <\mathsf{EvalArg}, 1, f, A_2, \cdots, A_n >:: P]\ ,\ M\ ,\ Q$$

$$S[v_i, \rho, <\mathsf{EvalArg}, i, f, v_1, \cdots, v_{i-1}, A_{i+1}, \cdots, A_n >:: P]_{done}\ ,\ M\ ,\ Q$$
$$\Longrightarrow S[A_{i+1}, \rho, <\mathsf{EvalArg}, i+1, f, v_1, \cdots, v_i, A_{i+2}, \cdots, A_n >:: P], M, Q, \quad 1 \le i \le n-1$$

$$S[v_n, \rho, <\mathsf{EvalArg}, n, f, v_1, \cdots, v_{n-1} >:: P]_{done}, M, Q \Longrightarrow$$
$$\begin{cases} S\ ,\ M\ ,\ enqueue(Q, q, list(v_1 \cdots v_n)) & \text{If } f = exprimop(F, a, q) \text{ or } exclosure(args, body, \rho, q) \\ S[f, \rho, <\mathsf{CallWith}, v_1, \cdots, v_n >:: P]_{done}, M, Q & \text{otherwise} \end{cases}$$

The Parallel Case

In a `pcall` expression, the function expression is evaluated first. At the same time, a queue is built as the implicit port for each argument position so that the evaluated value of can be passed to. Then, all the argument expressions are evaluated in parallel.

$$S[(\texttt{pcall}\ F\ A_1 \cdots A_n), \rho, P], M, Q \Longrightarrow S[F, \rho, < \mathsf{ParaApplyWith}, A_1, \cdots, A_n >:: P], M, Q$$

$$S[f, \rho, < \mathsf{ParaApplyWith}, A_1, \cdots, A_n >:: P]_{done}, M, Q$$
$$\Longrightarrow S[A_1, \rho, < \mathsf{enter}, q_1 >] \cdots [A_n, \rho, < \mathsf{enter}, q_n >][f, \rho, < \mathsf{getval}, q_1, \cdots, q_n >:: P]_{done},$$
$$M,\ Q \cup \{(q_i, \mathsf{empty}) | 1 \leq i \leq n\}$$

$$S[v, \rho, < \mathsf{enter}, q >]_{done}\ ,\ M\ ,\ Q \Longrightarrow S\ ,\ M\ ,\ enqueue(Q, q, v)$$

Function Call

In a function application expression, when the function and all the argument expressions have been evaluated, the function call occurs. Calling a non-exclusive function has the same semantics as in Scheme except that arguments evaluated in parallel have to synchronize and then be retrieved from the queue by a dispatching process.

$$S[primop(F), \rho, < \mathsf{CallWith}, v_1, \cdots, v_n >:: P]_{done}\ ,\ M\ ,\ Q$$
$$\Longrightarrow S[\mathcal{OP}(F, list(v_1 \cdots v_n)), \rho, P]_{done}\ ,\ M\ ,\ Q$$

[2]

[2]The formal definition of $\mathcal{OP}$ is deliberately eliminated. In general, the function $\mathcal{OP}$ applies a primitive function to a list of arguments and returns a result value.

$S[closure(list(x_1 \cdots x_n), body, \rho), \rho', <\mathsf{CallWith}, v_1, \cdots, v_n >:: P]_{done}\ ,\ M\ ,\ Q$

$\Longrightarrow S[body, \rho[x_1/a_1, \cdots, x_n/a_n], <\mathsf{getenv}, \rho' >:: P], M \cup \{(a_i, v_i) | 1 \leq i \leq n\}\ ,\ Q$

where $a_1 \cdots a_n$ are new addresses

If $v_i = hd(Qq_i) \neq \mathsf{empty}, 1 \leq i \leq n$

$S[primop(F), \rho, <\mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}\ ,\ M\ ,\ Q$

$\Longrightarrow S[primop(F), \rho, <\mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}$

$[\mathcal{OP}(F, list(v_1 \cdots v_n)), \rho, P]_{done}, M, dequeue(Q, \{q_i | 1 \leq i \leq n\})$

If $v_i = hd(Qq_i) \neq \mathsf{empty}, 1 \leq i \leq n$

$S[closure(list(x_1 \cdots x_n), body, \rho), \rho', <\mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}\ ,\ M\ ,\ Q$

$\Longrightarrow S[closure(list(x_1 \cdots x_n), body, \rho), \rho', <\mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}$

$[body, \rho[x_1/a_1, \cdots, x_n/a_n], <\mathsf{getenv}, \rho' >:: P],$

$M \cup \{(a_i, v_i) | 1 \leq i \leq n\}, dequeue(Q, \{q_i | 1 \leq i \leq n\}),$

where $a_1 \cdots a_n$ are new addresses

4.3.4 Blocking

If the lock of an exclusive function is not free, the invocation of it will be blocked until the shared lock is released. In the parallel case, when the lock is free, evaluated arguments that are dequeued and supplied to the pcalled exclusive function will be dispatched directly without going through the queue associated with the exclusive function. This is to preserve the order of values that are sent to these ports.

If $Ma = \mathsf{free}$ and $hd(Qq) = list(v_1 \cdots v_n)$

$S[exprimop(F, q, a), \rho', < \mathsf{exgetval}, q >:: P]_{done}\ ,\ M\ ,\ Q$

$\Longrightarrow S[exprimop(F, q, a), \rho', < \mathsf{exgetval}, q >:: P]_{done}$

$[\mathcal{OP}(F, list(v_1, \cdots, v_n)), \rho, < \mathsf{release}, a >:: P]_{done},$

$update(M, a, \mathsf{locked}), dequeue(Q, \{q_i | 1 \leq i \leq n\})$

If $Ma = \mathsf{free}$ and $hd(Qq) = list(v_1 \cdots v_n)$

$S[exclosure(list(x_1 \cdots x_n), body, \rho, q, a), \rho', < \mathsf{exgetval}, q >:: P]_{done}\ ,\ M\ ,\ Q$

$\Longrightarrow S[exclosure(list(x_1 \cdots x_n), body, \rho, q, a), \rho', < \mathsf{exgetval}, q >:: P]_{done}$

$[body, \rho[x_1/a_1, \cdots, x_n/a_n], < \mathsf{release}, a >::< \mathsf{getenv}, \rho' >:: P],$

$update(M, a, \mathsf{locked}) \cup \{(a_i, v_i) | 1 \leq i \leq n\}, dequeue(Q, \{q_i | 1 \leq i \leq n\})$

where $a_1 \cdots a_n$ are new addresses

$S[v, \rho, < \mathsf{release}, a >:: P]_{done}\ ,\ M\ ,\ Q \Longrightarrow S[v, \rho, P]_{done}\ ,\ update(M, a, \mathsf{free})\ ,\ Q$

If $Ma = \mathsf{free}$ and $v_i = hd(Qq_i) \neq \mathsf{empty}, 1 \leq i \leq n$

$S[exprimop(F, q, a), \rho, < \mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}, M, Q$

$\Longrightarrow S[exprimop(F, q, a), \rho, < \mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}$

$[\mathcal{OP}(F, list(v_1, \cdots, v_n)), \rho, < \mathsf{release}, a >:: P]_{done},$

$update(M, a, \mathsf{locked}), dequeue(Q, \{q_i | 1 \leq i \leq n\})$

If $Ma = \mathsf{free}$ and $v_i = hd(Qq_i), 1 \leq i \leq n$

$S[exclosure(list(x_1 \cdots x_n), body, \rho, q, a), \rho', < \mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}, M, Q$

$\Longrightarrow S[exclosure(list(x_1 \cdots x_n), body, \rho, q, a), \rho', < \mathsf{getval}, q_1, \cdots, q_n >:: P]_{done}$

$[body, \rho[x_1/a_1, \cdots, x_n/a_n], < \mathsf{release}, a >::< \mathsf{getenv}, \rho' >:: P],$

$update(M, a, \mathsf{locked}) \cup \{(a_i, v_i) | 1 \leq i \leq n\}, dequeue(Q, \{q_i | 1 \leq i \leq n\})$

where $a_1 \cdots a_n$ are new addresses

4.3.5 Environment Recovery

When a function is called, its lexical environment stored in the closure is referred for evaluation. When the call returns, the old environment is retrieved.

$S[v, \rho, < \mathsf{getenv}, \rho' >:: P]_{done}\ ,\ M\ ,\ Q \Longrightarrow S[v, \rho', P]_{done}\ ,\ M\ ,\ Q$

4.3.6 First Class Ports

Ports can be reified as language constructs. The shared information about the reified port, including the control point, current environment and its state, is stored in a shared memory location, and can be referenced by any thread. A singleport is represented by the control point, its current environment, and its active/closed state. In the case of multiports, if the current port is associated with an argument position of a pcall expression (the second rule below), or with the last argument position of an exclusive function call expression (the third and fourth rule below), the queuing information in the current port has to be stored and shared as well. Otherwise, only the control point and current environment information are stored, and the multiport degenerates to be a procedural abstraction (the fifth rule below). In general, a port provides ordering information only when ordering is important at the control point the port represents.

$S[(\texttt{call/sp}\ F), \rho, P]\ ,\ M\ ,\ Q$

$\Longrightarrow S[F, \rho, < \mathsf{CallWith}, a >:: < \mathsf{refport}, a >], M \cup \{(a, < \mathsf{singleport}, \rho >:: P)\},\ Q$

$S[(\texttt{call/mp}\ F), \rho, <\mathsf{enter}, q>]\ ,\ M\ ,\ Q$

$\Longrightarrow S[F, \rho, <\mathsf{CallWith}, a>::<\mathsf{refport}, a>], M \cup \{(a, <\mathsf{multiport}, \rho, q>)\}, Q$

$S[(\texttt{call/mp}\ F), \rho, <\mathsf{EvalArg}, n, exclosure(args, body, \rho', q, a), v_1, \cdots, v_{n-1}>:: P]\ ,\ M\ ,\ Q$

$\Longrightarrow S[F, \rho, <\mathsf{CallWith}, a>::<\mathsf{refport}, a>] M \cup \{(a, <\mathsf{multiport}, \rho, q>)\}, Q$

$S[(\texttt{call/mp}\ F), \rho, <\mathsf{EvalArg}, n, exprimop(F, q, a), v_1, \cdots, v_{n-1}>:: P]\ ,\ M\ ,\ Q$

$\Longrightarrow S[F, \rho, <\mathsf{CallWith}, a>::<\mathsf{refport}, a>] M \cup \{(a, <\mathsf{multiport}, \rho, q>)\}, Q$

If $P \neq <\mathsf{enter}, q>$, or

$<\mathsf{EvalArg}, n, exclosure(*), v_1, \cdots, v_n>:: P'$ or

$<\mathsf{EvalArg}, n, exprimop(*), v_1, \cdots, v_n>:: P'$

$S[(\texttt{call/mp}\ F), \rho, P]\ ,\ M\ ,\ Q$

$\Longrightarrow S[F, \rho, <\mathsf{CallWith}, a>:: P], M \cup \{(a, <\mathsf{getenv}, \rho>:: P)\}, Q$

4.3.7 Parallel Control

A `(die)` expression terminates the execution of the current thread without returning anything. If during the invocation of an exclusive function, a `(die)` is evaluated, the lock is released so that pending requests to call the exclusive function can be satisfied. A `(die)` releases all locks in the current thread if it happens in nested exclusive function calls.

$S[(\texttt{die}), \rho, \mathsf{RETURN}]\ ,\ M\ ,\ Q \Longrightarrow S\ ,\ M\ ,\ Q$

$$S[(\texttt{die}), \rho, < \mathsf{enter}, q >] \ , \ M \ , \ Q \Longrightarrow S \ , \ M \ , \ Q$$

$$S[(\texttt{die}), \rho, tag :: P] \ , \ M \ , \ Q$$
$$\Longrightarrow \begin{cases} S[(\texttt{die}), \rho, P] \ , \ update(M, a, \mathsf{free}) \ , \ Q & \text{if } tag =< \mathsf{release}, a, q > \\ S[(\texttt{die}), \rho, P] \ , \ M \ , \ Q & \text{otherwise} \end{cases}$$

A `throw` expression causes the left-to-right evaluation of its arguments, and passes the result of the second argument to the result of the first argument (a location storing a port). The semantic rules of `throw` guarantee that two throws by the same thread to a multiport with a queuing mechanism will cause the values to be enqueued in the throwing order.

$$S[(\texttt{throw}\ A_1\ A_2), \rho, P], M, Q \Longrightarrow S[A_1, \rho, < \mathsf{throw}, A_2 >:: P], M, Q$$

$$S[v, \rho, < \mathsf{throw}, A_2 >:: P]_{done}, M, Q \Longrightarrow S[A_2, \rho, < \mathsf{PassTo}, v >:: P], M, Q$$

$$S[v, \rho, < \mathsf{PassTo}, a >:: P]_{done}, M, Q \Longrightarrow$$
$$\begin{cases} S[v, \rho, P]_{done}[v, \rho', P']_{done}, update(M, a, \mathsf{closed}), Q & \text{if } Ma =< \mathsf{singleport}, \rho' >:: P' \\ S[v, \rho, P]_{done}, M, Q & \text{if } Ma = \mathsf{closed} \\ S[v, \rho, P]_{done}, M, enqueue(Q, q, v) & \text{if } Ma =< \mathsf{multiport}, \rho', q > \\ S[v, \rho, P]_{done}[v, \rho', P']_{done}, M, Q & \text{if } Ma =< \mathsf{getenv}, \rho' >:: P' \end{cases}$$

$$S[v, \rho, < \mathsf{singleport}, \rho' >:: P']_{done}, M, Q \Longrightarrow S[v, \rho', P'], update(M, a, \mathsf{closed}), Q$$

$$S[v, \rho, \mathsf{closed}]_{done}, M, Q \Longrightarrow S, M, Q$$

$S[v, \rho, <\mathsf{multiport}, \rho', q>]_{done}, M, Q \Longrightarrow S, M, enqueue(Q, q, v)$

$S[v, \rho, <\mathsf{refport}, a>]_{done}, M, Q \Longrightarrow S[v, \rho, Ma]_{done}, M, Q$

4.4 An Example

We illustrate in this section, with the short example `(pcall + (call/mp (lambda(p) 1)) 2)`, that our operational semantics correctly defines the meaning of a Pscheme expression. The following series of reduction shows that the expression will return 3 as its final answer. In this example, the series of rules to apply is chosen nondeterministically. However, it is not difficult to observe that the order of reduction rules chosen will not affect the terminating configuration, in which the value 3 will be returned as the final result.

[`(pcall + (call/mp (lambda(p) 1)) 2)`,{},RETURN], {}, {}

$\Longrightarrow$ [+,{},<ParaApplyWith,`(call/mp (lambda(p) 1))`,2>::RETURN], {}, {}

$\Longrightarrow$ $[primop(+), \{\}, <$ ParaApplyWith, `(call/mp (lambda(p) 1))`, $2>::$ RETURN$]_{done}$,
{}, {}

$\Longrightarrow$ $[primop(+), \{\}, <$ getval, $q_1, q_2 >::$ RETURN$]_{done}$
[`(call/mp (lambda(p) 1))`, $\{\}, <$ enter, $q_1 >$][2, $\{\}, <$ enter, $q_2 >$],
{}, $\{(q_1,$ empty$), (q_2,$ empty$)\}$

$\Longrightarrow$ $[primop(+), \{\}, <$ getval, $q_1, q_2 >::$ RETURN$]_{done}$
[`(call/mp (lambda(p) 1))`, $\{\}, <$ enter, $q_1 >$][2, $\{\}, <$ enter, $q_2 >$],

$\{\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$[\texttt{(lambda(P) 1)}, \{\}, < \mathsf{CallWith}, a > :: < \mathsf{refport}, a >][2, \{\}, < \mathsf{enter}, q_2 >],$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$[closure(list(p), 1, \{\}), \{\}, < \mathsf{CallWith}, a > :: < \mathsf{refport}, a >]_{done}[2, \{\}, < \mathsf{enter}, q_2 >],$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$[1, \{(p, a)\}, < \mathsf{refport}, a >][2, \{\}, < \mathsf{enter}, q_2 >],$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$[1, \{(p, a)\}, < \mathsf{refport}, a >]_{done}[2, \{\}, < \mathsf{enter}, q_2 >],$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$[1, \{\}, < \mathsf{multiport}, \{\}, q_1 >]_{done}[2, \{\}, < \mathsf{enter}, q_2 >],$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}[2, \{\}, < \mathsf{enter}, q_2 >],$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, 1 :: \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}[2, \{\}, < \mathsf{enter}, q_2 >]_{done},$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, 1 :: \mathsf{empty}), (q_2, \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$\{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, 1 :: \mathsf{empty}), (q_2, 2 :: \mathsf{empty})\}$

$\Longrightarrow [primop(+), \{\}, < \mathsf{getval}, q_1, q_2 > :: \mathsf{RETURN}]_{done}$

$[3, \{\}, \mathsf{RETURN}]_{done}, \{(a, < \mathsf{multiport}, \{\}, q_1 >\}, \{(q_1, \mathsf{empty}), (q_2, \mathsf{empty})\}$

(Termination condition)

Chapter 5

The Compiler

Our implementation of Pscheme consists of two parts: the compiler and the run time system. The compiler generates portable C code so that Pscheme can run on different parallel platforms. The run time system takes care of resource management, ordering, locking and process control. We describe our compilation process in this chapter and the run time system in the next.

The Pscheme compiler is based on the approach of the SML/NJ compiler [4,5]. Because our port mechanism is an extension of continuations, we only need to modify the continuation-passing-style program representation so as to be able to characterize multiple threads and ports. By continuation passing style, we mean all function calls are tail calls, and all function bodies end with function call expressions. In general, a compiled Pscheme program will have the following features.

- C code, instead of assembly, is generated to increase portability among different parallel platforms.

- All objects are heap allocated. The compiler does no escape analysis to decide whether a closure (or a continuation) should be stack-allocated or heap-allocated.

- Every thread is in the continuation passing style, and is abstracted as a single function application expression.

- Ports are parallel data structures containing parallel control information such synchronization, thread creation and thread termination.

Our operational semantics associates each subexpression in a Pscheme program with an implicit port, and interprets the program in the *port passing style*. Our compiler follows this viewpoint except that ports will only be constructed when they are made explicit (via `call/mp`, `call/sp` or `pcall`). Since the ordering is important only when there is an order among elements passed to an explicit port, the continuation passing thread representation suffices if ports can be encapsulated in continuations, and the order of elements passed to a port can be guaranteed. In other words, passing a value to a port is the same as invoking a continuation which does nothing but the *pass* operation. [1]

The Pscheme compiler is composed of a series of program transformation processes, which transforms the input program into an intermediate representation for the code generator to use. These stages are:

1. **Syntactic preprocessing**
2. **Scope analysis**
3. **Assignment conversion**
4. **CPS transform**
5. **Closure conversion**
6. **Thread abstraction**
7. **Code generation**

[1] The *pass* operation is implemented as a parallel primitive operation in the run time system.

Since the Pscheme compiler generates C code, we do not perform register allocation. Instead, we assume a sufficient number of virtual registers allocated by the run time system, and those virtual registers can be *cached* (for example, declared as register variables) in C.

5.1 Intermediate Representation

Our compiler assumes the input program to be a series of `define` expressions followed by a series of regular Pscheme expressions. The answer to the program will be the return value of the last expression under the bindings of all those `define` expressions and the side effects of previous expressions. With a legal input, these transformations generate an intermediate representation, which is a big letrec expression in CPS form. [2] A CPS expression *cexp* in our compiler is a tagged tuple defined as follows:

[2] The *port-passing style* representation used in the operational semantics is not adopted here for there is no need to create a port for each subexpression. As mentioned above, a port is only created when the ordering matters. Instead, continuation-passing style suffices as long as we are able to encapsulate relevant port information in continuations.

$$
\begin{aligned}
cexp \;=\; & < \mathsf{letrec}, binding \text{ list}, cexp > \\
\text{or} \; & < \mathsf{primop}, sexp \text{ list}, id, cexp > \\
\text{or} \; & < \mathsf{content}, id, cexp > \\
\text{or} \; & < \mathsf{assign}, id, sexp, id, cexp > \\
\text{or} \; & < \mathsf{if}, id, cexp, cexp > \\
\text{or} \; & < \mathsf{para}, \text{apply-expression list} > \\
\text{or} \; & < \mathsf{pass}, sexp, id > \\
\text{or} \; & < \mathsf{enq-port}, sexp, id > \\
\text{or} \; & < \mathsf{deq-port}, id > \\
\text{or} \; & < \mathsf{apply}, id, sexp \text{ list} > \\
\text{or} \; & < \mathsf{end}, sexp > \\
\text{or} \; & < \mathsf{die} >
\end{aligned}
$$

where

- *id* denotes a legal identifier in Pscheme,
- *sexp* denotes a "simple expression", which is either an identifier or a language level constant (which could be a number, a character, a boolean, a string or a quoted symbol), and
- a *binding* is a $(id, bexp)$ pair where *bexp* is a data structure expression defined as follows.

$$
\begin{aligned}
bexp \;=\;& < \textsf{lambda}, id \text{ list}, cexp > \\
\text{or}\;& < \textsf{singleport}, id > \\
\text{or}\;& < \textsf{multiport}, id > \\
\text{or}\;& < \textsf{exclusive}, id > \\
\text{or}\;& < \textsf{build} - \textsf{port}, id, int > \\
\text{or}\;& < \textsf{sync}, id, int, id > \\
\text{or}\;& < \textsf{bind}, id, sexp > \\
\text{or}\;& < \textsf{vector}, \text{constant list} > \\
\text{or}\;& < \textsf{cell}, id > \\
\text{or}\;& < \textsf{closure}, id \text{ list} > \\
\text{or}\;& < \textsf{select}, id, int > \\
\text{or}\;& < \textsf{offset}, id, int > \\
\text{or}\;& \text{a } cexp \text{ containing only } \textsf{primop}, \textsf{content}, \textsf{assign} \text{ or } \textsf{if} \text{ expressions}
\end{aligned}
$$

A binding expression *bexp* represents the construction of a compound data object to be bound to an identifier. It only appears in the `letrec` binding.

In the intermediate representation, the only thread-creating expression is the para expression, which specifies a list of thread abstractions to create, each of which is actually an apply expression. Passing values to ports are done by the pass expression, which can either terminate the current thread or activate a new continuation that the port object encapsulates depending on the locking information in the port object. The end expression returns the answer to the top level. The die expression terminates the current thread.

All other expressions are similar to their sequential CPS counterparts. The apply expression is a no-return function call (a tail call). The primitive function call (the

primop expression shown above) is also in CPS form except that its continuation is not explicitly constructed to take the result as the argument. Instead, we bind a new variable (the component *id*) with the result and perform the subsequent computation (the last component *cexp*) directly. The same case applies to the content and assign expressions.

The definition of the binding expression *bexp* represents objects of all data types aside from those of expressible constants (numbers, characters, booleans, strings and quoted symbols). It is worth noticing that we have distinguished *lambda objects* (or lambdas) and *closure objects* (or closures) so that lambdas contain primarily the code information of a lambda expression, and closures are records whose fields can be lambdas or free variable values. (Our transformation always builds closure records with several consecutive lambda fields followed by several consecutive free variable fields.)

Closures may possibly have common free variables. In order to let closures share fields, a closure is characterized by a pointer to a record and a non-negative *offset* that locates the closure's base address (where in the record the closure starts) by the distance to that record's first field. select selects a field of a closure. offset returns another closure sharing some fields with the original closure record but with a different base address (that is, a different *offset*). exclusive takes a lambda and returns another identical one except that calling it is exclusive.

bind binds the label with a simple constant or another identifier. A cell $< \mathsf{cell}, id >$ is a pointer object pointing to some location stored with the object that *id* denotes. Port-related expressions will be explained later in this chapter.

A binding expression can also be a CPS expression as long as function applications or parallel primitives do not occur in it. The principle is to allow only primitive operations and storage operations in a binding expression so as not to violate the CPS form.

Notice that our binding expression contains no construct for building lists or cons-cells. Cons-cells are only constructed through the primitive operator `CONS`. Even for a constant list, the compiler will create a new identifier (and allocate space) for each cons-cell needed to construct the list.

5.2 Syntactic Preprocessing, Scope analysis and Assignment Conversion

5.2.1 Syntactic Preprocessing

The first stage of the Pscheme compiler translates syntactic forms into Pscheme domain expressions. There are four kinds of syntactic forms related to conditional expressions: `case`, `cond`, `and` and `or`. They are all transformed into `if` expressions. [3]

The `let` form is translated into a lambda application expression as follows.

`(let ((x1` exp_1`)..(xn` exp_n`))` *body*`)`

$\Longrightarrow$ `((lambda (x1 .. xn)` *body*`)` exp_1 .. exp_n`)`

The `letrec` expression, according to Scheme's definition, should be translated as follows.

`(letrec ((x1` exp_1`)..(xn` exp_n`))` *body*`)`

$\Longrightarrow$ `(let ((x1 nil)..(xn nil)) (set! x1` exp_1`)..(set! xn` exp_n`)` *body*`)`

The `letrec` form is usually used to express mutually recursive functions, but Scheme's `letrec` form means more. Transforming a `letrec` expression according to the definition may cause extra inefficiency. We would like to keep a structure characterizing mutual recursion at the intermediate representation level. Such a *selective* transformation will be

[3] The transform algorithm is trivial by definition, and is thus omitted.

discussed in more details in section 5.3.3.

5.2.2 Scope Analysis

The next stage is *scope analysis*, which renames identifiers with the same names but appearing in different scopes so that every identifier name is unique. The reason to perform such transformation is that it makes it easier to restructure the program by removing nested binding blocks. (We call it *hoisting*.) This is done simply by recording the scope pointer in the entry of each identifier in the symbol table so that each occurrence of the variable to rename can be found.

5.2.3 Assignment Conversion

In Scheme (and all imperative languages), identifiers denote values sometimes and locations in other time. In most imperative languages, the compiler has to distinguish left-hand-side identifiers and right-hand-side identifiers to generate appropriate code. In an expression language such as Scheme, where every expression returns a value, this violates the orthogonality of an expression's meaning. One solution to this problem is to let the compiler explicitly distinguish locations and values in its intermediate representation. Some Scheme compilers (such as ORBIT [47]) transform all the `set!` expressions so that the meanings of identifiers are not subject to change, and the compiler can translate program expressions regardless where they appear. The Pscheme compiler also performs such a transformation. The principle is shown as follows.

Case 1 : Binding through `lambda`

```
(lambda (x) .. x ..  (set! x v) .. )
```

$\Downarrow$

```
(lambda (x)
   (let ((x' (cell x)))
      .. (content x') ..
      (assign x' v)
      ... ))
```

$\Downarrow$

```
(lambda (x)
   ((lambda (x')
       .. (content x') ..
       (assign x' v)
       ... )
    (cell x)))
```

Case 2 : Binding through `letrec`

```
(letrec (.. (x init-exp) ..)
   .. x ..
   (set! x v)
   .. )
```

$\Downarrow$

```
(letrec (.. (x init-exp) (x' (cell x)) ..)
   .. (content x') ..
   (assign x' v)
   .. )
```

That is, if an identifier `x` will be assigned a new value by `set!`, we create a new identifier `x'` denoting the location storing `x`. For all references to the *value* of `x`, we use `(content x')` in place, and for all `set!` expressions referring the location of `x` such as `(set! x exp)`, we use `(assign x' exp)` instead.

After assignment conversion, we are able to perform CPS transform, the kernel part of the compiler.

5.3 The CPS Transform Process

The CPS transform is a technique used in compilers for languages supporting first-class continuations. If the implementation is stack based, capturing a continuation may require saving the whole run time stack, and invoking the continuation requires restoring the stack. If the program is in continuation passing style, all function calls are tail calls. Invoking a continuation is the same as calling a function because neither needs to return. The run time stack degenerates to an activation record since the callee's activation can always overwrite the caller's activation.

Pscheme's compiler uses an extension of the CPS transform to characterize ports and parallelism. We will discuss the CPS transform in Pscheme's sequential part first, and then extend it to the whole parallel language.

5.3.1 Transforming Sequential Expressions

Continuation passing style is a form for program expressions in which any lambda expression and any function call satisfy the following property:

1. The last step of a lambda expression's body must be a function call expression
2. Any function call expression must appear at the tail position of a lambda expression's body

Before the CPS transform, if in an expression, a function call does not appear in a tail position, whatever follows the function call will be *wrapped* into another function, which we call the function's *continuation*. This continuation is passed as a parameter of the function call, and will be invoked later. After such a transform process, the expression is in the *continuation passing style* (CPS). In a CPS expression, each function takes one more argument, the continuation, than its original form. All continuation functions constructed during the transform are one-argument functions.

For example, the factorial function

```
(define (fac n) (if (eq? 0 n) 1 (* n (fac (- n 1)))))
```

can be transformed into continuation passing style as follows.

```
(define (fac n cont)
  (eq? 0 n (lambda (v1)
             (if v1
                 (cont 1)
                 (- n 1 (lambda (v2)
                          (fac v2 (lambda (v3)
                                    (* n v3 cont)))))))))
```

This new factorial function satisfies the two CPS properties stated above. When `fac` is called at the top level, the continuation argument `cont` is a function that just prints its argument and terminates.

The CPS transform algorithm for sequential expressions is shown in figure 5.1. The transformation function CPS takes a language domain expression (in bold fonts and Pscheme syntax) and a (meta-level) continuation function, and returns a transformed domain expression. The continuation function can be reified to become a language level

continuation (a one-argument lambda expression) during the transformation. The **begin** expression actually represents, in general, any list of expressions to be evaluated sequentially such as the body of a **lambda** or a **letrec**.

As we have explained in the intermediate representation, primitive function calls are also in CPS form, but no continuation for a primitive operation is explicitly constructed in a lambda form. Instead, we create a new variable bound to the result of the primitive operation, and perform the computation following it. Therefore, the factorial function, after the CPS transform, has the representation shown in figure 5.2.

The transform automatically builds a continuation function representing the *top level* (the `end` instruction) so that when it is invoked, the answer is printed and the computation is complete.

Transforming a `letrec` expression needs some preprocessing, and will be discussed in section 5.3.3.

5.3.2 Ports and Parallel Constructs

The CPS transform in the Pscheme compiler is extended to deal with **pcall**, **throw**, **die**, **call/sp**, **call/mp** and **exclusive**. The idea is to enscapsulate port information in continuations, and to use a uniform construct `para` to specify thread creation. (That is, parallelism is only generated via the command `para`.)

In the compiler's internal representation, a port created by `call/mp` or `call/sp` is an *asynchronous port*, which contains internally a FIFO queue for objects passed in and a pointer to a continuation function representing the rest of the computation. A port created by `pcall` is a *synchronous port*, which internally also contains a FIFO queue, a pointer to a *synchronizer*, (which is the function whose argument position is associated with the port) and an index indicating which argument position of the synchronizer the

$CPS : E \times (E \rightarrow E) \rightarrow E$

$CPS(\mathbf{e}, k) = k(\mathbf{e})$ if $\mathbf{e} \in Constants \cup Identifiers$

$CPS((\mathbf{if\ e1\ e2\ e3}), k)$
$= CPS(\mathbf{e1}, \lambda v.(\mathbf{if}\ v\ CPS(\mathbf{e2}, k)\ CPS(\mathbf{e3}, k)\ )\ )$

$CPS((\mathbf{content\ x}), k) = (\mathbf{content\ x}\ reify(k))$

$CPS((\mathbf{asign\ x\ e}), k) = CPS(\mathbf{e}, \lambda v.\ (\mathbf{assign\ x}\ v\ reify(k)\ )\ )$

$CPS((\mathbf{func\ arg1\ arg2\ ..\ argn}), k)$
$= CPS(\mathbf{func}, \lambda f.CPS(\mathbf{arg1}, \lambda v_1. \cdots CPS(\mathbf{argn}, \lambda v_n.(f\ v_1 \cdots v_n\ reify(k))) \cdots)$

$CPS((\mathbf{lambda\ args\ body}), k)$
$= k(\ (\mathbf{lambda}\ append(\mathbf{args}, list(k))\ \ CPS(\mathbf{body}, \lambda v.(\mathbf{k}\ v))\ )\ )$
where **k** is a new identifier

$CPS((\mathbf{begin\ e1\ ..\ en}), k)$
$= CPS(\mathbf{e1}, \lambda v_1.\ CPS(\cdots CPS(\mathbf{en}, \lambda v_n.(reify(k)\ v_n)) \cdots)$

$CPS((\mathbf{letrec\ ((x1\ e1)\ ..\ (xn\ en))\ body}), k)$
$= (\mathbf{letrec\ ((x1}\ CPS(\mathbf{e1},\ \lambda v.\ (\mathbf{bind\ x1}\ v)\ )\ )\ \mathbf{..\ (xn}\ CPS(\mathbf{en},\ \lambda v.\ (\mathbf{bind\ xn}\ v)\ )\ ))$
$\quad CPS(\mathbf{body}, k)\ )$

$reify(\lambda v.\mathbf{exp}) = (\mathbf{lambda(v)\ exp}[v/\mathbf{v}])$

Figure 5.1: The CPS transform algorithm for sequential constructs

```
(define (fac n) (if (zero? n) 1 (* n (fac (- n 1)))))
(fac 3)


LETREC
 FAC = LAMBDA N f1
         LETREC
           f3 = LAMBDA v2
                  APPLY f1 v2

           PRIM EQ? 0 N   v5
           IF v5
           THEN APPLY f3 1
           ELSE LET
                  f13 = LAMBDA v12
                          PRIM * N v12   v14
                        APPLY f3 v14

                  PRIM - N 1   v11
                  APPLY FAC v11 f13

 f17 = LAMBDA v17
         END    v17

 APPLY FAC 3 f17
```

Figure 5.2: The CPS transformed Factorial function

port represents.

Transforming a `pcall` expression creates a *synchronizer* object, which specifies internally the `pcall`ed function, its arity and its continuation so that when a data element is received at each of its *synchronous ports*, the `pcall`ed function can be invoked with the continuation argument specified in the synchronizer. These compiler-constructed objects are represented by the following instructions.

< singleport, *continuation* >

< multiport, *continuation* >

< buildport, *pcalled_function*, *index* >

< sync, *pcalled_function*, *arity*, *continuation* >

All of these are binding expressions that only occur in the `letrec` binding in a transformed expression. The CPS transform algorithm for parallel constructs is shown in figure 5.3, We give short examples for these instructions to show how the transformation works.

Transforming the expression `(pcall e0 e1 .. en)`

The compiler transforms `e0, e1, .. en`, and generates codes that evaluate `e0` first, and then `e1` to `en` in parallel. For example, the following `pcall` expression

```
(pcall (lambda(x y) (+ x y)) 1 2)
```

can be CPS transformed into:

```
LETREC
 f3 =  LAMBDA X Y f2
         PRIM + X Y   v4
         APPLY f2 v4
```

$CPS(($**pcall e0 e1 .. en**$), k)$
$= CPS($**e0**$, \lambda f.($**letrec ((s (make-synchronizer f** $reify(k))))$
(para
$CPS($**e1**$, \lambda v.($**pass** v **s**$))$
$\vdots$
$CPS($**en**$, \lambda v.($**pass** v **s**$)\ )))\ )$

$CPS(($**call/mp func**$), k)$
$= CPS($**func**$, \lambda f.($**letrec ((p (multiport** $reify(k)))))\ (f$ **p** $reify(k)))\ )$

$CPS(($**call/sp func**$), k)$
$= CPS($**func**$, \lambda f.($**letrec ((p (singleport** $reify(k)))))\ (f$ **p** $reify(k)))\ )$

$CPS(($**throw e1 e2**$), k)$
$= CPS($**e1**$, \lambda p.CPS($**e2**$, \lambda v.($**begin (enq-port** $v\ p$**)**
(para (deq-port p**)**
$(reify(k)\ v)))\ ))$

$CPS(($**exclusive func**$), k)$
$= CPS($**func**$, \lambda f.($**letrec ((f' (exclusive** f**))) (**$reify(k)$ **f'))**$\ )$

$CPS(($**die**$), k) =$ **(die)**

$reify(\lambda v.$**exp**$) =$ **(lambda(v) exp[**v**/v])**

Figure 5.3: The CPS transform algorithm for parallel constructs

```
f7 =  LAMBDA v5
        END    v5
s1 = SYNC f3 2 f7
p1 = BUILD-PORT s1 1
p2 = BUILD-PORT s1 2

PARA
  PASS 1 p1
  PASS 2 p2
```

Since a two-argument function is called, two synchronous ports and a synchronizer are built. Our idea is to keep a parallel expression in CPS form, while dealing with side effects concerning parallelism and synchronization through shared data structures.

Transforming `throw`, `call/sp` and call/mp expressions

The `throw` expression causes left-to-right evaluation of the arguments to get a port p and a value v. It passes v to p, and then spawns a new thread which dequeues the port p and performs the computation *encapsulated in* the port. [4]

In the next example, we introduce our three port operations: $<$ `pass`$, v, p >$, $<$ `enq-port`$, v, p >$, and $<$ `deq-port`$, p >$. $<$ `enq-port`$, v, p >$ puts an element v into the queue for port p. $<$ `deq-port`$, p >$ dequeues the queue for port p to get a value v, and starts the computation that follows if no blocking occurs. In some cases, such as barrier synchronization or exclusive access, where the computation *following* the port cannot

[4] A port contains internally a pointer to a continuation function, which will be invoked whenever an element, if any, is dequeued from its internal queue. Invoking this continuation performs the computation that is called *encapsulated in* (or *following*) the port.

proceed, the `deq-port` operation will not dequeue elements from the queue. Instead, the elements will stay in the queue until blocking is released. This preserves the order in which elements are sent to a port, and is important in stream based programming.

The `pass` operation generally combines `enq-port` and `deq-port`. The reason to decompose it into two operations is for the implementation of `throw`. As shown in figure 5.3, to throw an element into a port, we have to enqueue it before spawning a new dequeuing thread. This guarantees an observationally atomic throw, that is, the values thrown by the same thread to the same port will arrive in the order in which they are thrown.

The `call/mp` and `call/sp` expressions cause the creation of *asynchronous* ports explicitly. Here is an example combining **call/sp** and **throw**.

The expression `(call/sp (lambda(p) (throw p 1)))` can be transformed to:

```
LETREC
  f1 = LAMBDA v1
         END    v1
  p1 = SINGLEPORT f1
  f2 = LAMBDA v2
         PASS v2 p1
  f4 = LAMBDA P f3
         LETREC
           f5 = LAMBDA v4
                  APPLY f3 v4

           ENQ-PORT 1 P
           PARA
             APPLY f5 1
```

```
            DEQ-PORT P

  APPLY f4 p1 f2
```

Transforming the `exclusive` expressions

The `exclusive` construct is not implemented at the compiler level. The locking and ordering mechanism depends on the facilities provided by the underlying platform, so it is implemented in the run time system. The compiler simply binds a new identifier for every occurrence of the `exclusive` construct.

For example, the `((exclusive (lambda(x) x)) 1)` will be transformed into:

```
LETREC
  f3 = LAMBDA X f2
         APPLY f2 X
  f1 = EXCLUSIVE f3
  f5 = LAMBDA v3
         END    v3

  APPLY f1 1 f5
```

5.3.3 Mutual Recursion and `letrec`

In our CPS transform, we keep the `letrec` expression as our basic binding construct. (The `let` forms have been translated as lambda applications.) `letrec` is used to represent mutually recursive functions primarily. Unfortunately, unlike other languages such as ML, Scheme's `letrec` also introduces circular defined objects because the bindings are not restricted to lambdas. Therefore, there could be expressions like:

`(letrec ((x x)) x)` or `(letrec ((x1 x2) (x2 x1)) x1)`

They are well defined in the syntactic `letrec` definition, but will return unexpected objects depending on implementation.

Our Pscheme compiler, however, rejects such circular definitions among objects other than lambda expressions. That is, mutual recursion applies only to lambda definitions. For non-lambda objects defined in the same `letrec`, there can only be non-circular dependencies among them. As a result, we can reorder all non-lambda bindings so that there is no forward reference among them.

Recall that aside from data structure building operations, a binding expression *bexp* can also be a CPS expression containing none of these control constructs: `apply, para, pass, end` and `die`. If no control construct appears in a `letrec` binding part, the CPS transform is simply applied to each of the binding expressions. If, however, the ith non-lambda binding contains a control instruction, we perform the following transform before the CPS transform.

`(letrec ((`f_1 lbd_1`)` .. `(`f_m lbd_m`)` `(`x_1 e_1`)` .. `(`x_i e_i`)` .. `(`x_n e_n`))`
`body)`

$\Downarrow$

`(letrec ((`f_1 lbd_1`)` .. `(`f_m lbd_m`)`
`(`x_1 e_1`)` .. `(`x_{i-1} e_{i-1}`)` `(`x_i `nil)` .. `(`x_n `nil))`
`(set!` x_i e_i`)`
$\vdots$
`(set!` x_n e_n`)`
`body)`

The assignment conversion is needed again for such `letrec` expressions that generate

new assignments during the transform. At last, after the CPS transform, the binding of the `letrec` expression will contain only those *bexp* instructions defined in section 5.1.

Our treatment of the `letrec` form saves unnecessary indirection caused by mutual recursion. Keeping the constructs for defining mutually recursive functions allows as many functions as possible to share the same closure record, as long as we define them in the same `letrec`. This is what the next compilation stage, closure conversion, is based on.

5.4 Free Variable Set Computing and Hoisting

As we have mentioned above, a compiled Pscheme program is a big `letrec` construct with many mutually recursive functions defined at the tope level. This requires a structural rearrangement of the whole `letrec` expression. The principle is to move as many functions as possible into the same `letrec`. This is an auxiliary stage called *hoisting* in our compiler.

Hoisting removes unnecessary nested bindings, and is performed before the *closure conversion* stage, and also before the *thread abstraction* stage. In general, transforming a hoisted expression produces less redundancy than transforming an unhoisted expression. On the other hand, since the transformation is a recursive traversal of the expression tree, it usually generates some redundant nestings. This auxiliary stage helps remove them. Remember that after the scope analysis stage, every identifier has a unique name, so generally we need not worry about the scope of any identifier when rearranging an expression.

The two basic cases in which nesting can be removed are:

`(letrec` *def-list1* `(letrec` *def-list2* `body))`

→ `(letrec` *append(def-list1,def-list2)* `body)`

```
(letrec ((x1 e1)..(xi-1 ei-1)
         (xi (letrec ((y1 e'1)..(ym e'm)) bodyi))
         (xi+1 ei+1)..(xn en))
        body)

→ (letrec ((x1 e1)..(xi-1 ei-1)
           (y1 e'1)..(ym e'm)
           (xi bodyi)
           (xi+1 ei+1)..(xn en))
          body)
```

There are cases, however, in which we have to take care of scopes. That is when new bindings are introduced by lambda expressions. Consider the expression:

`(letrec` *def-list1* `(lambda(x)` `(letrec` *def-list2 body*`)))`

We can transform it to be

`(letrec` *append(def-list1,def-list2)* `(lambda(x)` *body*`))`

if `x` does not occur free in *def-list2*. The principle applies to the second basic case, too. In general, if the binding part of a `letrec` expression should be lifted out of a non-letrec binding such as a `lambda, primop, content`, or `assign` expression, a scope check is necessary to see if the new bound variable occurs free in the lifted part.

For example, the expression

```
(letrec ((f1 (lambda(x1) (letrec ((y1 x1)) y1)))
         (f2 (lambda(x2) (letrec ((y2 1)) 2))))
```

```
PPS form :

LET
 F1 = LAMBDA X1 f1
        LET
          Y1 = BIND  Y1 X1
          APPLY f1 Y1
 F2 = LAMBDA X2 f3
        LET
          Y2 = BIND  Y2 1
          APPLY f3 2
 PRIM + 1 2   v8
 LET
  Z = BIND  Z 3
  END   F2


Hosted form:

LET
 F1 = LAMBDA X1 f1
        LET
          Y1 = BIND  Y1 X1
          APPLY f1 Y1
 Y2 = BIND  Y2 1
 F2 = LAMBDA X2 f3
        APPLY f3 2
 Z = BIND  Z 3
 PRIM + 1 2   v8
 END  F2
```

Figure 5.4: An example of hoisting

```
(+ 1 2)
(letrec ((z 3)) f2))
```

can be transformed as shown in figure 5.4. The `letrec` in `f2`'s binding can be lifted but the one in `f1`'s can not due to the free variable `x1`.

In order to perform hoisting, we have to compute the free variable set for each expression. The free variable set computation is also a recursive traversal through the expression tree. The more tedious part is to maintain the free variable set during hoisting so that an expression node will correctly have its free variable information although it is rearranged.

The free variable set is also essential for the the next transform stage – closure conversion, a transformation that makes functions have no free variables.

5.5 Closure Conversion

A CPS expression is used in a compiler's intermediate representation because it is very close to the instructions of a Von Neumann machine. Function calls are just control transfers. However, the notion of functions is less primitive than on the Von Neumann machine because a CPS function has free variables. Therefore, a function representation must have free variable information aside from code information. Such a representation is more complicated than the instructions on the Von Neumann machine. Looking up free variable information also makes the function application more expensive.

One technique to transform functions into forms in which no function has free variables is the *closure conversion.* [5] Since a closure contains information for the code as well as the free variables, if it can be passed as a function argument, all the information that a function needs can be found in it, thus the function needs no reference to those original free variables. In other words, closure conversion makes the closure representation explicit

[5] There is another technique called *lambda lifting* proposed by Thomas Johnson [41].

at the language level. Closure construction and free variable lookups become language instructions, too. The explicit representation of closure operations also makes the internal representation more machine independent.

Specifically, the idea is to let a function with free variables take one more argument, its own closure, so that any free variable can be referenced through the closure argument. Therefore, the function is *closed*. The caller of this function, on the other hand, has to pass an extra closure argument. The closure is constructed in the same binding block where the function is defined so as to be visible to all call sites.

Since the closure is distinguished from the function, whenever there is a function object passed, stored or returned, we pass, store or return the *closure* object. Whenever a function is called, we invoke the *lambda* object.

In our implementation, closures are records with several fields of code pointers followed by several fields of free variable values. Functions are all represented uniformly in a *closure passing style*. However, explicitly constructing a closure for each function is too expensive. Our hoisting stage has put as many function definitions as possible in the same `letrec` binding block. These functions defined in the same binding block have the same lexical environment, so it is reasonable to let them share a closure record. The free variables in the shared closure record are the union of all sharing functions' free variables, and the code pointer fields are naturally pointing to these sharing functions' codes. Since free variable fields follow code pointer fields, the closure for each function can be expressed by the base address of the shared record and an offset.

Figure 5.5 shows a simple example of how the closure conversion transforms the input expression `(letrec ((f (lambda(x) x)) (f 0)))`. In the transformed expression, `F` and `f4` are two closures sharing the same record, and the code fields point to lambda `F'` and `f4'` respectively. When `F'` (that is `F'''`) is called, its self-closure `F` is passed as the

```
Input expression:

(letrec ((f (lambda(x) x))) (f 0))


Continuation passing style:  (hoisted)

LETREC
 F = LAMBDA X f1
       APPLY f1 X
 f4 = LAMBDA v4
        END    v4
 APPLY F 0 f4


Closure passing style:       (hoisted)

LETREC
 F' = LAMBDA F'' X f1
        LETREC
         f1' = SELECT f1 0
         APPLY f1' f1 X
 f4' = LAMBDA f4'' v4
         END    v4
 F = CLOSURE F' f4'
 f4 = OFFSET F 1
 F''' = SELECT F 0
 APPLY F''' F 0 f4
```

Figure 5.5: An example of closure conversion

first argument of `F'` (although there is no free variable in the original `F`). Again, hoisting is performed after closure conversion for subsequent processing.

Closure sharing is sometimes restricted. Functions defined in the same `letrec` binding cannot shared a closure record if it results in circularity. For example, the input expression in figure 5.6 is legal because there is no circularity among non-lambda objects. But, closure conversion with sharing creates circularity among closure `F` and variable `X`. A correct transformation needs a dependency check to partition bindings into groups for shared closure records. In the second transformation, `F` and `G` do not share a closure, and no circularity is created.

5.6 Thread Abstraction

Our execution model is to express a sequential thread with a series of CPS function applications, and to describe parallelism through data structures (ports) and the thread spawning construct `para`, which takes a list of CPS expressions. The Pscheme compiler abstracts these CPS expressions into the form of function applications. That is, using one function application to represent a series of function applications.

Thread abstraction is the last stage of transformation. It transforms argument expressions of every `para` expression into a function application form so that it is easy to store it into data structures in the run time system.

A thread can be abstracted by computing its free variables, and creating a new function taking those free variables as arguments. For example, the expression

```
PARA
  PASS 1 p1
  PASS 2 p2
```

can be abstracted to be:

```
Input:

(letrec ((f (lambda() 1))
         (x f)
         (g (lambda() x)))
  0)

Transform 1: closure sharing causes circular definition

LETREC
 F' = LAMBDA F'' f1
        LETREC
         f1' = SELECT f1 0
         APPLY f1' f1 1
 G' = LAMBDA G'' f3
        LETREC
         X' = SELECT G'' 1
         f3' = SELECT f3 0
         APPLY f3' f3 X'
 F = CLOSURE F' G' X
 G = OFFSET F 1
 X = BIND  X F
 END  0

Transform 2: closure conversion without sharing

LETREC
 F' = LAMBDA F'' f1
        LETREC
         f1' = SELECT f1 0
         APPLY f1' f1 1
 F = CLOSURE F'
 X = BIND  X F
 G' = LAMBDA G'' f3
        LETREC
         X' = SELECT G'' 1
         f3' = SELECT f3 0
         APPLY f3' f3 X'
 G = CLOSURE G' X
 END  0
```

Figure 5.6: Closure conversion with restricted sharing

```
LETREC
  f1 = LAMBDA v1
         PASS 1 v1
  f2 = LAMBDA v2
         PASS 2 v2
  PARA
    APPLY f1 p1
    APPLY f2 p2
```

Functions representing thread abstractions are not in CPS form. Since these are all *known* functions (the compiler knows all the call sites) , we need not worry about how to call them (in CPS or not). Again, hoisting is applied after thread abstraction to remove nested bindings.

At last, we have our intermediate representation, which is close to the instruction level of a Von Neumann machine. All function definitions are lifted to the top level because none of them has free variables. In fact, a function body can have free reference to other function (lambda) names, but those are considered label constants, not free variables. The code generator knows all these definitions, too, and will have no problem generating code for such free references.

We show the whole transform process through the parallel sum example in appendix A.

5.7 Heap-based Code Generation

To generate C code that are as portable as possible, we leave the implementation of ports and all the parallel control primitives to the run time system.

The execution model that Pscheme compiler assumes is queue-based multiprocessing

with a shared heap. That is, any thread newly created is put in a global shared task queue. A number of processors keep taking thread abstractions from the task queue to execute. Coordination among threads are through ports. Any thread returning a value to the top level signals the completion of the whole computation.

Due to the global naming space, we use a shared heap to store all objects. Since C is not a tail recursive language, our CPS representation is translated into C differently. Because a sequential thread is composed of a series of tail calls, that is, a series of control transfers, we can use a dispatching function to transfer control. Calling a function can be translated as the caller returns the callee's code address to the dispatching function, and let the dispatching function call it. Thus, we have the same effect as the tail recursive language.

For each thread, we only need to allocate one activation frame because the caller's frame can always be overwritten by the callee's. All pointers (for instance, in a cons-cell, port, or closure) go to the shared heap, so overwriting the caller's frame will not cause the loss of information about any object.

The heap is an array in C. The frame for each thread works as a set of virtual registers on a processor. The size need not be large for every CPS function is light weight and contains only a few temporary variables. The frame's actual layout may depend on the platform. Currently, we use register variables in C, and do not perform real register allocation.

The dispatching function interacts with the compiled functions through a dispatching loop so as to achieve the same effect as Pscheme function's tail calls. A simplified version of the dispatching loop may look as follows:

```
while (!Done)
  start = (function_ptr) (*start)();
```

Given a thread abstraction, we can initialize `start` to be our starting function to apply.

The intermediate representation of a transformed Pscheme program is a big `letrec` form with all the lambdas defined at its top level (which is good for C does not support nested defined functions). Before code-generation, the compiler computes the frame size (that is, number of arguments and temporary variables) needed for each lambda. This is the necessary information for the garbage collector to traverse activation frames.

Generating code for binding a variable to a lambda f causes the definition of a C function f, and the construction of a lambda object containing information for the code pointer to f, its frame size, and locking information (explained in the next chapter). Generating code for binding a variable to a non-lambda object simply causes the construction of the object. The main body of the top level `letrec` is abstracted into a function with zero arguments. An application of this function is put in the global task queue initially so that the run time system can start the whole computation with this initialized task queue.

All parallel primitives are translated into function calls to the run time system. These operations, as well as Scheme's primitive operations are the only function call instructions in the code generated by the compiler.

Chapter 6

The Run Time System

6.1 The Execution Model and The Current Platform

Our run time system assumes a shared memory machine. As mentioned in last chapter, the execution model is characterized as follows, and is shown in figure 6.1.

- queue-based multi-processing: A fixed number of virtual processors keep executing threads in a global shared task queue. [21]
- A thread represents a series of tail calls to compiled Pscheme functions.
- A tail call transfers control back to the dispatching function, and the callee is then dispatched.
- Thread coordination is achieved through shared ports and the shared task queue.
- All the shared objects are in the global shared heap, and the heap can be garbage collected.

The run time system contains

- the implementation of all parallel primitives,

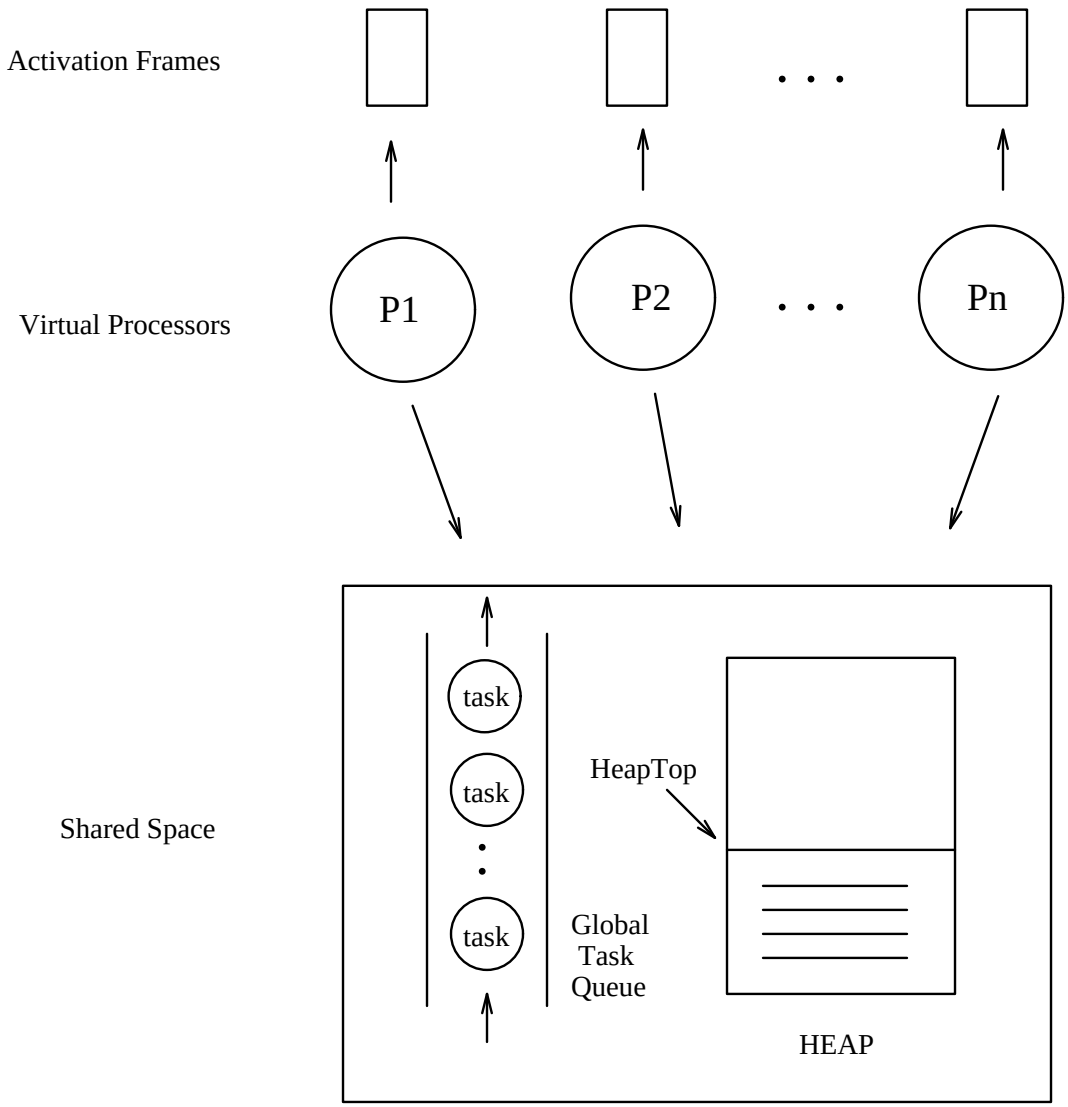


Figure 6.1: Pscheme's execution model

- an ordering mechanism,
- a garbage collector for the shared heap,
- a function dispatcher for sequential threads,
- a resource manager providing semaphores, queues, locks etc., and
- the implementation of all Scheme's primitive functions.

It is currently running on NYU Ultra II computer, which is a shared memory 8-processor machine [26,57]. Processors are 8M Hz MC68010 chips. The machine has a 32KB cache and 4MB user memory. When the 16-processor 64 MB Ultra III is available, we will be able to port our system onto it.

The key element of the Ultra computer is the hardware primitive *fetch-and-add* operation, which atomically adds an integer v to an integer variable n in the memory, and returns the old value of n. Since fetch-and-add is implemented with non-blocking parallel access to a shared word, it is naturally used to implement locks, semaphores, synchronizing barriers or parallel FIFO queues. [27]

Provided with these mechanisms, it is easy for our run time system to provide a shared heap with parallel access because the heap top pointer can be fetch-and-add'ed in parallel. Ports also support parallel access due to parallel FIFO queues, and together with locks and semaphores, they help implement an ordering mechanism, which is essential for exclusive functions and stream based programming. Also, of course, the global task queue is accessed in parallel. Synchronizing barriers are used in our garbage collection protocol to stop virtual processors and to reactivate them.

6.2 Implementation of The Run Time System

Before starting to execute the compiled program, the run time system will spawn a fixed number of virtual processors (no more than the number of real processors − 8), which are unix processes executing the same code. All of them are concurrently reading the global task queue and trying to take thread abstractions off the queue to execute. The global task queue has only one thread abstraction in the beginning. When a thread abstraction is taken by a virtual processor, it is handed to a dispatching function, and a series of function calls follows as it enters the dispatching loop. Those virtual processors having no threads to execute will keep reading the global task queue until it gets one to execute.

Whenever a virtual processor executes a `para` command, one or more new thread abstractions will be put into the task queue. Whenever one executes a `die` command, or any parallel primitive causing it to idle itself, it performs a *context switch*, meaning it accesses back to the task queue to take the next thread available to execute.

During the execution of a sequential thread, there can be primitive function calls. Recall that in the intermediate representation, the primitive function call is in a *semi-CPS* form. That is, no continuation argument is explicitly passed as its argument. Instead, the result is bound to a new variable, and the computation proceeds with this new binding. Therefore, all primitive function calls are directly implemented in terms of C function calls without going through the dispatcher.

Dynamic type checking is performed at each entrance to primitive functions. Symbolic primitive operations are mostly heap based pointer manipulation. Numerical primitive operations are not implemented according to Scheme's number system. Instead, we use C's number system.

Every object has a 4-bit type tag. The types are `CHAR`, `BOOLEAN`, `QUOTED_SBL`, `INTEGER`, `REAL`, `CELL`, `CONS_CELL`, `LAMBDA`, `CLOSURE`, `STRING`, `VECTOR`, `SYNC_PORT`, `ASYNC_PORT`, and

SYNCHRONIZER. The constant nil is of type BOOLEAN, and has the value FALSE. Ports captured at the language level are of type ASYNC_PORT, and ports constructed implicitly by pcall are of type SYNC_PORT. String constants are not stored in the heap. Instead, they are in an auxiliary shared memory which will be described in the next section. Detailed internal representation of objects can be found in Appendix B.

6.3 Storage Management

In our run time system, each virtual processor executes a sequential thread that is composed of a series of tail calls. The local space it needs is the current activation frame, which is implemented as a local object array. Calling a function causes the loading of arguments onto the activation frame. Temporary variables in a function are also allocated in the frame. Heap allocation is performed whenever a pointer is needed to construct an object. All pointers point to the heap so that no object is lost when a new frame overwrites the old one, and every pointer can be shared.

For example, constructing a closure allocates heap space for the shared record while the closure object containing the record pointer may be stored in current activation frame. Similarly, a cons-cell can be stored in the frame with its car and cdr fields pointing to the heap.

The heap stores only shared language-level objects. There is other shared space available for data elements invisible at the intermediate representation level such as shared queues, shared queue elements, locks and semaphores etc. We use another piece of preallocated shared space for them, and there also can be pointers to them from the frame or the heap. However, this piece of shared space cannot be garbage collected because internal pointers of library-provided constructs (queues, locks and so on) are not accessible. As mentioned in last section, string constants are also stored in this shared space.

6.4 Garbage Collection of The Shared Heap

The global shared heap is composed of two halves. One is in use (called the *from-space*). When it is exhausted, garbage collection is initiated, and live objects are to be copied to the other half (called the *to-space*). Cheney's copying collection algorithm [11] is among the simplest to implement. It is essentially a breadth-first traversal that copies objects from the *from-space* to the *to-space*. An object that has been visited contains a forwarding pointer that redirects any re-visits. The traversal starts from the activation frames of all threads and all thread abstractions in the global task queue.

Since there are more than one virtual processors allocating space from the shared heap, we need a protocol to stop all virtual processors and let one of them perform garbage collection when the heap is exhausted.

GC polling protocol:
A virtual processor stops for GC only when it fails to allocate enough space from the heap or it sees the GC request from another when context switching.

The GC polling happens only when a virtual processor performs context switching, that is, it has completed (or cannot go on further) the current thread and ready to take another from the task queue. When a virtual processor makes a GC request, it checks if it is the first one requesting GC. If so, it waits for every one to stop through a barrier. If not, it simply stops at the barrier. If a virtual processor is not polled or needs no heap allocation, it simply keeps running.

6.5 The Ordering Mechanism

Order preservation is an essential issue in Pscheme because ports are viewed as FIFO constructs, and different threads executing the same code need to be ordered through the use of ports and exclusive functions. This is often seen in stream-based programming examples. A stream of data elements may go through several processing stages. The output stream has to preserve the same order as the input stream.

Ordering is guaranteed in the run time system by locking and the use of ports. Ports are implemented with FIFO queues. An element can be dequeued from the port (synchronous or asynchronous) only when it will not out race previous dequeued elements. For example, if an asynchronous port p is *connected* [1] to a synchronous port q, in order to dequeue p and then enqueue q, we have to block subsequent dequeues until the enqueuing of q is done. This happens in the case of a `pcall` expression when the arguments' current ports are captured. For example, in the expression

```
(pcall f
       (call/mp (lambda (p1) ...))
       (call/mp (lambda (p2) ...)))
```

`p1` is a captured asynchronous port, which may keep receiving values, but in the implementation, a synchronous port `p1'` is also built because of `pcall`. A value v dequeued from `p1` is passed to a continuation k, which enqueues v into `p1'`. If we make the invocation of k exclusive [2], we are guaranteed that values thrown to `p1` repetitively will be enqueued to `p1'` in the same order. In this case, we say that `p1` is *connected* to `p1'` through the continuation k.

[1] to be explained later

[2] Actually, k's access lock is examined before v is dequeued from `p1`)

At the language level, programmers usually enforce the order through exclusive functions. For example, a filtering function for a data stream must be exclusive so that data elements can be explicitly thrown to the port leading to that function. When we are to dequeue the port and then call the exclusive function, we have to get access to the exclusive function (and thus lock it) first before any element is taken off the queue. This is because once data elements are dequeued, their order is lost. Both asynchronous ports and synchronous ports can lead to exclusive functions. This principle is obeyed in any case in our implementation of port related primitives such as `dequeue` and `pass`.

6.5.1 Implementation of Exclusive Functions

Exclusive functions are not implemented in the compiler. In the intermediate representation, the instruction <`exclusive`, f > constructs a lambda object f', which is the same as the lambda object f except that its access is exclusive. An exclusive lambda can appear in the field of any closure record. An *exclusive closure* is just a closure with an exclusive lambda as its field 0.

Our internal representation of a lambda has a field containing a pointer to a semaphore. If this pointer is not `null`, the lambda is exclusive, and calling the lambda requires the P operation on the semaphore first. If this pointer is `null`, the lambda is non-exclusive.

Constructing an exclusive lambda does not build a queue to store calling requests. Threads contending for a call to an exclusive lambda will be busy-waiting at the semaphore. For a port p leading to an exclusive function f, repetitively throwing values to p has the same effect as storing ordered calling requests to f. Therefore, we do not need another queue for f to store calling requests.

Releasing locks on exclusive lambdas is a little more complicated because our program is in CPS form, and there is no *return* operation in CPS form. But, since every function

except continuations takes a continuation argument, we know that when that continuation is invoked, the original function returns. Therefore, whenever an exclusive lambda is called, we store its semaphore in its continuation argument. But the continuation argument is a closure, so we actually store the semaphore in the closure's field 0, the real continuation function (a lambda object). Later on, when that continuation is invoked, we perform the V operation on that semaphore so that subsequent requests to call the exclusive lambda can be satisfied.

This approach works under the assumption that no continuation is exclusive so that the semaphore field of a lambda is not overloaded. Since continuations are all constructed by the compiler, they will not be exclusive. [3]

In addition to continuations, semaphores for exclusive lambdas have to be stored in each virtual processor as well because a `die` in the body of an exclusive lambda also releases the lock. As seen in our operational semantics, a `die` causes a trace all the way to the top level to release the locks for all nested exclusive function calls in the current thread. In the run time system, each virtual processor control block contains a stack of semaphores. Calling an exclusive lambda pushes its semaphore onto the stack., and releasing the lock for an exclusive lambda causes a pop on that stack. Clearly, a `die` empties the stack.

[3] except when a continuation is used to *connect* two ports, it has to be exclusive to guarantee elements enqueued to the second port have the same order as they are dequeued from the first port. (Please refer to last section for more explanation.) But, such a continuation function cannot possibly be the continuation argument of any user-defined exclusive lambda, so it does not violate our assumption.

Chapter 7

Performance

In this chapter, we show the performances of several sample programs. We will see how parallel Pscheme programs scale up in our system. We will also compare the performance of sequential Pscheme programs on Sun work stations with their performances on our current run time system.

7.1 Sample Parallel Programs

To observe the speedup of parallel programs in our implementation, we use the four samples:

- Parallel summation of integers from 1 to n.
- Quicksort a list of numbers
- The N-queens problem
- Computing Fibonacci numbers

The source codes of these four samples are provided in figure 7.1 and figure 7.1. The N-queens problem is to calculate the number of solutions for putting n queens on a n by

```
(define (sum m n)                          ; parallel sum
 (cond ((eq? m n) m)
       ((eq? n (+ 1 m)) (+ m n))
       (else (let ((x (/ (+ m n) 2)))
               (pcall + (sum m x) (sum (+ 1 x) n))))))



(define (fib n)                            ; Fibonacci number
  (if (<= n 3)
      n
      (pcall + (fib (- n 2)) (fib (- n 1)))))



(define (qs num-list)                      ; quicksort
 (cond ((null? num-list) nil)
       ((null? (cdr num-list)) num-list)
       (else
        (let ((pivot (car num-list)))
         (let ((l-r (partition pivot (cdr num-list) nil nil)))
           (pcall append
                  (qs (car l-r))
                  (cons pivot (qs (cdr l-r)))))))))

(define (partition p l l-buf r-buf)
  (if (null? l)
      (cons l-buf r-buf)
      (let ((x (car l)))
        (if (< x p)
            (partition p (cdr l) (cons x l-buf) r-buf)
            (partition p (cdr l) l-buf (cons x r-buf))))))

(qs (create-list 200))
```

Figure 7.1: Three sample programs for parallel performance

```
(define (queen n) (find-all n nil 0))

(define (find-all n board len)
  (if (eq? n len)
      1
      (letrec
        ((loop (lambda(v)
             (if (safe board v 1)
                     (pcall +
                            (find-all n (cons v board) (+ 1 len))
                            (if (< v n) (loop (+ v 1)) 0))
                     (if (< v n) (loop (+ v 1)) 0)))))
        (loop 1))))

(define (safe board elt distance)
  (if (null? board)
      #t
      (let ((last (car board)))
        (if (or (eq? elt last)
                (eq? elt (+ last distance))
                (eq? elt (- last distance)))
            #f
            (safe (cdr board) elt (+ distance 1))))))

(queen 6)
```

Figure 7.2: Sample program for the N-queens problem

n chess board so that they cannot attack one another. Computing Fibonacci numbers is via the parallel version of the recursive program with exponential complexity.

We vary the number of pre-spawned virtual processors to see how these programs speed up. Due to the memory limitation (4MB user space), we are unable to perform experiments with large amount of parallelism (that is, large input) . Here we provide the results of running these four programs with small parameters in figure 7.3.

Each of these programs has a very fine granularity. This affects the absolute performance because we only have a fixed number of virtual processors to execute a large number of threads in the global task queue, and no dynamic control of parallelism is provided. On the other hand, the speedup is good for programs with enough parallelism. The parallel summation program and the computation of Fibonacci numbers scale up well because of large and stable parallelism.

The quicksort program scales up the worst because there might be very uneven partitioning of a random list, and not enough parallelism may be generated. As the number of virtual processors increases, memory and bus contention increases. If there is not enough parallelism, the performance may scale poorly. The 6-queen problem has a better speedup than quick sort, but worse than parallel sum and Fibonacci. The speedup curve for these four samples is shown in figure 7.4. We expect better scalability for the quick sort and N-queens problem when the input parameter is larger.

Two important factors of parallel programs' performance are the degree of parallelism and locality. Pscheme is a shared memory language (one requiring no explicit copying of data in order to make it accessible to some part of the program), and it provides no mechanism to control excessive parallelism dynamically. Therefore, the implementation had better be able to take care of such optimizations.

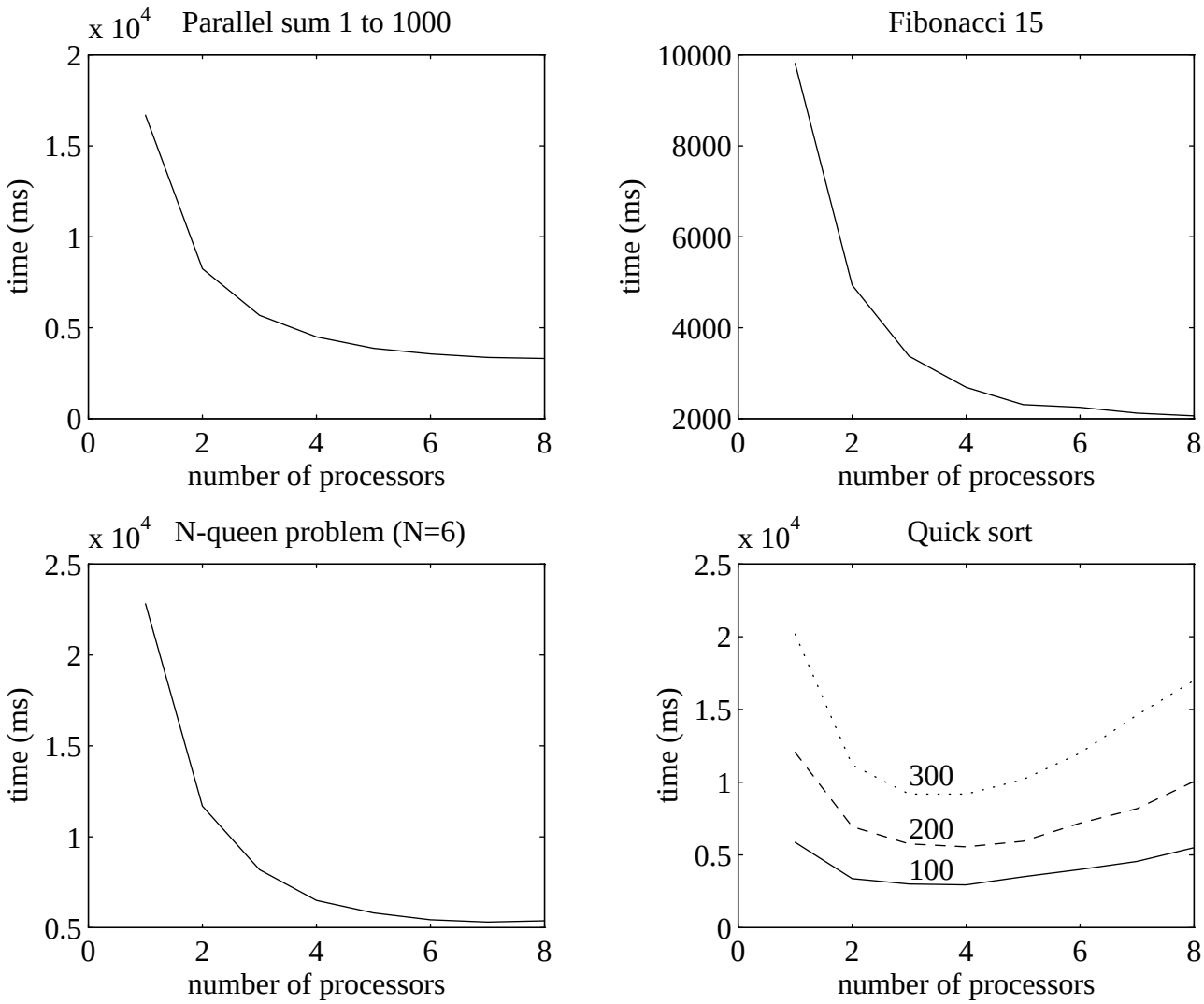


Figure 7.3: Performance of the four parallel sample programs

Our current compiler generates C code. All virtual registers and activation frames have the same access cost as the shared global heap unless the C compiler can successfully cache these hot spots. If the compiler generates assembly code and allocates machine registers directly, the locality can be improved.

Since programs in applicative languages are highly recursive, it is easy to create a lot of parallel tasks which consume resources quickly. To control excessive parallelism, some languages allow users to query the system load to decide whether to generate more tasks. For example, Qlisp [25] users can check if the global task queue is empty. Pscheme does not provide any such meta-level information at the language level. One way to solve this problem in the run time system is to apply an *unfair* scheduling policy as in Multilisp implementation [29,43,44]. That is, a newly created task is put in a local LIFO task queue for the creating processor. A processor with an empty task queue can steal tasks from other processor's task queue. The *unfair* scheduler simulates the sequential LIFO behavior to prevent combinatorial explosion of parallelism in a recursive program. At the same time, better locality can be achieved. Our run time system has not suffered from the problem of excessive parallelism when input is not too large in the experiments. When the memory limitation is relieved, we may modify the global task queue model to apply the *unfair* scheduling policy in order to experiment with large inputs.

7.2 Stream Based Programs

In chapter 3, we demonstrated two examples of stream based programming – quicksort and merge sort. The source code can be found in figure 3.13 and figure 3.14. These two examples show another dimension (vertical) of parallelism, and are expected to have a better performance speedup than programs exploiting only horizontal parallelism. Figure 7.5 shows the results of running these two programs with small inputs.

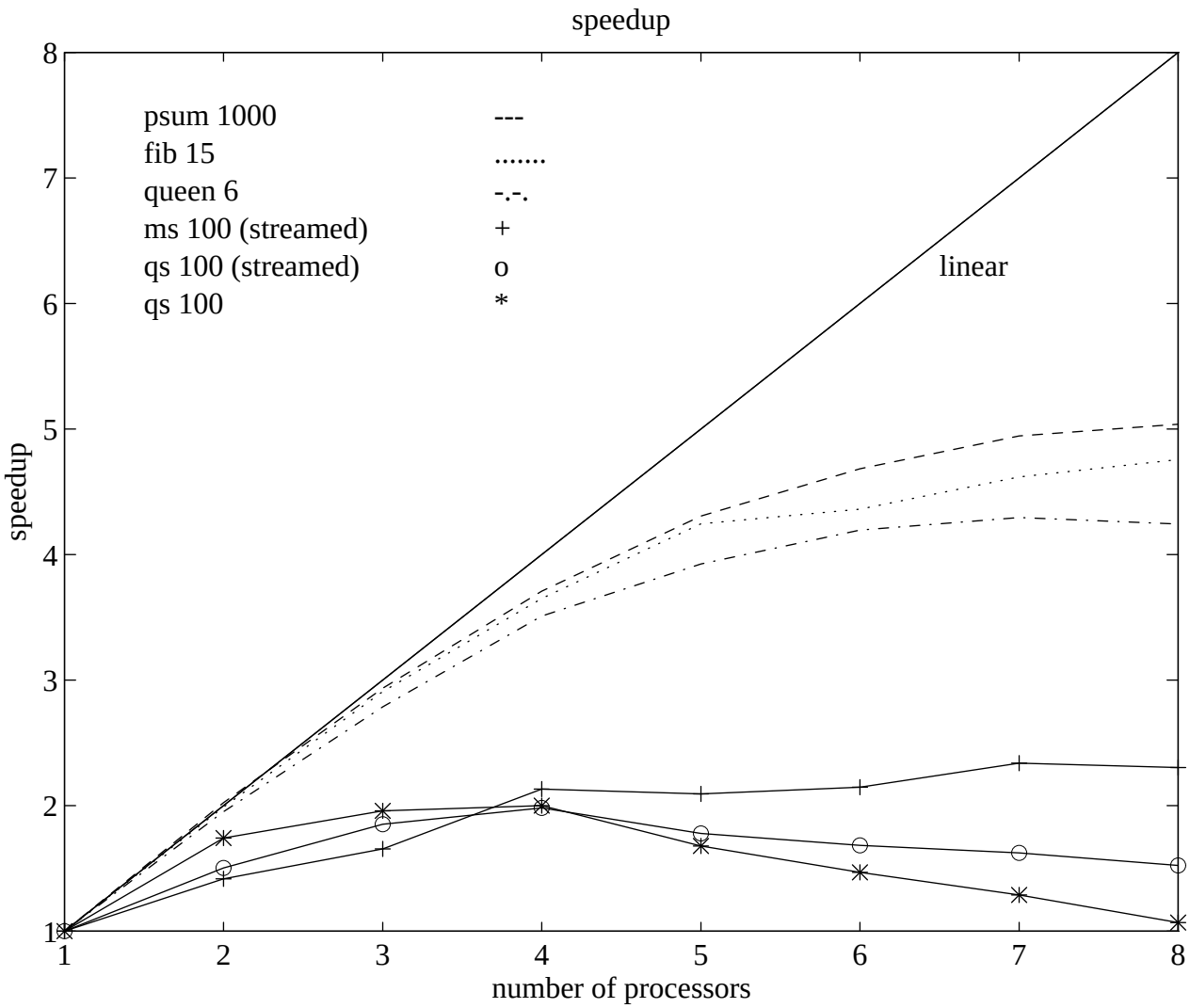


Figure 7.4: Speedup curve for all sample programs

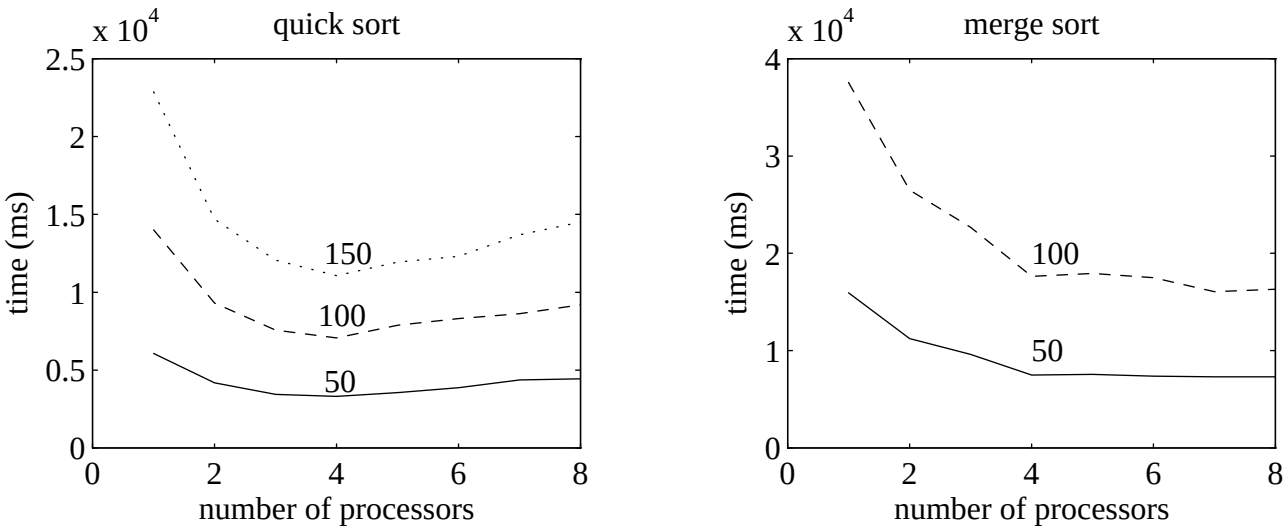


Figure 7.5: Performance for two stream based programs

Stream based programming is very resource consuming because data elements in a stream are implemented as light weight tasks. On a machine with a small number of processors or not supporting massive parallelism, the absolute performance suffers from insufficient resource or excessive parallelism. However, there will always be enough parallelism to keep all processors busy, so these programs scale up better. As we can see in figure 7.4, stream based quicksort has a larger speedup factor than normal parallel quicksort (although contention still affects it much when more processors provided), and merge sort's speedup is even larger. One reason that the speedup is not as satisfactory as for `psum` or `fib` is that processors spend more time busy waiting due to the heavy use of exclusive functions (though busy waiting implementation of exclusive functions decreases context switching and improves absolute performance).

The experiment tells that more non-trivial parallelism can be obtained via the use of streams. It also tells us, however, that stream based programming is more feasible when the language is implemented on a machine supporting massive parallelism. In any case, users are provided with the possibility to explore more useful parallelism in Pscheme.

7.3 Sequential Programs

In a practical computation, the absolute performance is as important as the scalability. We would like to observe the execution of sequential programs to see how efficient the codes that Pscheme compiler generates can be. To exclude any slow down factor in our run time system, we run the generated codes on top of a *sequential* run time system, which contains no global task queue or virtual processors, but only a dispatching loop.

Experiments are performed on a Sun3 (MC68020 CPU, 15M Hz clock) and a Sun4 (SPARC, 20M Hz clock) architecture respectively. Sequential sample programs are still the summation and Fibonacci. The Pscheme codes are in figure 7.6.

```
(define (sum  m n)                        ; sequential sum
  (if (eq? m n)
      m
      (+ m (sum (+ m 1) n))))



(define (fib n)                           ; Fibonacci number
  (if (<= n 3)
      n
      (+ (fib (- n 2)) (fib (- n 1)))))
```

Figure 7.6: Two sequential sample programs

	sun4	sun3	no. of processors on Ultra							
			1	2	3	4	5	6	7	8
sum 500	96	320	1750	1750	1812	1937	2125	2312	2563	2813
sum 1000	196	620	3375	3500	3500	3813	4187	5750	5187	5687
fib 10	14	40	188	250	188	250	250	375	313	375
fib 15	142	460	2375	2438	2500	2750	2938	3187	3563	4000

Table 7.1: Execution time (ms) of sequential programs on different platforms

	sum			fib		
	10000	20000	30000	20	21	22
Pscheme	1754	3590	5260	1492	2413	3908
SCM	766	1633	2216	417	716	1050

Table 7.2: Execution time (ms) of sequential programs through different implementations

We compare the performance with that of our run time system on Ultra computer (8M Hz MC68010 CPUs). The generated C codes for these samples are compiled through the same compiler: `cc`. From table 7.1, we see that sequential programs can have a slow down factor about 1.7 to 2 at worst. This explains how much the memory or bus contention on Ultra computer can affect the performance. As we have seen, the sequential programs run faster on faster platforms. But, how efficient are these sequential codes generated by the Pscheme compiler? We make a comparison here with SCM [40] in table 7.2 by running the same Scheme programs on the same platform (Sun3). It shows that the codes generated by the Pscheme-to-C translator and the C compiler will be two to four times slower.

The experiment tells us how much the absolute performance is affected by the contention on a parallel machine, and how much inefficiency is caused by *indirect* code generation through C for the purpose of portability. This means that the continuation-passing, closure-passing transform approach in the Pscheme compiler can produce intermediate representation that is efficient enough for the code generator. We expect competitive performance if machine codes can be generated directly.

In conclusion, Pscheme programs will have a good performance if

- the underlying machine supports massive parallelism,
- improved scheduling policy is provided in the run time system to control excessive parallelism, or
- efficient machine codes can be generated directly to better utilize machine registers and the cache memory.

While our goal is to add more expressive power in a parallel language, we have shown that Pscheme can also have satisfactory performance. Although experiments are per-

formed only on small inputs, we are able to improve the run time system in larger scaled machines to get a better speedup. This explains the feasibility of representing general control in a parallel applicative language. Therefore, programmers are provided with more expressive power without losing much performance.

Chapter 8

Related Work

Various parallel programming languages have been proposed. In this chapter, we will make comparisons among them based on language design issues. The work of designing Pscheme has benefited a lot from the concepts in these languages including two parallel Lisps Multilisp and Qlisp, the Actors model, Lucid, and proposals for parallel continuations.

8.1 Multilisp and Futures

Most parallel Lisps are shared memory languages with side effects because, with explicit parallelism provided, it is easier to model shared objects containing mutable states with side effects, and easier to express finer grained parallelism without worrying about copying shared data from one domain to another. Such a design decision also makes the language closest to the sequential Lisp counterpart, and hence is adopted by Multilisp, Qlisp and Pscheme. On the other hand, the way of introducing parallelism varies, and leads to different programming styles.

The concept of `pcall` is natural in parallel functional programming. Multilisp has another primary approach to parallelism – futures [1]. Futures enable programmers to

[1] The idea of futures had been around for a while when Multilisp was proposed, but Multilisp's future

express parallelism at different levels of granularity. It is a more fundamental construct than `pcall` in that it characterizes more parallel behaviors and in fact, it can implement `pcall`. On the other hand, it is high level enough to save programmers concerns of synchronization.

Since a future process commits the result it has computed [2] , a future value usually serves as a constant once it becomes available. When a language supports continuations, this becomes a problem. The code for the future process can be re-entered through the invocation of continuations captured inside. The interaction between continuations and futures makes the semantics for either construct hard to understand. It is suggested that the meaning of continuations are extended in the spirit that any mixed use of futures and continuations expects to have the same result as the sequential program without using futures. However, continuations, based on concepts of sequential computation, cause problems although there have been proposals to improve the interaction between futures and continuations [42]. That is why we feel the need to provide an alternative parallel extension for continuations.

Ports do provide a lower level construct than the future, and can implement more other parallel constructs. The future implemented with ports as shown in figure 3.7 restricts multiple returned results. This is guaranteed by a single-port captured at the future constructing point so that the first result that the future computes is committed as its value.

Exclusiveness is another concern of parallel languages. Multilisp has two low-level primitives `replace` and `replace-if-eq` supported by the Concert platform [3,31], a 32-processor multiprocessor. They are as low level as `test-and-set` and Ultra com-

---

is the first implementation on real parallel machines.

[2]Otherwise, there could be problems such as top level parallelism as mentioned in [42].

puter's `fetch-and-add`, and are actually used to implement semaphores, futures and so on. Pscheme's `exclusive` seems to provide a higher level mechanism to express shared objects with mutable values.

8.2 Qlisp

Qlisp's design goal is medium-grained parallelism [25,24,22]. It applies queue-based multiprocessing to support a variable number of processors, and allows users to check the task queue to control excessive parallelism.

Computation is in the form of tree-structured processes with AND/OR parallelism relating them. The OR parallelism is accomplished by a mechanism called *heavyweight future*, whose value is computed by several processes. When a value computed by a process satisfies an end test, the future is realized (or touched), and other processes associated with the future are killed.

Qlisp provides primitives for explicit process synchronization, process killing, and locking. Common Lisp's `catch` and `throw` are extended so that programmers can use `throw` to kill processes [3]. This is similar to passing a value to a single-port in Pscheme, but in Pscheme, no process is killed until processes themselves commit suicide when passing values to a closed single-port. In Qlisp, the `kill-process` primitive takes a future argument and kills all processes associated with it. Locks are available at the language level together with another locking construct – the *process closure* `qlambda`, which is the same as Pscheme's exclusive function.

A special feature of Qlisp is the **PMI** functions (*partially, multiply invoked functions*). The idea is to separate the coordination of the arrivals of arguments from the processing of

[3] This is actually the invocation of process continuations except that they cannot be *upward* continuations, and only serve as non-local exits.

them. Applying such a function can be in various forms depending on which parameter(s) is (are) submitted. That is, a function can be partially invoked as well as multiply invoked. This is similar to passing values to synchronous ports built by `pcall` in Pscheme. With PMI functions, streams can be implemented. Qlisp's approach differs from Pscheme in that data elements in a stream are submitted to PMI functions in the form of function application, while in Pscheme, explicit throws to the port are used.

Many constructs are introduced in Qlisp to express different concepts, but most of them can be explained in terms of ports. In Pscheme, ports have a more general meaning than Qlisp's continuations that are mainly used for non-local exits. Qlisp's PMI functions are examples of Pscheme's synchronous ports. Qlisp's design philosophy tends to provide language constructs for all concepts they consider important, but Pscheme does not emphasize any particular aspect. Instead, it allows users to build higher level facilities for various applications.

8.3 Actors

Actors [2] is a model of concurrent computation, which integrates functional programming and object based programming in the sense that computation is performed through message passing among concurrent active objects (*actors*) whose behaviors are *functional* and can be changed *functionally*.

Actors can be created dynamically, and communication between them is asynchronous so that maximal concurrency can be exploited. The language is expressive enough to model imperative data structures (such as stacks), recursive functions (such as factorial), and the nodes in a dataflow computing environment via the use of dynamically created actors.

Pscheme's approach to expressing message passing processes is similar to the Actors

model in the sense that exclusive function can also be created dynamically, and ports can be used to store incoming messages. Though Scheme takes a completely different approach from Actors, embedding ports in Pscheme makes it possible to apply message passing paradigms. Order preserving channels can be implemented in Actors as in Pscheme, where values thrown to a port by the same process preserves the sending order.

Actors define an orthogonal active construct in the message-passing paradigm, and therefore provide a fundamental model for concurrency. The message passing model is powerful enough to explain most concepts or activities in parallel computation. However, since message passing is intrinsically *operational*, it becomes a contradiction to the declarative programming style. Operational reasoning of parallel computation also makes programming more difficult. Pscheme's approach is to support the message passing paradigm in a *declarative* context. For example, there is no need to model recursion through message passing activations. [4] Provided with ports, Pscheme programmers are able to express middle grained message passing processes, and at the same time, keep the program declarative.

8.4 Lucid

Lucid is a data-flow language attempting to evolve from functional languages that are either Lisp-like or have single-assignments (such as ID [7]). It abstracts iteration and recursion into a higher level form of data streams. Data streams are viewed as lazy lists with element retrieving operations provided. Computation can be specified through recursive equations among stream identifiers while having an iterational semantics. It is a declarative language, but programmers are encouraged to think operationally in terms

[4] In the Actors model, each activation of a recursive function becomes a created actor which passes parameters in the forms of messages.

of dataflow.

Lucid is designed to completely depart from the Von Neumann model of computation, and expects to suit well on data-flow architectures. In fact, Lucid is a high level language but not a parallel language because programmers have no control of parallelism, which is contrary to Pscheme's design goal.

8.5 Other Models of Parallel Continuations

Hieb and Dybvig's *process continuation* [33] provides fine control of parallel processes in an environment where there is a tree structure relationship among them. That means, each process has a parent, who spawns it. A process continuation is a *partial* (or *local*) continuation representing the control state captured back to the spawn point. It is actually the parallel counterpart of Felleisen's $\mathcal{F}$-continuation. [15] Such partial continuations are *functional* in the sense that invoking them is like calling a function representing the captured computation. Capturing them suspends all tasks from below the spawn point to the capture point. Process continuations are useful to control the parallel search, implement parallel OR or kill other tasks. On other hand, the degree of parallelism is quite limited, and they only apply to tree structured parallelism.

PaiLisp [38,39] is another parallel extension of Lisp. It distinguishes single-use and multiple-use continuations. PaiLisp handles continuations according to the identity of the process that captures it (processes can be *spawned* by any expression.) When invoking a continuation, if the invoking process is the capturing process, sequential semantics applies (i.e. it is a *goto*). If not, the invoking process goes on, and the capturing process abandons its current computation (if any) and unconditionally jumps to the control point the continuation represents. PaiLisp may not have such precise control as process continuations, but a PaiLisp process has control of other processes in addition to those in the

same subtree.

Both versions of parallel continuations, together with the suggested continuation model in Multilisp, have one point in common. None of them allows the invocation of a continuation to fork a new thread within the invoking process so that activations are actually disjoint, and thus no race condition stated above will occur. Pscheme has no such limits, but it provides a `die` to control the interference among activations.

8.6 Pscheme's Approach to Parallelism

It is mentioned that we consider ports to be the natural parallel extension of continuations. When designing Pscheme, however, we have no intention to define ports so as to be compatible with continuations or programs with heavy use of continuations. This differs from previous attempts for parallel continuations, and we can thus come up with an orthogonal construct that is well defined across processes, and has its independent meaning.

Pscheme views parallel computation as a dynamic dataflow diagram. Processes do not have to have a tree-structure relationship. With the ports available, a process is able to communicate to one that would have been invisible in the process tree. Moreover, passing an object to a port has nothing to do with transferring control. That is why `throw` does not terminate the current thread (though it is a little bit misleading to continuation programmers).

Since no tree-structure relationship is among processes, Pscheme provides no process killing primitive except that suicides can be committed through `die`. Of course, exclusive functions are needed to enforce order and atomicity in the computation described by the dynamic dataflow diagram.

We think that this aspect (dynamic dataflow diagram) of parallel computation is

more general, and makes programmers more capable of modeling real world concurrent activities. On the other hand, Pscheme programmers are not suggested to commit to a particular programming paradigm. Concepts of ports have made programming much more flexible.

8.7 Comparison in Terms of Expressiveness

The design of Pscheme is actually an integration of the related work stated above. We start from a parallel Lisp supporting horizontal parallelism and exclusive functions, and then extend the sequential control mechanism, continuations, into the parallel world. Therefore, the proposal of ports makes possible parallelism in the message passing and the dataflow style.

In the example of stream based quicksort, processes can be created and passed around dynamically, which demonstrates the expressiveness as in the message passing Actor model. A partition cell can function in parallel with its producer and two consumers, which illustrates not only the vertical parallelism Pscheme can exploit, but the single-producer-multi-consumer relationship that many dataflow or non-strict/lazy languages have problems to express. Pscheme is more expressive in the sense that control flow can be relieved from the call-return structure, which expression languages are based on. Also, ports are more expressive than parallel continuations in the sense that they are clearly defined across thread boundaries.

Aside from those models motivating the design of Pscheme, there are many other parallel languages pursuing various programming paradigms. For example, the concurrent logic programming paradigm is not among those Pscheme attempts to model. In the family of concurrent logic languages, processes are used to solve conjunctive goals, and shared logical variables serve as communication channels. The implicit communication

	program structure	process model	communication model	aspect of parallelism
Ada	imperative	task	sync	msg passing
Actors	object based	actor	async	msg passing
Futures	applicative	thread [5]	implicit	speculative
Linda	–	implicit	shared memory / anonymous	shared memory
Concurrent Prologs	logic	implicit	shared variable	AND/OR /stream
Dataflow languages	declarative	implicit	implicit	dataflow
Pscheme	applicative	ex-function / thread	sync/async ports	various

Table 8.1: Paradigms for various parallel languages

among processes through these shared variables can generates sufficient parallelism to model stream based programming. (see [60]) For example, the stream based quicksort can be written in PARLOG declaratively. [28]

Linda [23,10] is a coordination language that provides a shared space (called *tuple space*) for tasks to communicate. Synchronization and dataflow relationship among tasks is enforced by tuple space operations implicitly. Linda puts little restriction on programming styles (, which actually depend on mother languages such as C that take care of the computation part,) because it only cares for the coordination part in parallel computation. Although a universal shared memory is expressive enough to model all kinds of process interaction, its aspect of parallelism is far from the message passing model in the sense that it ignores the locality of communication.

Table 8.1 illustrates the coordination paradigm in various parallel programming languages or models. We will not make comparisons between the applicative programming

[5] A thread is a sequential expression in evaluation.

paradigm and those adopted by other languages, but we would like to compare Pscheme's approach to parallelism with those of other models.

We view ports as higher level abstractions that contain information about queuing, synchronization, and subsequent computation. On the other hand, the implementation of ports makes them inexpensive (and thus low-level) enough to build up higher level constructs such as futures, rendezvous objects or asynchronous channels. Pscheme programmers are suggested to program in a *correct* style in the sense that ports should be used to model what is conceptually similar or close to them.

Speaking of the process model, ports can be as expressive as actors (first class processes). In terms of the communication model, ports serve as message queues, and can express synchronous or asynchronous communication, either by naming the receiver or anonymously. [6] As for the aspect of parallelism, ports can express all in the table but the AND/OR parallelism in most concurrent logic programming languages. The reason is that there is no tree structure relationship among Pscheme threads, and it is hard to define the parent-child relation of processes. Therefore, there is no way to kill processes implicitly as soon as their computation becomes redundant. The shared memory model is easy to express because Pscheme is a language with global name space, and we have mechanisms protecting shared variables.

To sum up, the concept of ports is not far from major coordination mechanisms in various parallel computation models. Pscheme provides optional additional expressiveness in a declarative programming context.

[6] Anonymous communication can be done by donating a channel process with an entry port for incoming messages and an exit port for read requests.

Chapter 9

Future Work and Conclusion

9.1 Future Work

Most of the work to be done in the future is regarding the Pscheme implementation. There will be improvement in Pscheme compiler as well as the run time system. System interfaces will also be implemented to facilitate programming.

9.1.1 Compiler Optimization

Uniform closure conversion may generate expensive code in that not every function needs to be converted into closure-passing style, and not every function application needs to first select field zero from a closure record to get the code to apply. The compiler can perform some dataflow analysis to decide whether to convert a closed function into closure-passing style so that some run time closure operations can be saved.

The dispatching loop is expensive because a C procedure call happens each time a compiled function is dispatched. If the compiler can generate code that transfer control to a known function through the label (i.e. via a `goto`) without going back to the dispatcher, many procedure calls can be saved. [1] This technique is used in the SML-to-C translator

[1] In a CPS transformed, closure converted program, instruction execution is all through the procedure

by Darditi, Lee and Acharya [61]. The dispatcher is still needed for functions that are not known by the compiler.

As is mentioned in chapter 7, to get better sequential performance, assembly code can be generated for various platforms, and machine registers will be allocated directly.

9.1.2 Run Time System Improvement

We would also like to have a more portable run time system that does not rely on the support of *fetch-and-add* or parallel queues. Queues that serialize requests can be used to implement ports because the contention among different threads is not high, but the global task queue had better be implemented without serialization, or can even be replaced by hierarchical queues that decrease contention. For instance, in Multilisp, each process has a local task queue, and there is a protocol for processes to retrieve non-local tasks so as to balance the load.

Dynamic control of excessive parallelism may also be implemented in the run time system with an alternative task scheduling policy such as in Multilisp.

9.1.3 I/O and System Interface

It is desired to provide I/O functions in Pscheme as well as other system interface primitives such as dynamic loading facility, foreign language interface, timing instructions and so on.

9.2 Conclusion

This research is an attempt to reason about control in parallel computation. Through our experiments in the parallel applicative language Pscheme, we found that various control

calls on continuations. A simple control transfer implemented by a calling sequence of several instructions results in a large slowdown factor, so it should be avoided as much as possible.

concepts such as synchronization, blocking and ordering can be explained in terms of ports, which we claim to be a general control mechanism.

Modeling computation as a dynamic dataflow net with the tangible links enables us to express different aspects of parallel computation, which makes it possible to apply different programming paradigms in one language. Because of the expressive power, we conclude that the port plays the same role in parallel computation as the continuation does in the sequential world.

Pscheme indicates how expressive a parallel language can be. At the same time, we claim that exposing control details in parallel computation through ports makes the programs no harder to understand than sequential programs with heavy use of continuations. In other words, it shows the extent of difficulty for a programmer to reason about or model parallel computation.

Is programming in Pscheme difficult? No, if higher level control libraries are provided. As a matter of fact, it makes it a lot easier to model other aspects of parallel computation (such as pipelining or message passing) for an applicative language programmer. More expressive power without sacrificing the advantage of the original declarative style never makes programming more difficult.

How efficient are Pscheme programs? We have seen good speedups for parallel programs. Dataflow programs, however, have a less satisfactory absolute performance. Also, programs with fine-grained parallelism are not efficient enough. Observing the performance of sequential Pscheme programs and the scalability of parallel programs, we think that, with architectural support, Pscheme can have performance as good as other dataflow languages or languages supporting fine-grained parallelism.

Our final conclusion is: *representing general control in parallel computation is feasible.*

Appendix A

Transforming The Parallel Sum Program

A.1 Input Program

```
(define (sum m n)
 (cond ((eq? m n) m)
       ((eq? n (+ 1 m)) (+ m n))
       (else (letrec ((x (/ (+ m n) 2)))
               (pcall + (sum m x) (sum (+ 1 x) n))))))

(sum 1 250)
```

A.2 Continuation Passing Style: (hoisted)

```
LETREC
 f23 = LAMBDA v23 v24 f24
```

```
        PRIM + v23 v24   v25
        APPLY f24 v25
SUM = LAMBDA M N f1
        LETREC
         f3 = LAMBDA v2
                APPLY f1 v2
         PRIM EQ? M N   v5
         IF v5
         THEN APPLY f3 M
         ELSE LETREC
               f7 = LAMBDA v6
                      APPLY f3 v6
               PRIM + 1 M   v11
               PRIM EQ? N v11   v12
               IF v12
               THEN PRIM + M N   v15
                    APPLY f7 v15
               ELSE LETREC
                     X = PRIM + M N   v19
                         PRIM / v19 2   v21
                         BIND X v21
                     s1 = SYNC f23 2 f7
                     p1 = BUILD-PORT s1 1
                     p2 = BUILD-PORT s1 2
                     PARA
```

```
                        LETREC
                         f28 = LAMBDA v29
                                 PASS v29 p1
                         APPLY SUM M X f28
                        LETREC
                         f34 = LAMBDA v35
                                 PASS v35 p2
                         PRIM + 1 X   v33
                         APPLY SUM v33 N f34
 f37 = LAMBDA v39
         END    v39
 APPLY SUM 1 250 f37
```

A.3 Closure Passing Style: (hoisted)

```
LETREC
 f23' = LAMBDA f23'' v23 v24 f24
          LETREC
           f24' = SELECT f24 0
           PRIM + v23 v24   v25
           APPLY f24' f24 v25
 f3' = LAMBDA f3'' v2
         LETREC
          f1' = SELECT f3'' 1
          f1'' = SELECT f1' 0
          APPLY f1'' f1' v2
```

```
f7' = LAMBDA f7'' v6
        LETREC
         f3'''' = SELECT f7'' 1
         f3''''' = SELECT f3'''' 0
         APPLY f3''''' f3'''' v6
f28' = LAMBDA f28'' v29
         LETREC
          p1' = SELECT f28'' 1
          PASS v29 p1'
f34' = LAMBDA f34'' v35
         LETREC
          p2' = SELECT f34'' 1
          PASS v35 p2'
SUM' = LAMBDA SUM'' M N f1
          LETREC
           f23''' = OFFSET SUM'' -1
           f3 = CLOSURE f3' f1
           PRIM EQ? M N   v5
           IF v5
           THEN LETREC
                  f3''' = SELECT f3 0
                  APPLY f3''' f3 M
           ELSE LETREC
                  f7 = CLOSURE f7' f3
                  PRIM + 1 M   v11
```

```
                    PRIM EQ? N v11   v12
                    IF v12
                    THEN LETREC
                          f7''' = SELECT f7 0
                          PRIM + M N   v15
                          APPLY f7''' f7 v15
                    ELSE LETREC
                          X = PRIM + M N   v19
                              PRIM / v19 2   v21
                              BIND           X v21
                          s1 = SYNC f23''' 2 f7
                          p1 = BUILD-PORT s1 1
                          p2 = BUILD-PORT s1 2
                          PARA
                            LETREC
                             f28 = CLOSURE f28' p1
                             SUM''' = SELECT SUM'' 0
                             APPLY SUM''' SUM'' M X f28
                            LETREC
                             f34 = CLOSURE f34' p2
                             SUM'''' = SELECT SUM'' 0
                             PRIM + 1 X   v33
                             APPLY SUM'''' SUM'' v33 N f34
f37' = LAMBDA f37'' v39
          END    v39
```

```
 f23 = CLOSURE f23' SUM' f37'

 SUM = OFFSET f23 1

 f37 = OFFSET f23 2

 SUM''''' = SELECT SUM 0

 APPLY SUM''''' SUM 1 250 f37
```

A.4 Thread Abstraction

```
LETREC
 f23' = LAMBDA f23'' v23 v24 f24
          LETREC
           f24' = SELECT f24 0
           PRIM + v23 v24   v25
           APPLY f24' f24 v25
 f3' = LAMBDA f3'' v2
         LETREC
          f1' = SELECT f3'' 1
          f1'' = SELECT f1' 0
          APPLY f1'' f1' v2
 f7' = LAMBDA f7'' v6
         LETREC
          f3'''' = SELECT f7'' 1
          f3''''' = SELECT f3'''' 0
          APPLY f3''''' f3'''' v6
 f28' = LAMBDA f28'' v29
          LETREC
```

```
          p1' = SELECT f28'' 1
          PASS v29 p1'
f34' = LAMBDA f34'' v35
         LETREC
          p2' = SELECT f34'' 1
          PASS v35 p2'
f38 = LAMBDA SUM'''''' p1'' f28''' X' M'
        LETREC
         f28 = CLOSURE f28''' p1''
         SUM''' = SELECT SUM'''''' 0
         APPLY SUM''' SUM'''''' M' X' f28
f39 = LAMBDA SUM''''''' p2'' f34''' X'' N'
        LETREC
         f34 = CLOSURE f34''' p2''
         SUM'''' = SELECT SUM''''''' 0
         PRIM + 1 X''   v33
         APPLY SUM'''' SUM''''''' v33 N' f34
SUM' = LAMBDA SUM'' M N f1
          LETREC
           f23''' = OFFSET SUM'' -1
           f3 = CLOSURE f3' f1
           PRIM EQ? M N   v5
           IF v5
           THEN LETREC
                 f3''' = SELECT f3 0
```

```
                APPLY f3''' f3 M
            ELSE LETREC
                f7 = CLOSURE f7' f3
                PRIM + 1 M   v11
                PRIM EQ? N v11   v12
                IF v12
                THEN LETREC
                    f7''' = SELECT f7 0
                    PRIM + M N   v15
                    APPLY f7''' f7 v15
                ELSE LETREC
                    X = PRIM + M N   v19
                        PRIM / v19 2   v21
                        BIND           X v21
                    s1 = SYNC f23''' 2 f7
                    p1 = BUILD-PORT s1 1
                    p2 = BUILD-PORT s1 2
                    PARA
                      APPLY f38 SUM'' p1 f28' X M
                      APPLY f39 SUM'' p2 f34' X N
f37' = LAMBDA f37'' v39
         END    v39
f23 = CLOSURE f23' SUM' f37'
SUM = OFFSET f23 1
f37 = OFFSET f23 2
```

```
SUM''''' = SELECT SUM 0

APPLY SUM''''' SUM 1 250 f37
```

Appendix B

Internal Representation of Pscheme Objects

```
typedef unsigned int   heap_ptr;          /* index of the heap array */
typedef int            (*function_ptr)();

struct b_i_l_s                            /* bool_int_label structure */
{
  heap_ptr forwarding;
  int val;
};

struct char_s
{
  heap_ptr forwarding;                    /* forwarding is for GC */
  char val;
};
```

```
struct real_s
{
  heap_ptr forwarding;
  float val;
};

struct string
{
  heap_ptr forwarding;
  short length;
  char *val;
};


struct lambda
{
  heap_ptr forwarding;
  unsigned int is_prim:1;
  int hv_no;
  function_ptr code;
  bw_sem_t *sem;
};
```

```
struct cons_cell
{
  heap_ptr car;           /* forwarding */
  heap_ptr cdr;
};

struct cell
{
  heap_ptr point_to;      /* forwarding */
};

struct closure
{
  heap_ptr fields;        /* forwarding */
  short field_no;
  short offset;           /* non-negative */
};

struct vector
{
  heap_ptr val;           /* forwarding */
  int length;
};
```

```
struct synchronizer
{
  unsigned int arity:4;
  unsigned int count:4;
  heap_ptr destination;   /* forwarding */
  heap_ptr cont;
  bw_rwlock_t *lock;
  queue_t **oqs;          /* pointer to shared bytes */
  short *qlens;           /* pointer to shared bytes */
};

struct async_port
{
  int single_closed;      /* 2-byte single and 2-byte closed */
  heap_ptr destination;   /* forwarding */
  queue_t *queue;
};

struct sync_port
{
  short index;
  heap_ptr sync;          /* forwarding */
};
```

```
union u_obj
{
  struct b_i_l_s label;
  struct b_i_l_s int_num;
  struct b_i_l_s bool;
  struct char_s ch;
  struct real_s real_num;
  struct cell pointer;
  struct cons_cell pair;
  struct lambda *lbd;
  struct closure clsr;
  struct string str;     /* also used by quoted symbol */
  struct vector vct;
  struct sync_port s_port;
  struct async_port a_port;
  struct synchronizer *sync;
};



typedef struct
  {
    unsigned int type: 4;
    union u_obj  content;
  }  obj;
```

Bibliography

[1] W.B. Ackerman and J.B. Dennis. VAL - a calue-oriented algorithmic language. Technical Report LCS/TR-218, MIT lab for computer science, 1979.

[2] Gul Agha. ACTORS: A model of concurrent computation in distributed systems. Technical report, MIT Press, 1986.

[3] T. Anderson. The design of a multiprocessor development system. Technical Report TR-279, MIT lab for computer science, 1982.

[4] A. W. Appel. Continuation-passing, closuring-passing style. In *Annual ACM Symposium on Principles of Programming Languages*, 1989.

[5] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In G. Goos and J. Hartmanis, editors, *Parallel Architectures and Languages Europe*. Springer-Verlag, 1987.

[7] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. In J. Fasel and R.M. Keller, editors, *Proceedings of Workshop on Graph Reduction*. Springer-Verlag, 1987.

[8] D. Berry, R. Milner, and D.N. Turner. A semantics for ml concurrency primitives. In *Annual ACM Symposium on Principles of Programming Languages*, 1992.

[9] Albert Cahana, Jan Edler, and Edith Schonberg. *How to write Parallel Programs for the NYU Ultracomputer Prototype*, 1986.

[10] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[11] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11), 1970.

[12] Jerome Chiabout. *Timing Programs on the NYU Ultracomputer*, 1986.

[13] O. Danvy and A. Fiinsky. Abstracting control. In *ACM Conference on Lisp and Functional Programming*, 1990.

[14] R.K. Dybvig and R. Hieb. Engines from continuations. *Computer Languages*, 14(2), 1989.

[15] M. Felleisen. The theory and practice of first-class prompts. In *Annual ACM Symposium on Principles of Programming Languages*, 1988.

[16] M. Felleisen. Modeling continuations without continuations. In *Annual ACM Symposium on Principles of Programming Languages*, 1991.

[17] M. Felleisen, D.P. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *Symposium on login in computer science*, 1986.

[18] A. Filinski. Linear continuations. In *Annual ACM Symposium on Principles of Programming Languages*, 1992.

[19] J. Franco, D.P.Friedman, and D. Johnson. Multiway streams in Scheme. *Computer Languages*, 1989.

[20] D. Friedman and M. Wand. Reification: Reflection without metaphysics. In *ACM Conference on Lisp and Functional Programming*, 1984.

[21] R.P. Gabriel and J. McCarthy. Queue-based multiproccessing Lisp. In *ACM Conference on Lisp and Functional Programming*, 1984.

[22] R.P. Gabriel and J. McCarthy. Qlisp. In J.S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*. Kluwer Academic Publishers, 1988.

[23] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in linda. In *International Conference on Parallel Processing*, 1985.

[24] R. Goldman and R.P. Gabriel. Qlisp: Experience and new directions. *Parallel Programming: Experience with Applications, Languages and Systems, SIGPLAN Notices*, 23(9), 1988.

[25] R. Goldman, R.P. Gabriel, and C. Sexton. Qlisp: An interim report. In *US/JAPAN Workshop on Parallel Lisp, Lecture Note in Computer Science 441*. Springer-Verlag, 1990.

[26] A. Gottlieb et al. The NYU Ultracomputer – designing an MIMD shared memory computer. *IEEE transactions on computers*, C-32(2), 1983.

[27] A. Gottlieb, B. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), 1983.

[28] Steve Gregory. *Parallel Logic Programming in PARLOG.* Addison Wesley, 1987.

[29] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[30] R. Halstead. New ideas in parallel lisp: Language design, implementation, and programming tools. In *US/JAPAN Workshop on Parallel Lisp, Lecture Note in Computer Science 441.* Springer-Verlag, 1990.

[31] R. Halstead, T. Anderson, R. Osborne, and T. Sterling. The design of a multiprocessor development system. In *13th annual international symposim on computer architecture*, 1986.

[32] C.T. Haynes, D.P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer languages*, 11(1), 1986.

[33] R. Hieb and R.K. Dybvig. Continuations and concurrency. In *ACM Conference on the Principles and Practice of Parallel Programming*, 1990.

[34] R. Hieb, R.K. Dybvig, and C. Bruggeman. Representing control in the presence of first-calss continuations. In *ACM Conference on Programming Language Design and Implementation*, 1990.

[35] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

[36] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[37] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict puely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1992.

[38] T. Ito and M. Matsui. A parallel lisp language PaiLisp and its kernel specification. In *US/JAPAN Workshop on Parallel Lisp, Lecture Note in Computer Science 441.* Springer-Verlag, 1990.

[39] T. Ito and T. Seino. On PaiLisp continuation and its implementation. In *ACM SIGPLAN Workshop on Continuations*, 1992.

[40] Aubrey Jaffer. SCM - a scheme language interpreter. personal communication.

[41] Thomas Johnson. Lambda-lifting: transforming programs to recursive equations. In *Conference on Functional Programming Languages and Computer architecture*, 1985.

[42] M. Katz and D. Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *ACM Conference on Lisp and Functional Programming*, 1990.

[43] R. Keller. Rediflow multiprocessing. In *IEEE COMPCON*, 1984.

[44] R. Keller. Simulated performance of a reduction-based multiprocessor. *IEEE transactions on computers*, 17(7), 1984.

[45] R.R. Kessler and M.R. Swanson. Concurrent scheme. In *US/JAPAN Workshop on Parallel Lisp, Lecture Note in Computer Science 441*. Springer-Verlag, 1990.

[46] D.A. Kranz, R.H. Halstead, and E. Mohr. Mul-T: a high performance parallel Lisp. In *US/JAPAN Workshop on Parallel Lisp, Lecture Note in Computer Science 441*. Springer-Verlag, 1990.

[47] David A. Kranz. *ORBIT: an optimizing compiler for Scheme*. PhD thesis, Yale University, 1988.

[48] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4), 1964.

[49] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, Brussels, Be lgium, 1987.

[50] Robin Milner. A proposal for standard ML. In *ACM Conference on Lisp and Functional Programming*, 84.

[51] R. Osborne. Speculative computation in Multilisp. In *US/JAPAN Workshop on Parallel Lisp, Lecture Note in Computer Science 441*. Springer-Verlag, 1990.

[52] K. Pingali and Arvind. Eficient demand-driven evaluation. *ACM Transactions on Programming Languages and Systems*, 7(2), 1985.

[53] G.D. Plotkin. Call-by-name, call-by-value and lambda calculus. *Theoretical Computer Science*, 1, 1975.

[54] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5, 1977.

[55] J.H. Reppy. An operational semantics of first-class synchronous operations. Technical report, Department of Computer Science, Cornell University, 1991.

[56] J.C. Reynolds. GEDANKEN – a simple typeless laguage based on the principle of completeness and the reference concept. *Comminications of the ACM*, 13(5), 1971.

[57] J. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4), 1980.

[58] D. Sitaram and M. Felleisen. Reasoning with continuations II: full abstraction for models of control. In *ACM Conference on Lisp and Functional Programming*, 1990.

[59] B. Smith. Reflection and semantics in LISP. In *11th Annual ACM Symposium on Principl es of Programming Languages*, 1984.

[60] A. Takeuchi and K. Furukawa. Parallel logic programming languages. *Concurrent Prolog*, 1987.

[61] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM letters on programming languages and systems*, 1(2), 1992.

[62] NYU Ultra Lab. *NYU Ultracomputer Unix Programmer's Manual: symunix 1.1*, 1986.

[63] Wadge and Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[64] S.C. Wray and J. Fairbairn. Non-strict languages – programming and implementation. *The Computer Journal*, 32(2), 1989.