EFFICIENT ARRAY UPDATE ANALYSIS OF STRICT FUNCTIONAL LANGUAGES

by

A.V.S. SASTRY

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 1994

"Efficient Array Update Analysis of Strict Functional Languages," a dissertation prepared by A.V.S. Sastry in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

William D Clinger
Chair of the Examining Committee

31 May 1994
Date

Committee in charge:
Dr. William D. Clinger, Chair
Dr. Zena M. Ariola
Dr. Arthur M. Farley
Dr. Richard M. Koch

Accepted by:

Vice Provost and Dean of the Graduate School

An Abstract of the Dissertation of

A.V.S. Sastry for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken June 1994

Title: EFFICIENT ARRAY UPDATE ANALYSIS OF STRICT FUNCTIONAL LANGUAGES

Approved: William D Clinger

Dr. William D Clinger

The usefulness of functional programming languages for large scale software development has been limited by the array update problem: Since there is no notion of state in these languages, modifying an array at a single index requires creating a new copy of the entire array, which leads to unacceptable performance degradation.

An array update can, however, be implemented in constant time without copying if there are no future uses of the old array.

This thesis presents the first practical interprocedural update analysis algorithm for strict first-order functional languages with arrays of scalars. The analysis runs in polynomial time, and in linear time for typical programs. All previous algorithms of this kind require exponential time in the worst case. The algorithm does not assume any fixed evaluation order but derives a good order that maximizes opportunities for destructive (non-copying) updating. The simplicity of the algorithm also makes it adaptable to separate compilation.

This thesis also describes a parallel functional language with new operations on arrays for expressing divide and conquer algorithms, and presents an extension of the basic algorithm for that language. The analysis has polynomial time complexity even for parallel evaluation, and is the first practical algorithm for interprocedural update analysis of parallel functional programs. The analysis is so effective that it removes all copying in parallel functional programs for many scientific applications including gaussian elimination with and without partial pivoting, LU, Cholesky and QR factorizations, and multigrid and relaxation algorithms for solving partial differential equations numerically. In cases where a copy cannot be eliminated, the compiler can advise the user about the source of the problem.

The thesis describes a new update operation for specifying a collection of updates on an array, which subsumes monolithic arrays as provided by most functional languages. It also considers the problem of non-flat or nested but non-recursive arrays and some of the difficulties introduced by non-flatness, and presents an extension of the algorithm for non-flat arrays.

Another contribution of the thesis is the implementation of a compiler for the proposed parallel functional language, and a runtime system for a shared memory multiprocessor. It describes the compiler and the runtime system and presents some preliminary performance results of the implementation on a Sequent Symmetry, a bus-based shared memory multiprocessor.

The thesis considers the implications of these algorithms for language design, and argues that programmers will be able to write efficient programs by relying on update optimization.

CURRICULUM VITA

NAME OF THE AUTHOR: A.V.S. Sastry

PLACE OF BIRTH: Visakhapatnam, India

DATE OF BIRTH: August 2, 1965

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Indian Institute of Science
Institute of Technology, Banaras Hindu University

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 1994,
University of Oregon
Master of Science (Engg.) in Computer Science, 1988,
Indian Institute of Science, Bangalore, India
Bachelor of Technology in Electronics Engineering, 1986,
Institute of Technology, B.H.U., Varanasi, India

AREAS OF SPECIAL INTEREST:

Programming Languages, Compilers, Parallel Computing

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow and Instructor, Department of Computer and Information Science, University of Oregon, Eugene, 1989-93

R&D Engineer, CMC Ltd., Secunderabad, India, 1989

AWARDS AND HONORS:

University of Oregon Doctoral Research Fellowship, 1993.
ACM SIGPLAN Student Travel Award, June 1993 and June 1994.
Research Scholarship, Indian Institute of Science, India, 1986-88.
National Merit Certificate, India, 1981.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Will Clinger for his guidance, encouragement, and support during the past five years. He has been instrumental in the culmination of our research into this thesis. I wish to thank Zena Ariola, Art Farley, and Richard Koch, who read the thesis meticulously and made several useful suggestions. Thanks to Bart and Lars for their comments on Chapter IV.

Andrzej and Chris taught me several classes in Data Structures, Algorithms, and Complexity Theory. I enjoyed their teaching immensely and would like to thank them for kindling my interest in theory.

I wish to thank Jan, Betty, Nagwa, and Barbara, who have helped me in many a ways at the office downstairs.

It has been wonderful to have Jan (pronounced as Yan) as my office-mate for the last four years, sharing our successes and disappointments together. Unlike me, he always kept his promise of bringing Danish cookies to the office. Thank you Jan. Srinath has been a great roommate. I wish to thank him, especially for having taken care of me after my ice skating adventure.

I wish to thank Sanjay and Sharmila with whom I stayed for one week when I came to Eugene five years back. I would also like to thank Susan and Richard for inviting me to their place on almost every Thanksgiving and Christmas and on several other occasions. Among other friends Chandra, Christof, Don, Vivienne, Ferenc, Judit, Joydip, Sreeram, Renga, Taku, and Xiaoxiong have made my stay at Eugene even more pleasant.

I am thankful to the Graduate School for their generous support last year through the University of Oregon Doctoral Research Fellowship. I would also like to acknowledge the financial support provided by the Computer Science Department and SIGPLAN PAC Committee for attending several conferences. Thanks are also due to Zena and Will for their support for my trip to Copenhagen last year.

I wish to thank all the folks back home who believed that I would definitely get a PhD.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Introduction

Today's programming languages, for instance Fortran, were designed for sequential machines. In these languages a program is basically a sequence of instructions that read or modify the contents of computer memory. These languages are called imperative languages because the programmer specifies *how* the computation is to be carried out as opposed to *what* is to be computed. Imperative programming is based on side-effects, *i.e.* an expression not only returns a value but may also change the contents of a memory location.

Programming multiprocessors with imperative languages is hard. Concurrent reads and writes to memory make these languages non-deterministic. The programmer has to worry about the details of resource management such as dividing the work among multiple processors, process synchronization, and avoidance of data races because of concurrent reads and writes. Debugging a parallel imperative program is a herculean task.

Functional languages, on the other hand, allow the programmer to specify *what* is to be computed instead of *how* it is to be computed. Functional languages forbid side-effects: there is no notion of memory or assignment. A variable in a functional language is a name for a value just as in mathematics. A functional program itself is no more than a mathematical definition of a partial function. The details of computation

that are not essential to the description of the algorithm, such as the use of memory and processors and the exact sequence of execution, are relegated to the compiler and the runtime system.

Functional languages offer several advantages over imperative languages. Side-effect-freeness and declarative semantics make functional programs easier to read, reason about, and maintain. These languages are also ideal for writing parallel programs because data dependencies are the only constraints on program execution. Two subcomputations that do not depend on one another can be computed in any order, even in parallel, without affecting the result [9, 49]. This Implicit parallelism makes parallel programming a much easier task. The problem of parallel debugging, a nightmare for the programmer, does not exist with functional languages. Because of the deterministic semantics, a functional program can be debugged on a single processor and run on multiple processors.

In spite of all these advantages, functional languages have not been accepted for mainstream programming because of inefficiency caused by excessive copying of data. One of the most serious problems in these languages is the aggregate update problem. Computation that involves an incremental change to a large data structure such an array requires copying the entire structure, causing severe performance degradation. Therefore, to make functional languages practical we need an efficient solution to the update problem.

This thesis provides the first practical solution to the array update problem for a class of functional languages known as first-order strict functional languages with arrays of scalars (*i.e.* flat arrays). These languages, though restrictive without recursive types and higher-order functions, are as powerful as Fortran — the programming

language that is most widely used in the scientific community. We believe that the results of this thesis should encourage users of Fortran not to dismiss pure functional languages on the basis of the common belief that these languages are inherently inefficient because of the copying problem. This thesis provides an effective solution to this problem.

The solution presented is based on a compile-time analysis of the source program. We demonstrate that the analysis eliminates all unnecessary copying in several common scientific programs such as matrix factorizations, direct and iterative equation solvers, and multigrid methods for solving partial differential equations.

1.2 The Aggregate Update Problem

Consider multiplying a row of an $n \times n$ array by a constant k. The row can be scaled by performing n updates of the array. In Fortran or C, these updates can be performed by assignment to the array. The original array is destroyed in the process. The programmer treats the array as storage for the resulting array and has to be aware that the original array is not needed anywhere else. We call such updates *destructive*. The complexity of the algorithm is $O(n)$.

Because functional languages forbid side-effects, each update requires creation of a new copy of the original array. The row scaling operation creates n $n \times n$ arrays, with the last one being the result. The complexity of the algorithm is $O(n^3)$, two orders of magnitude slower than the imperative version. The problem is how to avoid this copying.

An update of an array can be done in-place if there are no future uses of the original array. In the case of imperative languages like Fortran or C, it is the responsibility of the programmer to know that the original array is not used anywhere

else. In functional languages, this burden is shifted to the compiler or the runtime system. We would like for the compiler to analyze the program and to convert every non-destructive or copying update into a destructive update, if possible. If such a compiler can be written, then we can extract Fortran-like efficiency from functional programs without sacrificing the ease of programming. This thesis describes such a compiler.

Next we discuss various aspects of functional languages and their implications for this problem.

1.2.1 Strictness vs Non-strictness

Functional languages can be classified as strict or non-strict. A function f is *strict* if $f(\perp) = \perp$, where $\perp$ represents the undefined value of a non-terminating computation. In other words, non-termination of an argument to a strict function implies non-termination of the call. Strict functions can be implemented efficiently by evaluating the arguments before calling the function.

A function is non-strict if the non-termination of an argument to the function does not imply the non-termination of the function call. Therefore an argument to a function has to be passed unevaluated. It is evaluated subsequently only if it is needed to compute the result of the call. Non-strict languages are more expressive than strict languages in their ability to represent infinite data structures. However, non-strict languages are harder to implement [58, 74]. For a survey of functional languages see [10, 26].

One of the difficulties with update analysis of non-strict languages is that the relative order of evaluation of expressions in a non-strict language cannot be determined precisely at compile-time. Computing the liveness of an aggregate is harder

because it requires the order of evaluation information which is not known at compile time [14, 12]. In the rest of this thesis we work with strict languages. We believe our analysis is applicable to non-strict programs where the strictness information is available.

1.2.2 Order of Evaluation

In a strict language we know that the arguments of a function are evaluated before the call. But the arguments themselves can be evaluated in any order. The current solutions to the array update problem, however, rely on a fixed evaluation order [12, 45]. Fixing an order of evaluation for the arguments to a function *a priori* reduces the opportunity for destructive updating. A compiler should instead derive a good order of evaluation that increases the opportunities for destructive updating. In Chapter II we show how our compiler derives a good evaluation order.

1.2.3 Parallelism

Parallelism and destructive updating seem to be in conflict with each other. If two expressions are evaluated in parallel, the variables in one are necessarily live while the other expression is being evaluated. Thus updates to arrays referred by expressions cannot be performed destructively whereas, in a sequential evaluation it might be possible to find an ordering that makes an update in one of the expressions destructive. In Chapter III we show how we reconcile parallelism with destructive updating.

1.2.4 Non-flat or Nested Aggregates

With nested aggregates, one can store one aggregate inside another aggregate. Allowing nested aggregates makes the detection of liveness more difficult. Computing the liveness of an aggregate requires computing the liveness of all those aggregates that have references to it. Therefore we have to compute aggregates potentially stored inside another aggregate during program execution, as described in Chapter V.

1.2.5 Higher-Order Functions

In higher-order languages, functions can be passed as arguments to functions and returned as results of functions. Static analysis of languages with higher-order functions is generally more difficult than that of first-order languages because the call graph of a program is not known in the case of higher-order languages. The flow analysis required to determine the liveness of an aggregate becomes difficult because aggregates can be captured by function closures. In this thesis we work with first-order languages.

1.3 Importance of Update Optimization

Update optimization is a very significant optimization because conversion of a non-destructive update to a destructive update can improve the performance of a program by orders of magnitude as discussed earlier.

The drastic improvement in execution time of the matrix row scaling program, written in scheme, with update optimization over the unoptimized version is shown in Table 1. These speedups are not surprising, but the real question is whether a compiler can detect destructive updates. In this thesis we describe an effective and

Table 1: Effectiveness of the Update Optimization

Matrix Size	*Time(millisecs.)*	
	destr upd	*copying upd*
10x10	1.3	10
20x20	2.48	40
30x30	3.72	100
40x40	4.90	220
50x50	6.21	400

computationally practical compile-time update analysis algorithm.

1.4 Related Work

The aggregate update problem is a very important problem and is being addressed by three avenues of research. One approach requires the functional programmer to assert that it is safe to update an array by side effect (because the array is dead), leaving it to the compiler to verify that assertion and to report a static error if the assertion cannot be verified [40, 77]. We call this *verification*. The other approach is to leave it to the compiler to detect updates that can be implemented by side effect [12, 13, 16, 27, 29, 31, 36, 37, 41, 45, 63, 64, 66, 67]. We call this *optimization*. Verification is potentially easier, but optimization is more flexible. Since an optimizing compiler can warn of cases in which it is unable to implement an update by side effect, and the programmer can be given control over those warnings, including the ability to specify whether such warnings should be treated as fatal errors, optimization spans a range of compile-time behaviors up to and including verification.

The third approach is based on program structuring using monads [76, 78], mutable abstract datatypes [46], and continuations. The use of monads or the abstract datatype operations guarantees that the aggregate can be implemented efficiently. We

discuss the related work in detail in Chapter VI.

1.5 Contributions of the Thesis

This thesis describes the first practical interprocedural update analysis algorithm for strict first-order functional languages with arrays of scalars. Unlike previous methods, the algorithm does not assume any order of evaluation, but derives a good order that increases the opportunity for destructive updating. The algorithm is efficient and runs in polynomial time – in linear time for typical cases. All previous algorithms of this nature have required exponential time in the worst case. We extend the algorithm to parallel functional languages by introducing new primitives on arrays and show that the complexity of the analysis is polynomial even with parallel execution. The analysis has been implemented in a compiler and tested on several numerical algorithms using arrays. In each algorithm all updates have been made destructive by the compiler, thus achieving the efficiency of imperative languages.

1.6 Overview

Chapter II describes our source and intermediate languages, the flow analysis algorithm, its complexity, and implementation results. An earlier version of this chapter appears in [63]. Like all global analyses we assume that the complete source program is available for analysis. This obviously is a problem with separate compilation. One simple solution is always to copy arguments that cross module boundaries. We describe more effective scenarios for separate compilation in Chapter IV.

Chapter III describes a parallel functional language and an extension of the algorithm for parallel destructive updating. An earlier version of this chapter appears in [62].

Chapter IV describes our compiler, runtime system and preliminary performance results.

Chapter V discusses some problems with non-flat arrays and an extension of the analysis for non-flat but non-recursive arrays. It is a modified version of [61].

Chapter VI discusses related research.

Chapter VII concludes the thesis by describing problems for further research.

CHAPTER II

AN EFFICIENT UPDATE ANALYSIS ALGORITHM

2.1 Introduction

In this chapter we present an update analysis algorithm for first-order strict functional languages with flat aggregates (or arrays of scalars). In contrast to previous work, our analysis does not assume any fixed order of evaluation. The problem with fixing an order *a priori* is that opportunities for destructive updating may be lost. Furthermore we show that the time complexity of our algorithm is polynomial in the worst case, and close to linear in the typical case. No previous work on interprocedural update analysis [12, 27, 29, 37, 42, 45, 66] reports a polynomial time complexity for either case, so we believe ours is the first practical algorithm for this problem.

For cases where the analysis determines that an update cannot be made destructive, we present a simple heuristic to reduce copying.

We have implemented our update analysis algorithm and have run it on several examples. Our results show that for most examples, a good order of evaluation makes all the updates destructive, whereas an analysis that assumes a fixed order detects only the updates that can be optimized with that order.

The rest of the chapter is organized as follows. Section 2.2 describes the source and intermediate languages. Section 2.3 presents an overview of the solution with some notation used in later sections. Section 2.4 describes the abstract domains and the functions used in the analysis. Section 2.5 shows how to derive a good evaluation

order using the information obtained from these abstract functions. The abstract reference count analysis, which uses the order of evaluation derived previously, is described in Section 2.6. In Section 2.7 we show that our analysis algorithm runs in polynomial time. Section 2.8 presents our experimental results. In Section 2.9 we compare our work with Hudak's abstract reference counting. The last section describes a simple heuristic for copy reduction by judicious copy introduction.

2.2 The Source and Intermediate Languages

We consider a first-order, strict functional language (Figure 1) with flat aggregates (*i.e.* an aggregate can only contain non-aggregate values). Our language does not permit local definitions of variables or functions. This is not a serious restriction because it is always possible to eliminate all local definitions by "lambda lifting" [48]. To simplify the analysis we will work with an intermediate language, where each non-trivial subexpression is given a unique name which can be thought of as a compiler-generated temporary variable [3]. As we will see shortly, the analysis will distinguish these temporary variables from other identifiers. The syntax of the intermediate language is given in Figure 2. Notice that the only way a non-trivial expression can appear inside another expression is through a conditional or a let-expression. The scope of a let-binding $t_i = e_i$ in a let-expression `let` $[\ldots, t_i = e_i, \ldots, t_n = e_n]$ `in` t `end` consists of all the occurrences of t_i in expressions e_{i+1} to e_n and t.

The selection operator `sel` takes an aggregate and an index and returns the value stored at that index in the aggregate. The update operator `upd` takes an aggregate a, an index *i*, and a value *v* and returns a new aggregate which contains *v* at the index *i* but is otherwise like a. We assume that for an aggregate of size *n* the indices range from *0* to *n-1*.

A program is a set of mutually recursive function definitions.

$$
\begin{array}{lcll}
c & \in & \mathit{Cons} & \text{Constants} \\
x & \in & V & \text{Variables} \\
op & \in & \mathit{Prims} & \text{Primitive functions } (\mathit{i.e.}\ \texttt{+}, \texttt{-}, \ldots) \\
f & \in & F & \text{Defined functions}
\end{array}
$$

$$
\begin{array}{lcll}
e & \in & \mathit{Exp} & ::= \quad c \mid x \mid op(e_1, \cdots, e_n) \\
 & & & \quad \mid \texttt{sel}(e_1, e_2) \\
 & & & \quad \mid \texttt{upd}(e_1, e_2, e_3) \\
 & & & \quad \mid f(e_1, \cdots, e_n) \\
 & & & \quad \mid \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & & \\
pr & \in & \mathit{Program} & ::= \quad \{f_1\,(x_{1_1}, \cdots, x_{k_1}) = e_1; \\
 & & & \qquad\qquad \vdots \\
 & & & \qquad f_n\,(x_{1_n}, \cdots, x_{k_n}) = e_n\}
\end{array}
$$

Figure 1: The Syntax of the Source Language

$$
\begin{array}{lcll}
t_i & \in & TV & \text{Temporary variables} \\
 & & & \\
se & \in & SE & ::= \quad c \mid x \mid t_i \\
 & & & \\
e & \in & \mathit{IExp} & ::= \quad se \mid op(se_1, \ldots, se_n) \\
 & & & \quad \mid \texttt{sel}(se_1, se_2) \\
 & & & \quad \mid \texttt{upd}(se_1, se_2, se_3) \\
 & & & \quad \mid f(se_1, \ldots, se_n) \\
 & & & \quad \mid \texttt{if } se \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & & \quad \mid \texttt{let } [t_1 = e_1, \ldots, t_n = e_n] \texttt{ in } t_i \texttt{ end} \\
 & & & \\
pr & \in & \mathit{IProg} & ::= \quad \{f_1\,(x_{1_1}, \cdots, x_{k_1}) = e_1; \\
 & & & \qquad\qquad \vdots \\
 & & & \qquad f_n\,(x_{1_n}, \cdots, x_{k_n}) = e_n\}
\end{array}
$$

Figure 2: The Syntax of the Intermediate Language

Consider the problem of adding two vectors a and b of size n. The function `vector_add` takes two arrays `a` and `b` and their size `n` and returns a new array containing the vector sum. It uses a help function `vadd` that adds two vectors from index `i` to `j-1`.

```
vector_add (a,b,n) = vadd(a,b,0,n);
vadd(a,b,i,j) = if (i = j) then a
                else
                  vadd(upd(a,i,sel(a,i)+sel(b,i)),b,(i+1),n);
```

In the above program, the update operation is a non-destructive update. The aim of our analysis is to determine if such an update can be replaced by a destructive update.

2.3 Overview

The key insight that led to our algorithm is that, in a first-order language with flat aggregates, there is no need to track anonymous values because their "reference count" can never exceed 1. Any aggregate for which there are multiple references must be the value of some variable. Hence update analysis reduces to live variable analysis. Our contribution has been (1) to notice this and (2) to find an efficient way to combine liveness analysis with an algorithm for choosing a good order of evaluation.

Assuming a fixed order of evaluation limits the effectiveness of update analysis, as shown by the following example:

```
f (x,i)   = if i=0 then x else  upd(x,i,i);
g (x,i,j) = sel(x,i) - sel(f(x,i),i*j)) - sel(x,2i);
```

No fixed order of evaluation of the operands to the - operator can make the update in `f` destructive. Consider left to right evaluation of the operands. Destructive

updating in `f` would interfere with the evaluation of `sel(x,2i)`. For a right to left evaluation order, destructive updating would interfere with `sel(x,i)`. In general, for a given evaluation order, one can construct examples where destructive updating is not possible. However, in the above example the update can be done destructively by choosing first to evaluate the first term in the body of g, then the third term, and finally the middle term.

The first aim of our analysis is to find an order in which the selection operations on an aggregate precede the updates for that aggregate. As the example above suggests, this requires an interprocedural analysis. An update can then be converted into a destructive update if it updates an aggregate that is no longer live, that is, an aggregate for which there are no further references.

To find a good ordering, we associate with each expression the aggregates that are selected and updated (*selects-and-updates analysis*). To the expression `sel(x,i)`, for example, we will associate the information that the aggregate `x` is selected by that expression. For `sel(t_1,i)`, where `t_1=f(x,y)`, this information depends on which aggregates are returned by the function `f`. In particular we need to know whether `f` returns any of its arguments. The *propagation analysis* therefore collects information regarding which variables are returned or propagated by the evaluation of an expression. For example, the variables propagated by the expression `g(x,y)`, where g simply returns `x`, consist of `x`, provided `x` and `y` are not aliased. Hence aliasing information must be computed. To devise a good order of evaluation we need therefore to collect the following information: propagation of variables, aliasing, and variables selected and updated.

We use flow analysis to collect the above information. We can then create a

data dependency graph for each expression and augment it with additional edges, which we call *interference edges*. An expression e_2 is said to *interfere* with another expression e_1 if e_2 updates an aggregate that is selected or updated by e_1. If e_2 is evaluated before e_1, then the interfering updates in e_2 cannot be made destructive, so we would like to evaluate e_1 before e_2. The augmented data dependency graph is used to derive an order of evaluation for the expressions.

In the following we assume that all variables in a program are distinct. Moreover, given a binding $t_i = e_i$, we assume the existence of a function *expr-of* which, given the temporary variable t_i, returns the corresponding expression, *i.e.* e_i. We assume that the reader is familiar with partial orders, the least upper bound (lub) operation denoted by $\bigsqcup$, and fixpoints. We avoid subscripts for least upper bound operators when they are clear from the context. We use fix as the least fixpoint operator. All the abstract domains are finite and the functions are monotonic, so least fixpoints exist and are computable. For a good introduction to these concepts, the reader may refer to [65, 71].

Environments are finite maps from the syntactic domain of identifiers to some other domain of interest. The empty environment, which is the least element in the domain of environments, is denoted by $\perp$. The value of an identifier x in an environment σ is written as $\sigma[x]$. The environment obtained by extending another environment σ with a binding $x \mapsto v$ is written as $\sigma[x \mapsto v]$. An environment mapping the variables x_1 to $v_1, \ldots, x_n$ to v_n is written as $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. The notation $[f_i \mapsto e_i]$ stands for $[f_1 \mapsto e_1, \ldots, f_n \mapsto e_n]$. The least upper bound operation on the domain of environments can be defined in terms of the least upper bound operation on the range of the environments. If $Env = Ide \rightarrow D$, then the lub of two environments

env_1 and env_2 is defined as

$$env_1 \sqcup_{Env} env_2 = \lambda x \in Ide.\, env_1[x] \sqcup_D env_2[x]$$

2.4 Flow Analysis

We define three detailed functions for flow analysis. Each flow function computes one particular kind of flow information. $\mathcal{H}$ computes the variables propagated by an expression, $\mathcal{A}$ computes the aliasing of formal parameters in a program, and $\mathcal{S}$ computes the sets of aggregates selected and updated by an expression. These three functions are used to define three summarizing functions $\mathcal{H}_p$, $\mathcal{A}_p$, and $\mathcal{S}_p$ on programs.

2.4.1 Propagation Analysis

The abstract domains needed for propagation analysis are shown in Figure 3, where V represents the set of all distinct variables in a program (not including the temporary variables) and D is ordered by the subset relation. The function $\mathcal{H}$ (see Figure 4) takes an expression, a variable and a function environment and returns the set of variables propagated by that expression.

Notice that while the set of variables propagated by an identifier is obtained by looking it up in the variable environment, the set of variables propagated by a temporary variable is obtained by computing the set of variables propagated by its associated expression. To simplify the presentation, we assume the primitive operators of our language do not propagate any of their arguments; this assumption is not needed by the algorithm. As our language does not permit non-flat aggregates, a selection does not propagate any variable. The set of variables propagated by

	V			Program Variables
	F			Defined Functions
	D	$=$	$\mathcal{P}(V)$	Powerset Domain over V
$\sigma \in$	$VEnv$	$=$	$V \to D$	Variable Environments
$\rho \in$	$FEnv$	$=$	$F \to D^n \to D$	Function Environments

Figure 3: Domains for Propagation Analysis

$$\mathcal{H} \; : \; IExp \to VEnv \to FEnv \to D$$

$$\begin{aligned}
&\mathcal{H}[\![\mathrm{c}]\!]\sigma\ \rho &&= \emptyset \\
&\mathcal{H}[\![x]\!]\sigma\ \rho &&= \sigma[x] \\
&\mathcal{H}[\![t_i]\!]\sigma\ \rho &&= \mathcal{H}[\![\text{expr-of}(t_i)]\!]\sigma\ \rho \\
&\mathcal{H}[\![op(se_1, \ldots, se_n)]\!]\sigma\ \rho &&= \emptyset \\
&\mathcal{H}[\![\mathtt{sel}(se_1, se_2)]\!]\sigma\ \rho &&= \emptyset \\
&\mathcal{H}[\![\mathtt{upd}(se_1, se_2, se_3)]\!]\sigma\ \rho &&= \emptyset \\
&\mathcal{H}[\![f_k(se_1, \ldots, se_n)]\!]\sigma\rho = \rho[f_k](\mathcal{H}[\![se_1]\!]\sigma\rho, \ldots, \mathcal{H}[\![se_n]\!]\sigma\ \rho) \\
&\mathcal{H}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\sigma\ \rho &&= \mathcal{H}[\![e_1]\!]\sigma\ \rho \ \cup \ \mathcal{H}[\![e_2]\!]\sigma\ \rho \\
&\mathcal{H}[\![\mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ t_i\ \mathtt{end}]\!]\sigma\ \rho = \mathcal{H}[\![t_i]\!]\sigma\ \rho
\end{aligned}$$

Figure 4: The Function $\mathcal{H}$

$$\mathcal{H}_p : IProg \to FEnv$$

$$\mathcal{H}_p[\![pr]\!] = fix(\lambda\rho.\ \rho[f_i \mapsto \lambda y_1, \ldots, y_k.\mathcal{H}[\![e_i]\!][x_{1_i} \mapsto y_1, \ldots, x_{k_i} \mapsto y_k]\ \rho])$$

Figure 5: The Function $\mathcal{H}_p$

an update is also empty because semantically the update operation returns a new aggregate which is different from any of the aggregates passed to it as arguments. For a function call, the set of variables propagated by each actual parameter is computed recursively. The abstraction of the function, looked up in the function environment, is applied to the abstract values of the actual parameters.

The function $\mathcal{H}$ is used to define $\mathcal{H}_p$ (Figure 5), which takes a program pr and returns an environment in which each user defined function is mapped to an abstract function that gives information about the variables propagated by the body of the function. Notice that we do not build a table to represent the input-output behavior of a function, as shown by the following example:

```
f (x,y,z) = if g(x) then x else f(y,z,x)
```

Its $\mathcal{H}$-meaning is

$$f\ x\ y\ z = x \cup f(y,\ z,\ x)$$

The least fixpoint of the functional associated with the above equation is computed by successive approximations [65]. The sequence of these successive approximations will be:

$$f_0 = \lambda xyz.\emptyset$$
$$f_1 = \lambda xyz.x \cup \emptyset$$
$$f_2 = \lambda xyz.x \cup y$$
$$f_3 = \lambda xyz.x \cup y \cup z$$

where f_3, the fixpoint, conveys the information that all the parameters of f can be propagated.

2.4.2 Aliasing Analysis

We represent the aliasing information as an environment in which each variable is bound to the set of variables, including itself, that it may alias. The domains necessary to compute aliasing information are given in Figure 6. The function $\mathcal{A}$ (see Figure 7) takes an expression, an aliasing environment, and a function environment, and returns a new aliasing environment.

The only kind of expression that can give rise to aliasing is the function call. For a function call, we determine the variables propagated by its actual parameters using the aliasing environment as the variable environment, and the propagation environment as the function environment. For each pair of formal parameters of the function, we determine whether the sets of variables propagated by the corresponding actual parameters are disjoint. If the two sets are disjoint, then no aliasing between those two formal parameters is caused by the particular function call. If the sets are not disjoint, then the two formal parameters of the function can potentially be aliased. Computationally, if the formal parameters x_{i_k} and x_{j_k} could be aliased, then their alias sets must be merged, and not only x_{i_k} and x_{j_k} but all their aliases must be updated with the new set of aliases.

Notice that we treat aliasing as a transitive relation. If one call to a function aliases the first and second arguments, and a different call aliases the first and third arguments, then we assume that all three formal parameters of that function are aliased to one another. This is the main source of imprecision in our analysis. More precise aliasing information can be obtained from a more expensive analysis.

The function $\mathcal{A}_p$ is defined in Figure 8. In this definition, σ_{id} is the identity environment in which every variable is bound to a singleton set containing itself. It

$$
\begin{array}{rrcl}
 & D & = & \mathcal{P}(V) \quad \text{Power set of Variables} \\
\sigma \in & AEnv & = & V \rightarrow D \quad \text{Aliasing Environments}
\end{array}
$$

Figure 6: Domains for Aliasing Analysis

$$\mathcal{A} : IExp \rightarrow AEnv \rightarrow FEnv \rightarrow AEnv$$

$$
\begin{array}{ll}
\mathcal{A}[\![\mathrm{c}]\!]\sigma\ \rho & = \sigma \\
\mathcal{A}[\![x]\!]\sigma\ \rho & = \sigma \\
\mathcal{A}[\![t_i]\!]\sigma\ \rho & = \sigma \\
\mathcal{A}[\![op(se_1, \ldots, se_n)]\!]\sigma\ \rho & = \sigma \\
\mathcal{A}[\![\mathtt{sel}(se_1, se_2)]\!]\sigma\ \rho & = \sigma \\
\mathcal{A}[\![\mathrm{upd}(se_1, se_2, se_3)]\!]\sigma\ \rho & = \sigma \\
\mathcal{A}[\![f_k(se_1, \ldots, se_n)]\!]\sigma\rho & = \\
\quad let & \\
\quad\quad v_i = \mathcal{H}[\![se_i]\!]\sigma\ \rho & \\
\quad\quad a_{ij} = v_i \cap v_j, 1 \leq i, j \leq n, i \neq j & \\
\quad in & \\
\quad\quad \sigma \sqcup [x_{l_k} \mapsto (\sigma[x_{i_k}]\ \cup\ \sigma[x_{j_k}]) \mid a_{ij} \neq \emptyset, x_{l_k} \in (\sigma[x_{i_k}]\ \cup\ \sigma[x_{j_k}])] & \\
\quad end &
\end{array}
$$

$$\mathcal{A}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\sigma\ \rho\ =\ \mathcal{A}[\![e_1]\!]\sigma\ \rho\ \sqcup\ \mathcal{A}[\![e_2]\!]\sigma\ \rho$$

$$\mathcal{A}[\![\mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ t_i\ \mathtt{end}]\!]\sigma\ \rho = \bigsqcup_{i=1}^{n} \mathcal{A}[\![e_i]\!]\sigma\ \rho$$

Figure 7: The Function $\mathcal{A}$

$$\mathcal{A}_p : IProg \rightarrow AEnv$$

$$
\begin{array}{l}
\mathcal{A}_p[\![pr]\!]\ = \\
\quad let\quad \rho = \mathcal{H}_p[\![pr]\!] \\
\quad in \\
\qquad\qquad fix(\lambda\sigma.\ (\bigsqcup_{i=1}^{n} \mathcal{A}[\![e_i]\!]\sigma\ \rho\ \sqcup\ \sigma_{id})) \\
\quad end
\end{array}
$$

Figure 8: The Function $\mathcal{A}_p$

represents the initial program aliasing where every variable is aliased to itself. For a program pr and a variable x, $\mathcal{A}_p[\![pr]\!]\,x$ is the set of all variables x may alias.

2.4.3 Selects-and-Updates Analysis

The domains needed for selects-and-updates analysis are given in Figure 9. The first component of the abstract domain D_{su} represents the set of variables that are possibly selected in the evaluation of the expression. The second component gives the set of variables possibly updated by the expression evaluation. The domain $SEnv$ represents the abstraction of each user defined function to a function that returns the set of variables selected and updated by the function, given the set of aggregates bound to each of its arguments. The functions $\mathcal{S}$ and $\mathcal{S}_p$ are defined in Figure 10 and Figure 11, respectively.

2.5 Deriving an Order of Evaluation

Our objective is to choose an order of evaluation for the bindings of a let-expression in which the selection operations on each aggregate precede update operations on that aggregate. Given a let expression $\texttt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \texttt{in}\ t_i\ \texttt{end}$, we first construct a data dependency graph whose nodes are the expressions $t_1, \ldots, t_n$. We say an expression t_j depends on t_i, if t_i appears in the free variables of t_j. In the dependency graph, this dependence is represented as a directed edge (i, j), indicating that the node i must be evaluated before the node j. The dependency graph will necessarily represent a partial order (*i.e.* a directed acyclic graph, or dag) because we are assuming strict semantics and our language doesn't allow cyclic data structures.

We then augment the dependency graph with additional edges which we call *interference edges*. An interference edge (i, j) conveys the information that e_j possibly

$$
\begin{array}{rrcll}
 & D_{su} & = & \mathcal{P}(V) \times \mathcal{P}(V) & \text{Selected and Updated Variables} \\
\sigma \in & VEnv & = & V \to \mathcal{P}(V) & \text{Variable Enviroments} \\
\rho \in & FEnv & = & F \to \mathcal{P}(V)^n \to \mathcal{P}(V) & \text{Propagation Enviroments} \\
\delta \in & SEnv & = & F \to \mathcal{P}(V)^n \to D_{su} & \text{Selects-and-Updates Function Environments}
\end{array}
$$

Figure 9: Domains for Selects-and-Updates Analysis

$$\mathcal{S} \; : \; IExp \to SEnv \to FEnv \to VEnv \to D_{su}$$

$$
\begin{array}{lcl}
\mathcal{S}[\![\mathtt{c}]\!]\delta\,\rho\,\sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![x]\!]\delta\,\rho\,\sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![op(se_1, \ldots, se_n)]\!]\delta\,\rho\,\sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![\mathtt{sel}(se_1,se_2)]\!]\delta\,\rho\,\sigma & = & \langle \mathcal{H}[\![se_1]\!]\sigma\,\rho, \emptyset \rangle \\
\mathcal{S}[\![\mathtt{upd}(se_1,se_2,\, se_3)]\!]\delta\,\rho\,\sigma & = & \langle \emptyset, \mathcal{H}[\![se_1]\!]\sigma\,\rho \rangle \\
\mathcal{S}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\delta\,\rho\,\sigma & = & \mathcal{S}[\![e_1]\!]\delta\,\rho\,\sigma \;\sqcup\; \mathcal{S}[\![e_2]\!]\delta\,\rho\,\sigma \\
\mathcal{S}[\![f_k(se_1, \ldots, se_n)]\!]\delta\,\rho\,\sigma & = & \delta[f_k](\mathcal{H}[\![se_1]\!]\sigma\,\rho, \ldots, \mathcal{H}[\![se_n]\!]\sigma\,\rho)
\end{array}
$$

$$\mathcal{S}[\![\mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ t_i\ \mathtt{end}]\!]\delta\,\rho\,\sigma = \bigsqcup_{i=1}^{n} \mathcal{S}[\![e_i]\!]\delta\,\rho\,\sigma$$

Figure 10: The Function $\mathcal{S}$

$$\mathcal{S}_p : IProg \to SEnv$$

$$\mathcal{S}_p[\![pr]\!] = fix(\lambda\delta.\ \delta[f_i \mapsto \lambda y_1, \ldots, y_k.\mathcal{S}[\![e_i]\!]\ \delta\ (\mathcal{H}_p[\![pr]\!])\ [x_{1_i} \mapsto y_1, \ldots, x_{k_i} \mapsto y_k]])$$

Figure 11: The Function $\mathcal{S}_p$

updates an aggregate needed by e_i. (The predicate *Interferes* is defined formally in Figure 12.) We call the graph so obtained a *precedence graph*.

$$
\begin{array}{l}
interferes\ e_i\ e_j = \\
\quad let \\
\qquad fenv = \mathcal{H}_p[\![pr]\!] \\
\qquad aenv = \mathcal{A}_p[\![pr]\!] \\
\qquad suenv = \mathcal{S}_p[\![pr]\!] \\
\qquad < s_i, u_i >= \mathcal{S}[\![e_i]\!] suenv\ fenv\ aenv \\
\qquad < s_j, u_j >= \mathcal{S}[\![e_j]\!] suenv\ fenv\ aenv \\
\quad in \\
\qquad u_j \cap (s_i \cup u_i) \neq \emptyset \\
\quad end
\end{array}
$$

Figure 12: The Predicate *Interferes*

The precedence graph represents a preorder but not necessarily a partial order (dag), so we take its quotient under the induced equivalence relation. That is, we find the strongly connected components of the precedence graph using the algorithm given in [2]. We construct a new graph whose nodes are the strongly connected components of the precedence graph. There is an edge E_{ij} between the nodes V_i and V_j if $\exists v_i \in V_i$ and $v_j \in V_j$ such that (i, j) is an edge of the precedence graph. The new graph is necessarily a partial order. A topological sort of the partial order gives a total order for the evaluation of the strongly connected components of the precedence graph. A total ordering of all the expressions is then obtained by replacing each component by any ordering of its elements that is consistent with the data dependencies.

The complexity of choosing an order for a let-expression containing n bindings is $O(n^2)$. Every topological sort of the dependency graph is safe, so a faster algorithm

could be obtained at the risk of choosing a less desirable ordering.

In the final ordered let-expression, `let` $[t_i = e_i, \ldots, t_k = e_k]$ `in` t `end`, the expression t_i is evaluated before t_k. Given the order of expression evaluation, we can then determine the set of live variables at each binding as explained in the next section.

2.6 Abstract Reference Count Analysis

Our abstraction of reference counts is a 2-point domain R whose least element 1 represents the existence of exactly one reference to an aggregate, and whose top element $\top$ represents multiple references. Intuitively, a variable is *live* at a program point if there are any future references to it. Depending on where the reference occurs we will distinguish between *local* and *global* liveness. Consider the following example:

```
f (x,i)  =  let t_1 = g(x,i);
                t_2 = sel(t_1,i);
                t_3 = sel(x,i);
                t_4 = t_2 + t_3;
            in
              t_4
            end;

g (y,j)  =  let  t = upd(y,i,i)
            in
               t
            end;
```

In the body of `f`, we will say that `x` is locally live at point `t_1` because there exists another reference to `x`, namely at point `t_3`. In the body of g, `y` is not locally live because there are no further references to `y` in g after `t`. However, `y` is globally live because there exists a call to g (*i.e.* `g(x,i)`) with a live actual parameter. The global liveness is computed by $\mathcal{R}$, which returns the abstract reference counts of objects bound to the formal parameters of a function in all possible calls that could arise in

any program execution.

The abstract domains and the function $\mathcal{R}$ used to compute the abstract reference environment are shown in Figures 13 and 14, respectively.

In the definition of $\mathcal{R}$ we make use of the functions $FV(e_j)$ and $Vars(e_j)$. The function $FV(e_j)$ returns the set of free variables in the expression e_j, and $Vars(e_j)$ is $FV(e_j) \setminus \{t_1, \ldots, t_n\}$.

The definition of $\mathcal{R}$ needs explanation only for the function call and the let-expression. Given a function call, we determine the set of variables propagated by each actual parameter of the function using $\mathcal{H}$. The liveness of each parameter is tested by checking whether it is globally or locally live. An actual parameter is globally live if at least one of the variables (or its aliases) propagated by the actual parameter has the value $\top$ in the reference environment *renv*. An actual parameter is locally live if at least one of the variables (or its aliases) propagated by the actual parameter is in the live variable set *lset*, which is given as an argument to $\mathcal{R}$.

In the case of a let-expression, the set of live variables at each binding is computed and the bindings are analyzed recursively. The live variables at a binding are the variables, and their aliases, that appear in the expressions yet to be evaluated; the variables propagated by already evaluated expressions that are used in some expression that has yet to be evaluated; and the set of variables live (*lset*) after the evaluation of the whole let-expression.

The abstract reference environment for an entire program *pr* is computed by the function $\mathcal{R}_p$, which is defined in Figure 15.

Given $\mathcal{R}_p[\![pr]\!]$, it is easy to decide whether an update can be performed destructively. Consider an update expression $t_i = \mathrm{upd}(x, y, z)$. Suppose $lset_i$ is the set of

$$
\begin{array}{lcll}
FEnv & = & F \to \mathcal{P}(V)^n \to \mathcal{P}(V) & \\
AEnv & = & V \to \mathcal{P}(V) & \\
R & = & \{1, \top\} & \text{Reference Counts} \\
REnv & = & V \to R & \text{Reference Count Environments} \\
LSet & = & \mathcal{P}(V) & \text{Live Variables}
\end{array}
$$

Figure 13: Domains for Reference Count Analysis

$$\mathcal{R} \; : \; IExp \to REnv \to FEnv \to AEnv \to LSet \to REnv$$

$$
\begin{array}{l}
\mathcal{R}[\![\mathtt{c}]\!]\ renv\ fenv\ aenv\ lset = renv \\
\mathcal{R}[\![x]\!]\ renv\ fenv\ aenv\ lset = renv \\
\mathcal{R}[\![t_i]\!]\ renv\ fenv\ aenv\ lset = renv \\
\mathcal{R}[\![op(se_1, \ldots, se_n)]\!]\ renv\ fenv\ aenv\ lset = renv \\
\mathcal{R}[\![\mathtt{sel}(se_1,se_2)]\!]\ renv\ fenv\ aenv\ lset = renv \\
\mathcal{R}[\![\mathtt{upd}(se_1,se_2,se_3)]\!]\ renv\ fenv\ aenv\ lset = renv \\
\mathcal{R}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\ renv\ fenv\ aenv\ lset = \\
\quad \mathcal{R}[\![e_1]\!]\ renv\ fenv\ aenv\ lset \sqcup \mathcal{R}[\![e_2]\!]\ renv\ fenv\ aenv\ lset \\
\mathcal{R}[\![f_k(se_1,\ldots,se_n)]\!]\ renv\ fenv\ aenv\ lset = \\
\quad let\ \ v_1 = \mathcal{H}[\![se_1]\!] aenv\ fenv \\
\qquad \vdots \\
\qquad v_n = \mathcal{H}[\![se_n]\!] aenv\ fenv \\
\quad in \\
\qquad renv \sqcup [x_{j_k} \mapsto \top \mid \exists x \in v_j, renv[x] = \top \ \vee x \in lset] \\
\quad end \\
\mathcal{R}[\![\mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ t_l\ \mathtt{end}]\!]\ renv\ fenv\ aenv\ lset = \\
\ let \\
\ lset_i = (\cup\{aenv[x] \mid x \in \bigcup_{j=i+1}^{n} Vars(e_j)\}) \cup \\
\qquad (\cup\{\mathcal{H}[\![t_k]\!]\ aenv\ fenv \mid k<i, \exists j\, i<j\leq n, t_k \in FV(t_j)\}) \\
\qquad \cup\ lset \\
\ in \\
\quad \bigsqcup_{i=1}^{n} \mathcal{R}[\![e_i]\!]\ renv\ fenv\ aenv\ lset_i \\
\ end
\end{array}
$$

Figure 14: The Function $\mathcal{R}$

$$
\begin{array}{lll}
\mathcal{R}_p & : & IProg \rightarrow REnv \\
\mathcal{R}_p[\![pr]\!] & = & \\
 & & \quad let \quad fenv = \mathcal{H}_p[\![pr]\!] \\
 & & \qquad\quad aenv = \mathcal{A}_p[\![pr]\!] \\
 & & \quad in \\
 & & \qquad fix(\lambda\sigma.\ \bigsqcup_{i=1}^{n} \mathcal{R}[\![e_i]\!]\ \sigma\ fenv\ aenv\ \emptyset) \\
 & & \quad end
\end{array}
$$

Figure 15: The Function $\mathcal{R}_p$

variables that are live at point t_i. This update *cannot* be made destructive if there is at least one variable propagated by x which is either globally or locally live. This condition can be expressed formally as follows:

$$\exists y \in (\mathcal{H}[\![x]\!]\ \mathcal{A}_p[\![pr]\!]\ \mathcal{H}_p[\![pr]\!]) \text{ such that } \mathcal{R}_p[\![pr]\!][y] = \top \quad \text{or} \quad y \in lset_i$$

If this condition does not hold, then it is safe to perform the update destructively.

Notice that we can use the same analysis to introduce explicit deallocation instructions. That is, we can safely deallocate an array if it is no longer live.

2.7 Complexity of the Analysis

In this section we bound the cost of computing the functions $\mathcal{H}_p$, $\mathcal{A}_p$, $\mathcal{S}_p$ and $\mathcal{R}_p$. Each of these functions involves a fixpoint calculation. The complexity is estimated by bounding both the number of iterations needed for the fixpoint computation and the complexity of each iteration. The program size is represented in terms of three parameters: n, the number of functions in the program; k, the maximum number of arguments of aggregate type to any one function, which is bounded by the maximum

function arity; and m, the number of non-let-expressions in the intermediate program (*i.e.* the size of the original program), where $m \geq n$. The number of functions and the maximum function arity are used to bound the number of iterations needed in a fixpoint computation. The number of non-let-expressions in the program, together with the function arity, is used to estimate the complexity of each iteration. The basic unit of analysis is a non-let-expression. The complexity of an iteration is determined by multiplying the complexity of analyzing a non-let-expression by the number of non-let-expressions in the program.

$\mathcal{H}_p$ computes an element of $FEnv$ in which each function symbol is mapped to an abstract propagation function, which can only be a union of some of its formal parameters. The maximum number of iterations needed to compute the fixpoint is proportional to the height of the domain of $FEnv$ which is kn. Analyzing a function call is the most expensive operation, involving union of at most k sets of at most size k. The complexity of the set union is $O(k^2)$. The complexity of a single iteration is $O(mk^2)$, so the complexity of $\mathcal{H}_p$ is $O(mnk^3)$.

Each variable can be aliased to at most k variables as we only consider aliasing among the formal parameters of a function. Therefore the number of iterations needed to compute the aliasing environment is the height of the domain of aliasing environments which is $O(nk^2)$. Analyzing a function call, the most expensive operation, requires $O(k^2)$ set intersections of at most size k. Updating an aliasing environment also takes $O(k^2)$ time. Each set intersection has a complexity $O(k)$. Therefore the complexity of a single iteration is $O(mk^3)$, which gives an overall complexity of $O(mnk^5)$ for the function $\mathcal{A}_p$.

$\mathcal{S}_p$ computes an element of $SEnv$ where each function symbol is mapped to

an abstract selects-and-updates function. Any selects-and-updates function is a pair whose components are unions of some arguments to the functions. If the maximum arity of a function is k, then the height of this domain is $2k$. Thus the number of iterations needed for the fixpoint computation is $O(nk)$. Analyzing a call requires computing a pair of k unions of sets of size k. Complexity of a single iteration is $O(mk^2)$, so the complexity of $\mathcal{S}_p$ is $O(mnk^3)$.

The number of variables in the reference environment is bounded by nk. Each variable takes values from the 2-point domain R. The number of iterations in the fixpoint computation is $O(nk)$ because each variable starts with a value 1 and at least one variable becomes $\top$ in each iteration. Complexity of computing liveness of an actual parameter is $O(k)$; this gives a bound of $O(mk^2)$ on the time for each iteration. The complexity of $\mathcal{R}_p$ is $O(mnk^3)$.

The cost of choosing orders of evaluation is $O(m^2)$. If the source code for all functions were of a fixed constant size c, independent of program size, then $c = m/n$ and the complexity would be $O(nc^2) = O(m^2/n) = O(m)$. Since larger programs can have larger functions, the typical cost is slightly worse than linear in m.

The entire algorithm therefore has a worst-case complexity of $O(\max(m^2, mnk^5))$. The worst case for the mnk^5 term can be achieved only when the call graph is strongly connected and all variables are aliased to all variables. The high degree of k does not seem to matter in practice because aliasing is not the usual case, and most functions take only a few aggregates as arguments so the "effective value" of k is quite small. Two iterations ordinarily suffice for each fixpoint, instead of the worst case's nk or nk^2. Another factor of k ordinarily disappears because k is less than the number of bits in a machine word, allowing most set unions and intersections to be computed in

constant time. The typical cost of the flow analysis is therefore less than proportional to mk^2, which is to say it is practically linear in the size of the program.

In our experience the dominant cost has been that of choosing orders of evaluation, so the algorithm's typical cost appears to be a little worse than linear but much better than quadratic in the size of the program.

2.8 Results

We have implemented the above algorithm in Standard ML and tested it on several programs, representing two-dimensional arrays as one-dimensional arrays. The programs include gaussian elimination, matrix transpose, matrix multiplication, LU-decomposition, quicksort, bubble sort, counting sort (where the range of numbers is known), array initialization, and three artificial programs c_1, c_2, and a 1025-line program derived from gaussian elimination much as Takr was derived from Tak in the Gabriel benchmark suite for Common Lisp [32].

Our implementation analyzed the 1025-line program in 6.5 seconds on a SPARC-station IPC, finding that all updates could be done destructively. Each program listed in Table 2 was analyzed in less than half a second; the largest of these, gaussian elimination, is 67 lines. Our implementation took 0.16 seconds to analyze a 38-line LU-decomposition program, in contrast to Bloss's report that a 40-line LU-decomposition brought the algorithm of [12] "to a near halt" on a Macintosh II. We could easily increase the speed of our implementation if there were reason to do so.

Table 2 shows the number of updates that are converted into destructive updates under various ordering strategies. The first column of the table shows the total number of update operators in the program. Our results are shown in the column headed "no ordering". The next two columns show the results of the analysis with a

Table 2: Effectiveness of the Update Analysis Algorithm

Program	*updates*	*destructive updates*		
		no ordering	*ltr*	*rtl*
gauss-elm-1	5	5	4	5
gauss-elm-2	5	5	5	4
transpose	2	2	1	2
matmul	3	3	2	3
LU-decomp	2	2	2	2
recursive-fft	4	4	4	4
qsort	4	4	4	4
bubblesort	2	2	1	2
count-sort	4	4	3	4
init	1	1	1	1
c_1	2	2	0	1
c_2	1	0	0	0

fixed left-to-right or right-to-left ordering, respectively.

The three programs matmul, transpose and bubblesort use the function *swap* which is defined as follows:

```
swap (a,i,j)  = upd(upd(a,i, sel(a,j)),j,sel(a,i))
```

To make these updates destructive, the operands of the update operator have to be evaluated from right to left. Our analysis derives this order whereas Bloss [12] simply assumes it. The order of evaluation is similarly important for destructive updating in the gaussian elimination program. We wrote the same function with different orderings of the formal parameters and the analyzer finds an appropriate ordering in each case. These two different versions correspond to the two entries gauss-elm-1 and gauss-elm-2 in the table. For the programs quicksort, init, and recursive-fft, any evaluation order is good for destructive updating. Our analysis is also able to find an ordering that interleaves the evaluation of arguments of two

different function calls, as shown in the following program c_1:

```
f (a,j,k) = sel(a,j+k);
g (x,y,i) = f(upd(y,i,i),sel(x,i),sel(y,2*i)) +
            f(upd(x,i,i),sel(y,i),sel(x,2*i))
```

In this example, both updates can be performed destructively only if all the selections are evaluated before the updates. For the following program c_2:

```
f (x,i)   = if i = 0 then x
            else  upd(x,i,2i);
h (y,i,j) =  sel(f(y,i),j) + sel(f(y,j),i);
```

there is no ordering that makes the update destructive. Our analysis safely concludes that the update cannot be made destructive.

Having the compiler derive a good order of evaluation relieves the programmer of trying to come up with the most efficient ordering for the arguments to each function. The effectiveness of the update analysis makes it practical for the compiler to issue a warning whenever it cannot make an update destructive.

2.9 Comparison with Hudak's Work

Hudak described an abstraction of reference counting for update analysis in [45]. In this section we show that, in the absence of aliasing, our analysis is more precise. We first summarize Hudak's approach and its sources of imprecision and discuss how we avoid these imprecisions.

Hudak defines an abstract store semantics of a first-order strict language in which the reference count operation is modeled as a side-effect. This is the reason he fixes an order of evaluation *a priori*, which is one source of imprecision. The abstract store represents the abstract reference counts of each object. When a function is called, the reference count of its actual parameter is incremented by the total number

of occurrences of the corresponding formal parameter in the body of the function. The reference count is decremented when the variable is encountered in the body of the function, mimicking the actual execution. With a finite domain of sticky reference counts with a maximum reference count r_{max}, the increment and decrement operations become imprecise when the reference count exceeds r_{max}.

We show that our analysis is more precise by considering a program with an update that is converted into a destructive update by our analysis but not by Hudak's. Consider the following program

```
f (x,i,j) = g(x) + sel(upd(x,i,i),j);
g y = e;
```

Suppose the body of g has r_{max} occurrences of y with no updates of y. When `g(x)` is called from `f`, `x` has a reference count of 2. By the initialization of Hudak's approach, the reference count of `x` becomes $(2 - 1 + r_{max}) = \infty$ just before the execution of the body of g. If a variable gets a reference count of ∞ it stays there. Therefore, the reference count at the update is ∞ indicating that the update cannot be made destructive.

Our analysis would correctly determine that the update can be made destructive because by our abstraction, `g(x)` does not propagate `x`. This means that all references to `x` created by the call will have been consumed by the time g returns. The update has the last reference to `x`, so it can be made destructive.

Now we show that for programs with no aliasing, our approach is as precise as Hudak's in the event that the order of evaluation chosen by our algorithm coincides with Hudak's. Suppose our approach marks an update as non-destructive. There are two possibilities. The first is that one of the variables propagated by the first

argument of the update operator is locally live. Then there is at least one occurrence of that variable (or an alias) in the rest of the body of the function. It follows that the reference count of the object denoted by that variable must be at least 2 just before the update, so Hudak's method would conclude that the update cannot be made destructive.

The second possibility is that one of the variables propagated by the first argument of the update operator is globally live. Then there is a function call in which the object denoted by the variable is locally live in some function f which eventually calls the function performing the update. This implies that the object denoted by the variable has a reference count of at least 2 (one for the occurrence of the variable in the body of f that caused it to be locally live and the other for the reference held by the update operator). In this case also, Hudak's analysis would conclude that the update cannot be made destructive.

Another source of imprecision in Hudak's approach results from the association of objects to expressions generating them. Two different objects generated by different dynamic instances of a single static update are assumed to be the same. This imprecision can be reduced by a better approximation of the domain of abstract locations. But the better approximations would increase the height of the domain of abstract locations. In our approach we represent the objects by program variables thus avoiding the problem.

As noted in Section 4.2, our aliasing analysis may be imprecise, so Hudak's analysis may do better than ours in the presence of aliasing. It is also possible that, in cases where there is no order of evaluation that avoids copying altogether, our algorithm may choose an ordering that happens to involve copying a larger aggregate,

or more frequent copying of a smaller aggregate. We believe our analysis would dominate Hudak's if we used a more expensive, but still polynomial-time, aliasing analysis, and if we replaced our algorithm for choosing a good evaluation order by Hudak's fixed ordering.

No complexity results are cited in [45], but the complexity of Hudak's abstract reference counting appears to be

$$\Omega(2^{m^2 k})$$

where m is the number of static occurrences of an update operation and k is the number of bound variables in the program under analysis.

2.10 Copy Reduction by Judicious Copying

Can we reduce copying in those cases where the analysis determines that an update cannot be performed destructively? We use the simple heuristic that if possible, an update appearing in the body of a recursively defined function should be made destructive by explicitly copying arguments that are passed from some other function. Consider the following example:

```
rev  (a,n)   = rev1(a,a,0,n);
rev1 (a,b,i,n) = if i = n then a
                 else
                    rev1(upd(a,i,sel(b,n-1-i)),b,i+1, n);
```

In this example, any update analysis algorithm has to conclude that the update must be non-destructive because of the aliasing of `a` and `b`. This algorithm therefore has a cost of $O(n^2)$ in both time and space.

We know, however, that the update could have been made destructive had `a` not been aliased to `b` by the call from `rev`. We would like to determine whether the

update in `rev1` could be made destructive if `rev1` were called appropriately (*i.e.* with no aliasing and no live actual parameters). This can be done by analyzing `rev1` with a reference environment that maps every variable to 1 and by ignoring any aliasing caused by calls to `rev1` that do not arise from the body of `rev1`. In terms of the call graph of the program, we reanalyze `rev1` ignoring the effects of those functions that do not belong to the strongly connected component of `rev1`. We then introduce explicit copying for those arguments of a call to `rev1` that are live or can cause aliasing and can potentially be updated in `rev1`. The transformed program is:

```
rev  (a,n)   = rev1(copy(a),a,0,n);
rev1 (a,b,i,n) = if i = n then a
                 else
                    rev1(upd(a,i,sel(b,n-1-i)),b,i+1, n);
```

By introducing the copy operation, our heuristic has reduced the time and space complexity of this program to $O(n)$. On the other hand, our heuristic may sometimes introduce copying in order to make a seldom-executed update destructive, as would happen if the second argument to *rev* were normally zero.

This heuristic is also relevant to separate compilation, where aggregates that cross the boundary of a compilation unit would otherwise have to be excluded from update analysis. Update analysis with separation compilation is further discussed in Chapter IV.

CHAPTER III

PARALLEL DESTRUCTIVE UPDATING

3.1 Introduction

Although pure functional programming languages show great promise for parallel programming, their success is limited by two problems. One is the array update problem: modification of an array at an index, also called *incremental update* [44], in general requires a new copy of the entire array. The second problem is: how to express parallel updates on an array? Can parallel updates on distinct indices be performed destructively? Can parallel updates on non-distinct indices be performed destructively?

Specifying a collection of updates using the incremental update operator results in a sequential solution. Monolithic arrays [6, 44] were devised to express parallelism at the expense of creation of a new array. Consider the operation of multiplying a row of a matrix by a scalar. With monolithic arrays, a new copy of the entire matrix is required. With our update analysis, the incremental updates yield a sequential solution with no space overhead. Ideally, we would like to update the matrix in parallel without copying the matrix.

In this chapter we show that incremental updates can be used to specify parallel updates. We also present an extension of the algorithm of Chapter II, which we believe to be the first practical algorithm for interprocedural update analysis in first-order functional languages with flat arrays and parallel evaluation.

To handle parallel updates on indices that are not known to be distinct, we devise a new incremental update operator called an *accumulating update* and show that problems like histogram, polynomial multiplication and inverse permutation [6, 44, 75] can be expressed naturally and implemented efficiently using update analysis.

In Section 3.2 we describe a parallel functional language with new operations on arrays. In Section 3.3 we present an extension of the algorithm described in Chapter II and discuss the complexity of our analysis. In Section 3.4 we present the results of the analysis on several programs written in our language. In Section 3.5 we discuss the issue of predictability of the analysis and advocate a programming style for which the analysis would be very effective. In Section 3.6 we describe a new operation to express a collection of updates on an array.

3.2 A Parallel Functional Language

The incremental update operator does not lend itself well for expressing parallel updates on a single array. Consider updating an array `a` at indices `i` and `j` with values 3 and 4. If these two updates are performed in parallel, we get two new arrays each containing only one update. These updates must be non-destructive because the first argument of each update operator is live when the update is performed. Moreover, it is not clear how to incorporate both updates in a single array subsequently. The only way to express these two updates is to choose a sequential order of updates, for example `upd(upd(a,i,3),j,4)`. This criticism of incremental update operator has already been made in [6]. In this chapter we show that by defining new operations on arrays, one can express parallel updates using the incremental update operator and update analysis can determine whether these updates can be made destructively.

Our language is a first-order functional language with flat multi-dimensional

arrays. We introduce an implicitly parallel `let` expression `let` $[t_1 = e_1, \ldots, t_n = e_n]$ `in` e `end`. The scope of a `let` binding $t_i = e_i$ is the entire `let` expression except itself or any region shadowed by a nested `let` binding. We also assume that all t_i's are distinct. Bindings with cyclic dependencies are not permitted. The `let` bindings and the body of the `let` expression can be evaluated in parallel subject to dependency constraints. The source language is described in Figure 16. We introduce a new operation called `partition` which returns multiple values. This construct can only appear on the right hand side of a `let` binding. In the intermediate language (see Figure 17), a partition operator is split into two operators `left_part` and `right_part`. The intermediate language does not have multiple values. There are no nested `let` expressions in the intermediate language.

3.2.1 Partition and Combine operations

A multi-dimensional array $a : [l_1 : b_1, \ldots, l_d : b_d]$ has dimension d and for an index $[i_1, \ldots, i_d]$ to be valid, it must be the case that $l_1 \leq i_1 < b_1, \ldots, l_d \leq i_d < b_d$.

Intuitively, partitioning an array means dividing an array into two subarrays with disjoint index spaces whose union is the index space of the original array. Combining, the inverse operation, is the concatenation of two arrays. A more precise description follows.

Semantics

The operator `partition` takes an array $a : [l_1 : b_1, \ldots, l_d : b_d]$, a dimension k such that $1 \leq k \leq d$, and a partitioning index i such that $l_k < i < b_k$, and returns two new arrays $a_1 : [l_1 : b_1, \ldots, l_k : i, \ldots, l_d : b_d]$ and $a_2 : [l_1 : b_1, \ldots, i : b_k, \ldots, l_d : b_d]$. The value of a_1 or a_2 at index $[i_1, \ldots, i_d]$ is the corresponding value of the array a.

$$
\begin{array}{lllll}
c & \in & \mathit{Cons} & & \text{Constants} \\
x & \in & V & & \text{Variables} \\
op & \in & \mathit{Prims} & & \text{Primitive Operators} \\
 & & & & (\textit{i.e. } \texttt{+,-,sel,upd,partition,combine,array,...}) \\
f & \in & F & & \text{User Defined Functions} \\
 & & & & \\
e & \in & \mathit{Exp} & ::= & c \mid x \mid op(e_1, \cdots, e_n) \\
 & & & & \mid f(e_1, \cdots, e_n) \\
 & & & & \mid \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & & & \mid \texttt{let } \mathit{binds} \texttt{ in } \mathit{exp} \texttt{ end} \\
 & & & & \\
\mathit{binds} & \in & \mathit{Bindings} & ::= & (Ids = exp;)^* \\
Ids & \in & \mathit{Identifiers} & ::= & x | x, x \\
pr & \in & \mathit{Program} & ::= & \{f_1\ (x_{1_1}, \cdots, x_{k_1}) = e_1; \\
 & & & & \qquad \vdots \\
 & & & & f_n\ (x_{1_n}, \cdots, x_{k_n}) = e_n\}
\end{array}
$$

Figure 16: A Parallel Functional Language

$$
\begin{array}{lllll}
op & \in & & & \{\texttt{sel}, \texttt{upd}, \texttt{left\_part}, \texttt{right\_part}, \texttt{+}, \texttt{array}, \texttt{combine}, \ldots\} \\
t_i & \in & TV & & \text{Temporary variables} \\
 & & & & \\
se & \in & SE & ::= & c \mid x \mid t_i \\
 & & & & \\
e & \in & IExp & ::= & se \mid op(se_1, \ldots, se_n) \\
 & & & & \mid f(se_1, \ldots, se_n) \\
 & & & & \mid \texttt{if } se \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & & & \mid \texttt{let } [t_1 = e_1, \ldots, t_n = e_n] \texttt{ in } t_i \texttt{end} \\
 & & & & \\
pr & \in & IProg & ::= & \{f_1\ (x_{1_1}, \cdots, x_{m_1}) = e_1; \\
 & & & & \qquad \vdots \\
 & & & & f_n\ (x_{1_n}, \cdots, x_{m_n}) = e_n\}
\end{array}
$$

Figure 17: The Parallel Intermediate Language

By this definition, indices of a_1 and a_2 are disjoint.

The `combine` operator is the inverse of `partition`. It takes two arrays $a_1 : [l_1 : b_1, \ldots, l_k : b_k, \ldots, l_d : b_d]$ and $a_2 : [h_1 : u_1, \ldots, h_k : u_k, \ldots, h_d : u_d]$ that are compatible for combining, and a dimension k such that $1 \leq k \leq d$, and returns a new array

$$a : [l_1 : b_1, \ldots, l_k : (b_k + u_k - h_k), \ldots, l_d : b_d].$$

Two arrays are compatible for combination if their corresponding sizes in all the dimensions except the one along which they are being combined, are equal. a_1 and a_2 are compatible for combining along dimension k if

$$b_i - l_i = u_i - h_i,\ 1 \leq i \leq d, i \neq k$$

The value of a at index $[i_1, \ldots, i_d]$ is $a_1[i_1, \ldots, i_d]$ if $[i_1, \ldots, i_d]$ lies in the index range of a_1. Otherwise, it is an element of a_2 given by

$$a_2[h_1 + (i_1 - l_1), \ldots, h_k + (i_k - b_k), \ldots, h_d + (i_d - l_d))].$$

3.2.2 An Example

Consider the problem of adding two vectors. In an imperative language, vector addition can be performed by a simple `do` or `for` loop in $O(n)$ time. In a functional language without update analysis, the corresponding loop takes $O(n^2)$ time.

The `partition` and `combine` operators can be used to write more efficient functional programs. The following program runs in $O(n \log n)$ time without update analysis, in $O(n)$ time with update analysis, and in $O(\log n)$ time with update analysis

and parallel execution.

```
\* // is integer division. *\
\* dim(a,i) = no. of elements in dimension i. *\
\* help_vector_add(a,b,i) adds vectors *\
\* a and b from index i onwards. *\

vector_add (a,b) = help_vector_add(a,b,0);

help_vector_add (a,b,i) =
  if dim(a,1) = 1
    then upd(a,i,a[i]+b[i])
  else let midpoint = i + dim(a,1)//2;
           a_1, a_2 = partition (a, midpoint, 1);
           rv_1 = help_vector_add(a_1,b,i);
           rv_2 = help_vector_add(a_2,b,midpoint)
       in
           combine(rv_1,rv_2,1)
       end
  endif;
```

If the size of a is larger than 1, then a is partitioned into two vectors a_1 and a_2. The problem is solved recursively on each vector and the solutions are combined using the `combine` operator.

3.2.3 Implementation Choices

There are two choices of implementation for the `partition` and `combine` operators. In a copying implementation, the two partitions are created by copying data from the original array. In a sharing implementation, the new arrays share data with the original array. This can be achieved by creating new array headers with the new ranges for the two partitions while sharing the data with the original array. The overhead of creating a header is $O(d)$, proportional to the dimension of the array which is usually a small number.

The `combine` operator can also be implemented by copying. However, if arrays that are combined are adjacent partitions implemented by sharing, then `combine`

can also be implemented by sharing. In such a case, it is required to create a new header for the resulting array. The `combine` operator cannot always be implemented by sharing even if `partition` is implemented by sharing. The reason is that one may be combining arrays that are not physically adjacent to one another.

Suppose we have established that all `partition` and `combine` operators in a program can be implemented by sharing. Can we also avoid creation of array headers thus making `partition` an identity operator and `combine` a synchronizing operator? The array headers are used for bounds checking and operations like determining the size of the array. If bounds checking is not performed and the programmer does not use any operation that needs the array header, a situation similar to programming in a language like C, then `partition` and `combine` become operators that return their first argument. The partition operation can even be performed at compile time.

3.3 Update Analysis

One of the key insights that led to our simple algorithm was that anonymous aggregates (new aggregates) need not be tracked. We defined four analyses called *propagation analysis*, *aliasing analysis*, *selects-updates analysis*, and *reference count analysis*. We have to define the flow equations for `partition` and `combine` operators for these analyses. We assume copying semantics for these operators. Since we have extended the source language with `let` expressions, we have to extend the analysis for them.

Since `partition` and `combine` create new aggregates, they do not propagate any of their arguments. Including the `let` expression in the source language allows the user to name an expression and use the name elsewhere one or more times, causing sharing. Consider a `let` binding $t_i = e_i$. If e_i returns an anonymous aggregate, it

V	=	Program Variables	
F	=	User Defined Functions	
D	=	$\mathcal{P}(V \cup \{b\})$	Abstract Domain
$VEnv$	=	$V \rightarrow D$	Variable Environments
$FEnv$	=	$F \rightarrow D^* \rightarrow D$	Propagation Function Environments

Figure 18: Domains for Propagation Analysis

$$\mathcal{O}[\![op]\!] = \lambda x_1, \ldots, x_n.\emptyset \text{ where } op \in \{ \texttt{upd}, \texttt{left\_part}, \texttt{right\_part}, \texttt{array}, \texttt{combine}, \ldots \}$$
$$\mathcal{O}[\![op]\!] = \lambda x_1, \ldots, x_n.\{b\} \text{ where } op \in \text{is } \{ \texttt{+}, \texttt{-}, \texttt{sel}, \ldots \}$$

Figure 19: Propagation Functions for Primitive Operators

$$\mathcal{H} : IExp \rightarrow VEnv \rightarrow FEnv \rightarrow D$$

$$\mathcal{H}[\![\mathrm{c}]\!]\sigma\rho = \{b\}$$
$$\mathcal{H}[\![x]\!]\sigma\rho = \sigma[x]$$
$$\mathcal{H}[\![t_i]\!]\sigma\ \rho = \textit{let } v = \mathcal{H}[\![expr\_of(t_i)]\!]\sigma\ \rho \textit{ in if } (v = \emptyset) \textit{ then } \{t_i\} \textit{ else } v$$
$$\mathcal{H}[\![op(se_1, \ldots, se_n)]\!]\sigma\rho = \mathcal{O}[\![op]\!](\mathcal{H}[\![e_1]\!]\sigma\ \rho, \ldots, \mathcal{H}[\![e_n]\!]\sigma\ \rho)$$
$$\mathcal{H}[\![\texttt{if } se_0 \texttt{ then } e_1 \texttt{ else } e_2\ ]\!]\sigma\rho = \mathcal{H}[\![e_1]\!]\sigma\rho \ \cup\ \mathcal{H}[\![e_2]\!]\sigma\rho$$
$$\mathcal{H}[\![f_k(se_1, \ldots, se_n)]\!]\sigma\rho = \rho[f_k](\mathcal{H}[\![se_1]\!]\sigma\rho, \ldots, \mathcal{H}[\![se_n]\!]\sigma\rho)$$
$$\mathcal{H}[\![\ \texttt{let } [t_1 = e_1, \ldots, t_n = e_n] \texttt{ in } \ t_i \texttt{ end }\ ]\!]\sigma\rho = \mathcal{H}[\![t_i]\!]\sigma\ \rho$$

Figure 20: The Function $\mathcal{H}$

$$\mathcal{H}_p : IProg \rightarrow FEnv$$

$$\mathcal{H}_p[\![pr]\!] = fix(\lambda\rho.\ \rho[f_i \mapsto \lambda y_1, \ldots, y_k.\ (\mathcal{H}[\![e_i]\!][x_{1_i} \mapsto y_1, \ldots, x_{k_i} \mapsto y_k]\rho) \cap (\bigcup_{j=1}^{k} y_j)])$$

Figure 21: The Function $\mathcal{H}_p$

can be shared in the rest of the `let` expression through the name t_i. Therefore, an anonymous aggregate can always be identified with the name of the `let` variable to which it is bound. When an anonymous aggregate is returned as a result of the function, any local name that was associated with the result can be discarded because the scope of the local name is limited to the `let` expression in which it is introduced.

In our previous analysis we used the emptyset to denote an anonymous aggregate as well as a non-aggregate value. In this chapter we introduce a new value b to represent a non-aggregate value. The emptyset denotes an anonymous aggregate. The reason for this change is that it reduces the complexity of the algorithm.

The domains and functions for propagation analysis are given in Figures 18, 19, 20, and 21. Details of the rest of the analyses are very similar to the ones in Chapter II and have been omitted.

Aliasing analysis is exactly the same as described in Chapter II. The *selects-updates* analysis is also the same except that `partition` and `combine` neither select nor update any of their arguments.

The dependence graph of a `let` expression is updated by adding an edge between node t_i and t_j if t_j is not a predecessor of t_i and t_i updates any array that is selected by t_j. After adding these edges, the dependence graph remains a directed acyclic graph. In Chapter II, we added edges without checking for the existence of a path between the two nodes, which could result in a graph with cycles. The dependence graph is used to compute the set of syntactically live variables at any binding. Our current decision is to sacrifice parallelism in favor of destructive updating, whenever there is a conflict between the two.

3.3.1 Computing Live Variables

Given a dependence graph where each node represents an expression (and the corresponding temporary variable), we have to determine the variables that are live at each node. A variable x is live at t_i if x is neither t_i nor its successor and there exists another node $t_j \neq t_i$ that is not yet evaluated and uses x. The reason we don't need to consider t_i or its successors is that these variables are not defined before the evaluation of t_i. Since we are considering parallel evaluation, we conservatively assume that all the nodes that are not predecessors of t_i are yet to be evaluated.

Recall that in the intermediate language (see Figure 17), we split a `partition` operator into two `left_part` and `right_part` operators which inherit the label of the `partition` operator. Since `partition` is a single operator at the source level, the variables used in a `left_part` operator with label `1` need not be considered as live at the node `right_part` with the same label. In general, variables in nodes with the same label as that of t_i need not be considered in determining the live variables at t_i. Therefore the set of live variables at t_i is given as

$$Live(t_i) = \bigcup \{x \mid \exists j \; x \in Vars(t_j), t_j \notin preds(t_i), x \notin (succs(t_i) \cup \{t_i\}), \\ label(t_i) \neq label(t_j)\}$$

Consider the intermediate form for the helper function in vector addition example.

```
help_vector_add(a,b,i) =
  let t_1 = dim(a,1);
      t_2 = (t_1 = 1);
      t_3 = if t_1 then
              let t_4 = a[i];
                  t_5 = b[i];
                  t_6 = t_4 + t_5;
                  t_7 = upd(a,i,t_6)
              in
                 t_7
              end
            else
              let  t_8  = t_1//2;
                   mid  = i + t_8;
                   a_1  = left_part (a,mid,1);
                   a_2  = right_part(a,mid,1);
                   rv_1 = help_vector_add(a_1,b,i);
                   rv_2 = help_vector_add(a_2,b,mid);
                   t_10 = combine(rv_1,rv_2,1)
              in
                 t_10
              end
  in
     t_3
  end
```

The live variables at a_1 and a_2 are {b,i,a_2} and {b,i,a_1} respectively.

3.3.2 Complexity

We show that the complexity of the analysis remains polynomial as in our earlier algorithm. The parameters are m (the number of internal nodes in the parse tree of a program, k (the maximum function arity), n (the number of functions), and p (the maximum number of operators that return anonymous aggregates). The parameter p includes the `partition`, `combine`, and upd operators and all the function calls that return new aggregates.

Since temporary variables are discarded when considering the value propagated by any function, the number of fixpoint iterations needed for all the analyses is the same as in Chapter II. Recall that propagation analysis requires $O(nk)$ iterations.

The maximum number of values that can be propagated by an expression is $k + p + 1$; 1 represents the non-aggregate value, although it will be a type-error to return an aggregate as well as non-aggregate value. Analyzing a function call, the most expensive operation, requires at most k unions of sets of size at most $k + p + 1$. The overall complexity of propagation analysis becomes $O(mnk^2(k+p))$. Recall from previous chapter that the worst case complexity of propagation analysis was $O(mnk^3)$. If we had not distinguished between non-aggregate values and anonymous aggregates, then the maximum size of a set would have been $O(m)$ instead of $O(k + p)$.

Similarly it can be shown that the worst case complexity of *aliasing analysis* is $O(mnk^4(k + p))$, although in practice it takes only a few iterations. *selects-updates analysis* and *reference count analysis* take $O(mnk^2(k + p))$ time. For reference count analysis, we assume that the live variables are already computed as discussed in previous subsection. In the above estimates, we haven't included the complexity of adding edges to the dependence graph and computing the live variables, both of which can be shown to be of polynomial complexity.

The main intent of the complexity estimate is to show that the analysis runs in polynomial time, even with parallel evaluation. For typical cases, these worst case estimates do not reflect the actual running times. Our previous analysis, for example, runs in near linear time on typical programs.

3.3.3 Optimizing Combines

Suppose after the update analysis we discover that a `partition` operator can be converted into a non-copying `partition!`, as is the case if its first argument is not live. The question is: can we always convert a `combine` to a non-copying `combine`? The answer is no. Consider the function `f (x,y) = combine(x,y,1)`. Since we don't

know anything about the storage layout of `x` and `y`, `combine` cannot be known to be non-copying at compile time.

$$
\begin{array}{lcl}
Part & = & \{\text{Occurrences of Partition Operators}\} \\
P & = & Part \times \{l, r\} \\
Add & = & ((V + \{\top\}) \times P^*)^{\top}_{\bot}
\end{array}
$$

Figure 22: Domain of Array Addresses

$$
\begin{array}{lcl}
\mathcal{K}[\![\texttt{upd!}]\!]\ x & = & x \\
\mathcal{K}[\![\texttt{upd}]\!]\ x & = & \top \\
\mathcal{K}[\![^{a}\texttt{left\_part}]\!]\ x & = & \top \\
\mathcal{K}[\![^{a}\texttt{right\_part}]\!]\ x & = & \top \\
\mathcal{K}[\![^{a}\texttt{left\_part!}]\!]\ x & = & \mathit{if}\ (x = \top) \\
 & & \mathit{then}\ \langle \top, a.l \rangle \\
 & & \mathit{else}\ \langle \mathit{fst}(x),\ \mathit{rest}(x).a.l \rangle \\
\mathcal{K}[\![^{a}\texttt{right\_part!}]\!]\ x & = & \mathit{if}\ (x = \top) \\
 & & \mathit{then}\ \langle \top,\ a.r \rangle \\
 & & \mathit{else}\ \langle \mathit{fst}(x),\ \mathit{rest}(x).a.r \rangle \\
\mathcal{K}[\![\texttt{combine}]\!]\ x\ y & = & \mathit{if}\ (x = \top\ \mathit{or}\ y = \top) \\
 & & \mathit{then}\ \top \\
 & & \mathit{else\ if}\ (\mathit{fst}(x) = \mathit{fst}(y)\ \mathit{and} \\
 & & \quad (\mathit{rest}(x) = s.b.l)\ \mathit{and} \\
 & & \quad (\mathit{rest}(y) = s.b.r) \\
 & & \mathit{then}\ \langle \mathit{fst}(x),\ s \rangle \\
 & & \mathit{else}\ \top
\end{array}
$$

Figure 23: The Function $\mathcal{K}$

We can perform a simple analysis to detect if `combine` can be non-copying. We use an analysis similar to propagation analysis that determines addresses propagated by an expression. The flat domain of addresses is given in Figure 22. $\top$ represents an unknown address. An address $\langle v, s \rangle$ is the address of an array obtained by applying a sequence of s `left_part` or `right_part` operators on the array with address v. The function $\mathcal{K}$ which defines the address propagation behavior of primitive operators is

given in Figure 23.

All operators are bottom strict. The function *fst* returns the first element of a sequence; the function *rest* returns the tail of a sequence; the function . returns a new sequence by concatenating an element to a sequence. All operators except `combine` that return new aggregates return $\top$ as the address. We also assume that arithmetic operators return $\top$. The operators with a ! are non-copying. If the two arguments to `combine` are addresses of the left and right partitions created by a single partition operator whose label is b then the result is obtained by removing the last two elements in the sequence of the first argument to `combine`. In all other cases, the result is $\top$. The interprocedural analysis for address propagation can be defined using the function $\mathcal{K}$. If a function returns an address $\langle v, s \rangle$ where s is non-empty, it is replaced by $\top$. In other words, information about the partition of an array created inside a function and returned as its result is forgotten outside the function as shown by the example below.

```
f x =
     let t1,t2 = partition(x,1,1)
     in
        t1
     end
```

Variable `t1` gets the value $\langle x, 1.l \rangle$ where 1 is the label of `partition`). Since the result is of the form $\langle x, s \rangle$ where s is not the empty sequence, it is immediately changed to $\top$. The address propagation function for `f` is $f\ x = \top$.

Now consider the helper function for vector addition described in Section 3.2. After update analysis, the function is

```
help_vector_add (A,b,i) =
  if dim(A,1) = 1 then upd!(A,i,A[i]+B[i])
  else
    let midpoint = i + dim(a.1)//2;
        a_1 = left-part!(a,midpoint,1);
        a_2 = right-part!(a,midpoint,1);
        rv_1 = help_vector_add a_1 b i;
        rv_2 = help_vector_add a_2 b midpoint
    in
       combine(rv_1, rv_2,1)
    end
  endif;
```

For the purposes of address propagation, the flow equation is

$$\begin{aligned} vadd(a,B,i) \;=\; & a \sqcup \\ & \mathcal{K}[\![\texttt{combine}]\!]\; vadd(\langle fst(a),\; rest(a).1.l\rangle, B, i) \\ & \qquad vadd(\langle fst(a),\; rest(a).1.r\rangle, B, \top) \end{aligned}$$

The equation can be solved by fixpoint computation as

$$\begin{aligned} vadd^0(a,B,i) &= \bot \\ vadd^1(a,B,i) &= a \\ vadd^2(a,B,i) &= a \sqcup \\ & \qquad \mathcal{K}[\![\texttt{combine}]\!]\; \langle fst(a),\; rest(a).1.l\rangle \\ & \qquad\qquad \langle fst(a),\; rest(a).1.r\rangle \\ &= a \end{aligned}$$

If any of the arguments to a combine operator is $\top$ or of the form $\langle x, s_1\rangle$ and $\langle y, s_2\rangle$ and either $x \neq y$ or s_1 and s_2 are not of the form $s.b.l$ and $s.b.r$ respectively

then we cannot make that `combine` non-copying. The fixpoint can be computed in linear time (at most 2n iterations where n is the number of functions).

Interaction with update analysis

Given that a `combine` operator could be made non-copying, we can make it actually non-copying, provided the two arguments to combine are not live. We need this condition because our update analysis assumes that combine always returns a new array that is not live elsewhere.

Even if a `combine` is not known to be non-copying at compile time, it is not always the case that it requires copying. A simple test at run-time can check if the two arguments are actually contiguous and then avoid copying if the arguments to `combine` are not live. The liveness information is available from update analysis. The real advantage of a statically non-copying combine operation is that in the absence of bounds checking and operators that access the array header, both partition and combine operations can be converted into identity operations, thus avoiding the overheads of header creation.

3.4 Experimental Results

We have designed and implemented a compiler that takes a source program of our language and generates a Scheme or C program. The main phases of the compiler are alpha renaming, cycle detection among `let` bindings, flattening of `let` expressions and conversion to the intermediate form, common subexpression elimination, and update analysis. The compiler is implemented in Standard ML.

Our example programs are mostly taken from numerical computations [35, 34]. These examples can be classified as direct methods for solving linear equations, iter-

ative methods, and miscellaneous. Direct methods include gaussian elimination with and without partial pivoting, and various matrix factorizations such as LU, QR, and Cholesky. Examples of iterative methods are point jacobi, red-black method, successive over-relaxation, conjugate gradient method and the multigrid method for solving partial differential equations numerically. All these examples are written in a recursive style using `partition`, `combine`, and upd operations with implicit parallelism.

Table 3: Results of Update Analysis of Parallel Functional Programs

Program	*Size* (m,n,k)	*No. of upds*	*Destructive upds*	*Analysis time* (*in secs*)
LU	(78,3,7)	2	2	0.59
cholesky	(86,3,7)	4	4	0.71
QR	(181,13,7)	6	6	1.49
gauss	(126,6,7)	8	8	1.24
gauss (with pivoting)	(154,10,7)	12	12	1.91
jacobi	(32,3,7)	1	1	0.29
red-black	(68,6,8)	4	4	0.56
conjugate gradient	(84,9,6)	2	2	0.61
SOR	(78,6,9)	4	4	0.74
multigrid	(163,9,6)	4	4	0.86
matmul	(54,2,7)	1	1	0.41
prefix sum	(26,2,3)	1	1	0.12
quicksort	(42,5,4)	4	4	0.21
vadd	(14,1,3)	1	1	0.05
ram simulator	(59,1,6)	7	7	0.33

The miscellaneous examples include basic operations such as vector addition, matrix multiplication, prefix computation, and quicksort.

Table 3 shows the effectiveness of our algorithm. In these examples, all updates are made destructive. The program size is characterized by m: the number of nodes in the parse tree, n: the number of functions in the program, and k: the maximum function arity. The last column of the table shows the analysis time on a

3.5 On Programming Style and Predictability

We have demonstrated that our update analysis algorithm is effective and efficient. We now address the question of whether programmers will be able to understand the update optimization well enough to write efficient code.

This is an important question because update analysis is a powerful optimization that can easily change the complexity of an algorithm by orders of magnitude. Programmers need to know whether the code they write is efficient. It would be disastrous for programmers to write functional programs that they mistakenly believe will be made efficient by update analysis.

For this reason we believe that functional languages should be equipped with two kinds of update operator: a copying update and a destructive update. These operators would both have the purely functional semantics of the copying update operator, but their pragmatics would differ: The compiler would refuse to accept any program that contains a destructive update that our update analysis algorithm cannot prove to be equivalent to a copying update. The efficiency of an update operation would then be clear to programmers: A destructive update executes in constant time, but a copying update must be assumed to be inefficient until proved otherwise (by changing it to a destructive update and passing it through the compiler).

Programmers would find this very frustrating if the compiler were unable to accept destructive updates that the programmer knows are safe, but our update analysis is so effective that this would hardly ever happen.

Programmers would also be frustrated if they were unable to understand why the compiler rejects a destructive update. We know, however, that programmers will be able to understand the outcome of update optimization because they are able

to understand a similar but more difficult issue—not perfectly, but well enough: the problem of dropping all pointers to a data structure so it can be garbage collected. The garbage collection problem is dynamic, and it involves dropping all pointers, whereas the update problem is static and involves dropping all but one pointer; otherwise these two problems are the same.

Programmers do struggle with the garbage collection problem, and not always successfully, but they do well enough. The penalty for failing to drop all pointers to a structure is that the program is less efficient than it should be, and may catastrophically run out of space when it shouldn't. The penalty for failing to drop all but one pointer to an update structure is that the update operator will have to be changed to a copying update operator, and the program will be less efficient than it should be. At least the programmer will know the program is inefficient, which is not always true with the garbage collection problem.

Furthermore the compiler can *explain* why it thinks a destructive update is unsafe. If aliasing is the problem, then the compiler can report the variables that it fears may be aliased. If two expressions update the same array, the compiler will detect the problem while adding precedence edges to the dependence graph. Again, the compiler can indicate the expressions that interfere.

Our analysis is independent of the choice of order of evaluation, so long as there exists any order of evaluation for which the compiler can prove that all destructive updates are safe. Therefore the compiler would not be sensitive to the order in which formal parameters are declared. (This is a distinct improvement over previous algorithms [13, 45].)

The `partition` and `combine` operators can be used efficiently by following a

few simple rules. When an array is partitioned, for example, it should not be live elsewhere. Neither partition of an array should be returned as a result of a function. Every function should have a matching number of `partition` and `combine` operators. The left and right arrays that result from a `partition` operation should be the left and right arguments to a subsequent `combine` operator within the same function body, and the proof of this should be obvious to the programmer (so it will also be obvious to the compiler). If these simple rules are followed, then most unnecessary copying can be avoided.

Sometimes it is possible to reduce copying in one part of a program by introducing copying in another part. This is very hard for the compiler to notice, but easy for the compiler to confirm once it is pointed out. We are led therefore to propose an explicit copy operation that has the semantics of the identity function but serves also as a declaration. Explicit copy operations declare not only the programmer's awareness that copying will be required, but they also declare the places in the program where the programmer believes copying should occur in order to obtain the most efficient results.

Since the copying update operation that we took as the starting point for our research is equivalent to a composition of the destructive update and copy operators, and the copy operator is more versatile than the copying update operator, we refine our proposal by suggesting that functional languages should replace the copying incremental update operator by the combination of an explicit copy operator and an explicitly destructive update operator—both of which, we hasten to add, have a purely functional semantics provided the compiler refuses to accept any destructive updates that cannot be proved to be equivalent to the traditional copying update.

It may fairly be said that we are advocating a more imperative approach to functional programming. We believe this is consistent with other recent research into the problem of state in functional languages. We suggest that recent research may even be leading toward a rejuvenating redesign of imperative languages from a functional perspective, which would not be a bad thing at all.

For predictability and portability, compilers should behave uniformly. Update optimization could be implemented just as uniformly across compilers as tail-recursion optimization and type-checking. Our update analysis is no more complicated than type-checking in ML, and we believe programmers will find update analysis at least as easy to understand as ML-style type-checking.

3.6 Expressing Collection of Updates

In this section, we define a new operation on arrays to express a collection of updates. This operation has not yet been incorporated in our language. The partition operation is useful for expressing parallel updates when it is known at compile time that the updates are on distinct indices of the array. Our experience has been that for several numerical algorithms, the `partition` operator suffices for expressing parallelism. However, there are cases when either the updates are not known to be distinct at compile time or there are multiple updates at the same index. The histogram and polynomial multiplication problems require updating at the same index. For the inverse permutation problem, the updates are performed on distinct indices but this is not known at compile time. How does one express such multiple updates without losing deterministic behavior?

We define a new update operator called an *accumulating update*. Given a suit-

able operator $\oplus$, the corresponding accumulating update is written as

$$A\{i \oplus= v\}.$$

It returns a new array like A except that at index i its value is $A[i] \oplus v$. Analogously, one can also define another operator

$$A\{i =\oplus v\}$$

for which the new array has a value $v \oplus A[i]$ at index i. Using the accumulating update operator, we can now specify a collection of updates on an array as

$$A\{\langle e_{index} \rangle \oplus= e \mid e_1 \leq i \leq e_2\}$$

The index expression e_{index} and e can have i as a free variable. This expression describes a set of updates one for each value of i from e_1 to e_n. There can be more than one generator. We need to require a certain property of the update operator to ensure a deterministic result. Consider two updates v_1 and v_2 at index i. In order for the result to be the same at the end of the updates, we require that

$$A\{i \oplus= v_1\}\{i \oplus= v_2\} = A\{i \oplus= v_2\}\{i \oplus= v_1\}.$$

In other words, $(A[i] \oplus v_1) \oplus v_2 = (A[i] \oplus v_2) \oplus v_1$. Thus we require that $\oplus$ obey the following identity.

$$(a \oplus b) \oplus c = (a \oplus c) \oplus b$$

Any operator that is associative and commutative has this property.

Since the order in which the updates are performed does not matter as long as they are serialized, multiple updates can be implemented on a shared memory multiprocessor using an extra array of locks. The overall space complexity is $O(n)$ where n is the size of the array.

An optimization useful for implementing a collection of updates is to avoid locks whenever it can be determined that the updates are all disjoint. One simple case that occurs very commonly is with updates of the form

$$A\{\langle i \rangle \oplus= \ e \mid l \leq i \leq b\}$$

We know by the nature of the generator that all values of i are distinct. Therefore all updates are on disjoint indices and no synchronization is required. In such a case we can even replace $\oplus=$ by $=$ if we know that the initial array contains the identity element of the operator. For example, vector addition can be written as

```
vadd(A,B) =  A{<i> += B[i] | 0 <= i < dim(A,1)}
```

If `A` is not known to be live elsewhere, then it can be updated destructively in parallel. Compare this program with the program using `partition` and `combine`. Currently we are investigating the issues involved in the compilation of the collection of updates operator.

3.6.1 Examples

Given an array of n numbers ranging from 0 to $m-1$, the histogram problem is to compute the number of occurrences of each element in the array. A one line solution using a collection of updates is

```
hist(A,n,m) = array(m,0){<A[i]> += 1 | 0 <= i < n }
```

Polynomial multiplication can be expressed naturally by a collection of updates with + as the accumulating operator.

```
pmult(A,B,m,n) =
  let a = array(m+n,0);
  in
    a{<i+j> += A[i]*B[j] | 0 <= i < m, 0 <= j < n}
  end
```

The inverse permutation problem takes an array I that holds a permutation of 0 to n-1 and returns an array A such that A[I[i]] = i. The difficulty is that it is not known at compile time if I is a permutation. We define an operator $*$ and an identity element e such that $x*e = e*x = x$ and $x*y = n$ otherwise. The inverse permutation problem can then be written as

```
inv_perm(I,n) = array(n,e){<I[i]> *= i | 0 <= i < n}
```

A monolithic array is a array all of whose elements are defined once [55]. A monolithic array construct takes f and n as arguments and returns a new array whose value at i is $f(i)$. It can be described by using the collection of updates as

```
marray(f,n) = array(n,0){<i> = f i | 0 <= i < n }
```

We do not need any accumulating operator, because from the syntax we know that all the updates are on disjoint indices. One can also write functions to compute scan primitives such as prefix sum, array compaction, copying, enumerate, and distribute-sums used in data-parallel computing[11].

3.7 Summary

In this chapter we have presented a strict functional language with incremental updates, partitioning and combining operations, and a collection of updates for expressing parallelism. We have described an efficient update analysis algorithm and its performance on typical numerical algorithms. We have considered the implications of our algorithm for language design, and have explained why we believe programmers will be able to write efficient programs that rely on update optimization. In the next chapter, we describe the compiler and the runtime system for our parallel functional language on a shared memory multiprocessor system.

CHAPTER IV

A COMPILER AND A RUNTIME SYSTEM

4.1 Introduction

In this chapter we describe a compiler and a runtime system for the parallel functional language described in Chapter III. The target machine is a shared memory multiprocessor. In the next section we describe the important phases of the compiler. In Section 4.3, we illustrate the code generation phase with a few examples followed by a detailed description of the algorithm. In Section 4.4 we describe the scenarios for supporting update analysis with separate compilation. In Section 4.5 we describe our runtime system. We conclude the chapter with preliminary performance results.

4.2 Compiler Phases

The important phases of the compiler are preprocessing, update analysis, address propagation, and code generation. For portability, we have chosen C as the target language. The compiler is written in Standard ML of New Jersey, with each of these phases implemented as an ML module.

4.2.1 Preprocessing

During preprocessing, a source program is converted into the intermediate language suitable for update analysis. Certain auxiliary functions useful in the later phases are also defined.

After parsing the source program, the first step is cycle detection among parallel `let` bindings. Recall that the `let` expression of the source language does not specify any ordering among `let` bindings. Cycles are detected by constructing a dependence graph of the `let` bindings. Compilation is aborted if a cycle is detected.

Cycle detection is followed by conversion of the source program to the intermediate language given in Chapter III. Every subexpression that is not a constant or a variable is given a unique variable name. Nested `let` bindings are flattened. For examples, the body of the function

```
f x = let a = let y = 3*z;
                  z = 2*x
              in
                y+z
              end
      in
        a + x*3
      end
```

is converted to

```
f x = let t_1 = 2*x;
          t_2 = 3*t_1;
          t_3 = t_1 + t_2;
          t_4 = 3*x;
          t_5 = t_4 + t_3
      in
         t_5
      end
```

The ordering among the `let`-bindings is the data dependency ordering. Common subexpression elimination and type-checking are performed on the representation of the program in the intermediate language.

The last phase of preprocessing is alpha conversion where every variable and function symbol is given a unique name. After this conversion, each program variable is of the form t_{ij} meaning that it is the jth variable of the function i. The advantage

of this representation is that information about program variables is represented by a two dimensional array, providing easy access and update.

Auxiliary Functions

The preprocessing phase also defines the following functions used in later phases.

expr_of : takes a program variable and returns the corresponding expression. For instance, *expr_of*(t_3) in the previous example is t_1+t_2. If the variable is a formal parameter, *expr_of* returns the variable itself. This function is used in the computation of $\mathcal{H}$ defined in previous chapters.

free_vars : takes a variable t_i and returns the free variables of *expr_of*(t_i). The free variables of an expression are given by the function FV defined below.

$$
\begin{aligned}
FV(c) &= \emptyset \\
FV(x) &= \{x\} \\
FV(op(se_1, \ldots, se_n)) &= \bigcup_{i=1}^{n} FV(se_i) \\
FV(f(se_i, \ldots, se_n)) &= \bigcup_{i=1}^{n} FV(se_i) \\
FV(\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2) &= \bigcup_{i=0}^{2} FV(e_i) \\
FV(\texttt{let } [t_1 = e_1, \ldots, t_n = e_n] \texttt{ in } t_i \texttt{ end}) &= \bigcup_{i=1}^{n} FV(e_i) \setminus \{t_1, \ldots, t_n\}
\end{aligned}
$$

We construct the dependence graph of every `let` expression of the intermediate program.

dep_graph_of : takes a program variable representing the result of a `let` expression and returns the dependence graph of the `let` expression. For all other variables, it

returns the empty graph.

4.2.2 Update Analysis

Update Analysis consists of five phases. These five phases are *propagation analysis*, *aliasing analysis*, *selects-updates analysis*, *mark dependencies phase*, and *reference count analysis*. The dependencies among these phases are shown in Figure 24.

Propagation Analysis

This phase implements the functions $\mathcal{H}$ and $\mathcal{H}_p$ of a program. It takes a program p and returns the propagation environment $\mathcal{H}_p[\![\mathrm{p}]\!]$. It also provides a function *prop_info* that returns the variables propagated by a temporary variable in the program.

Aliasing Analysis

This phase implements the functions $\mathcal{A}$ and $\mathcal{A}_p$ of a program. Aliasing information of a program p is $\mathcal{A}_p[\![\mathrm{p}]\!]$. It uses the propagation information computed by propagation analysis.

Selects-Updates Analysis

This phase implements the functions $\mathcal{S}$ and $\mathcal{S}_p$ of a program. It uses the propagation information computed earlier. In addition it also provides a function *su_info* that returns the variables selected and updated by the expressions corresponding to every temporary variable in the program.

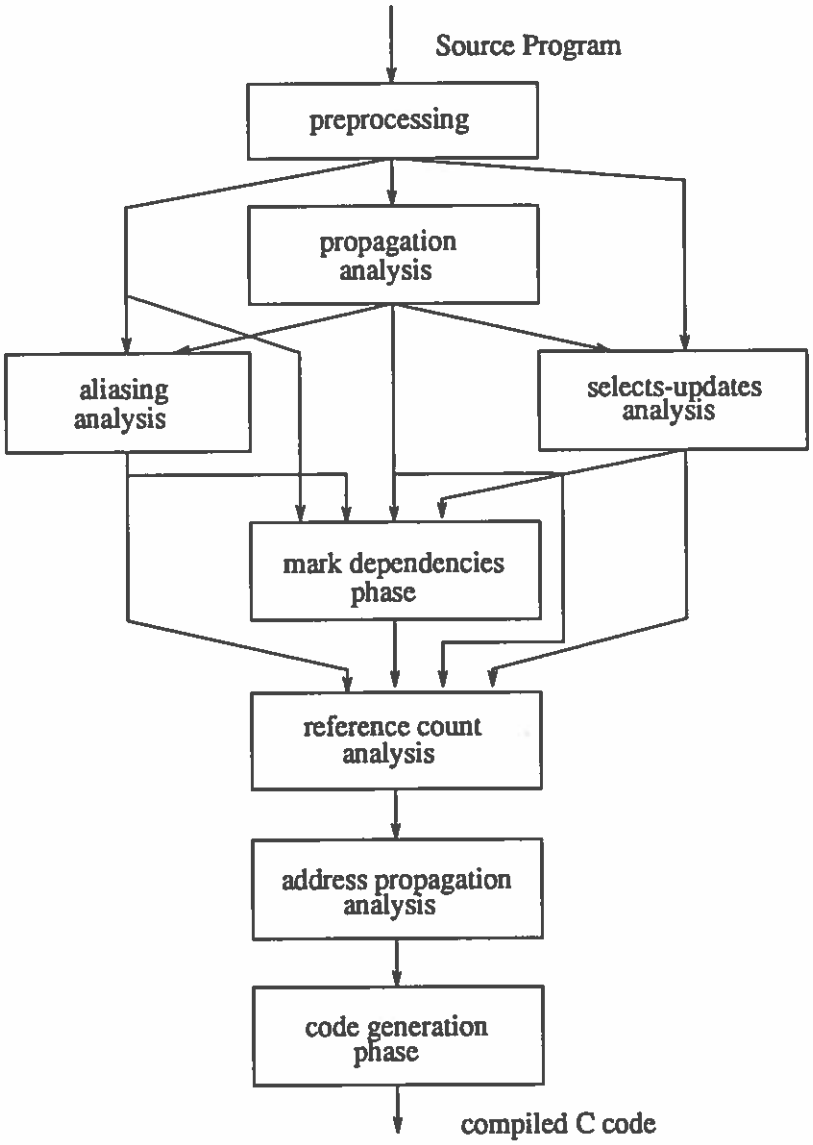

Figure 24: The Structure of the Compiler

Dependency Marking

In this phase new precedences are added to the dependence graph of every `let` expression. Given a `let` expression, the corresponding dependence graph is obtained using the function *dep_graph_of*. A directed edge (t_i, t_j) is added if neither t_i nor t_j is a predecessor of the other and the variables (and their aliases) updated by t_i are selected or updated by t_j. This phase uses the information provided by *aliasing analysis* and *selects-updates analysis*.

Reference Count Analysis

This phase implements the functions $\mathcal{R}$ and $\mathcal{R}_p$. Live variables at each program point are computed as described in the previous chapter. At the end of this phase, every update that can be done destructively is converted to `upd!`. Using the same criterion as that of destructive updating, we convert `left_part`, `right_part`, and `combine` operators to `left_part!`, `right_part!`, and `combine!` respectively.

4.2.3 Optimizing Partitions/Combines

In this phase, we perform the address propagation analysis described in Chapter III to determine if a `combine!` can be implemented by sharing as described in the previous chapter. This phase is not yet implemented.

4.3 Code Generation

We have chosen C as a target language of the code generator, a decision taken by several researchers compiling high-level functional languages [73, 25, 18, 59] and logic languages [39]. One advantage of compiling to C is portability because C compilers are

expected to be available on most machines. Another advantage is ease of debugging, since C is a high-level language. Moreover, C optimizers can be used to improve the performance of the resulting program.

However, there are two problems with C as the target language. First is implementing proper tail-recursion. Most C compilers do not perform a fully general tail-recursion optimization, so a straightforward translation of a function in the source language to the corresponding C function is not desirable. We have to abandon the C stack and the C function calling convention and maintain the stack ourselves. This immediately raises another issue: how to represent return addresses or continuations? In C, statement labels are not storable values, therefore one cannot use C labels as return addresses, a flexibility one enjoys using in assembly language. We have to encode return addresses somehow. This causes extra overhead for encoding and decoding return addresses.

The second problem is the use of machine registers. The C programmer does not have control over allocation of machine registers to program variables. When we do not use the function calling mechanism of C, we also lose the advantage of passing function arguments in the machine registers on some architectures, and of having the stack pointer as one of the machine registers. Register declarations available in C are ignored by most C compilers and are in any case useless for interprocedural register allocation. The GNU C compiler allows targetting variables to registers by using assembly language statements, which obviously would limit portability. For these reasons, which have to do with the inefficiency of C rather than any inefficiency of our functional language, the performance of our compiled code should be expected to be worse than the performance of a comparable program written directly in C.

4.3.1 Representing Continuations

A continuation is a pair consisting of some encoding of the function representing the rest of the program and its live variables. In an assembly language one can use instruction labels to represent the rest of the program. Creating a continuation before a function call means saving the return address and the live variables on the stack. Depending on the convention, live variables of the continuation are saved and restored either by the caller or the callee. The problem is how to represent continuations in C since we cannot use C statement labels.

One way is to encode statement labels by storable values. For instance, we could encode labels by integers and use a `switch` statement for decoding labels. Suppose that an integer variable `label` holds a label. The `switch` statement would be

```
switch (label) {
   case 0 : goto l_0;
   case 1 : goto l_1;
            .
            .
            .
   case n : goto l_n;
}
```

The labels `l_0`,..., `l_n` are C labels. The entire source program would be compiled into one giant C `switch` statement. Each basic block of a source function would be compiled as a block of statements of a `case` label of the `switch` statement. A function call would be implemented by assigning the appropriate value to `label` and reexecuting the `switch` statement. A return address represented as an integer would be a storable value. Jumping to the return address would be achieved by assigning the corresponding integer to `label` and executing the `switch` statement.

There are a couple of problems with this approach. One is the overhead of the

`switch` statement. The second and more serious problem is that the entire program would have to be compiled as one single `switch` statement. Existing C compilers have problems compiling large functions. Therefore compiling large programs would be problematic. Separate compilation would also pose a problem. In view of these reasons we chose another approach.

In the second approach, a function in the source language is compiled into a set of parameterless C functions — each one representing a basic block of the source function. Transfer of control from one basic block to another is via a C function call. A return address is a C function pointer, which is a storable value. Continuation is a pair: a C function pointer and a set of variables.

The disadvantage of this method is that every basic block entry and exit is implemented as a function call which is typically an expensive operation in C. But a post-pass of the assembly code can convert calls to known functions into jumps, thus avoiding the overhead of a function call as well as the interpretive loop, thereby improving efficiency. Such a technique has been adopted in The Glasgow Haskell implementation [59]. In our current compiler, the post-pass has not been implemented.

4.3.2 The Abstract Machine

The abstract machine consists of several processors accessing a global shared memory through some interconnection network, as shown in Figure 25. Each processor has some private memory which is exclusive to the processor. We call the local memory the *Processor Stack*. Global memory is used for allocating global data structures such as arrays, and for sharing information among processors.

Each processor has a set of argument registers, a stack pointer pointing to its stack, a heap pointer pointing to the global shared memory, and a register called code

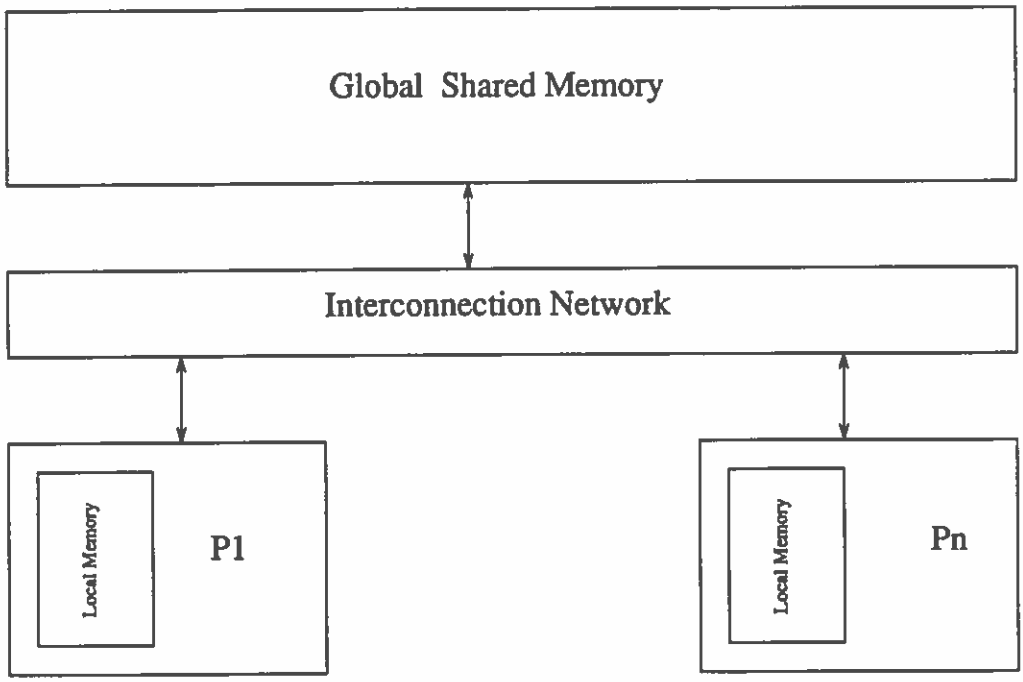

Figure 25: The Abstract Multiprocessor Machine

which is equivalent to the program counter. Each processor executes the following interpretive loop:

```
while (1) (*code)();
```

The important abstract instructions of the machine are control instructions such as `call` and `return`; the instructions `fork`, `spawn`, and `synchronize` for forking and synchronizing parallel processes; the memory instructions such as `push`, `pop`, `move`, `allocate`, and `deallocate`; and the primitive operations such as `+`, `-`, `if ...` that are available in most languages.

Sychronization of processes is implemented on the processor stack. Before spawning processes, the `fork` allocates a synchronization counter on the stack which is initialized to the number of parallel processes. A process for a function call `f(t_1,...,t_n)` is created by the `spawn` instruction, which moves the data for ex-

ecuting the function to the global shared memory. Subsequently, synchronization of processes is achieved by decrementing the synchronization counter by executing the `join` instruction.

Parameters to a function call are passed in the argument registers. The result of a function is returned in register 0. Before a call to a function `f`, the caller moves the actual parameters to the argument registers using `move` and for a non-tail-recursive call, creates the continuation; the function is invoked by the `call` instruction.

Implementing the Abstract Instructions

Each instruction is implemented as a C macro. Instructions such as `push`, `pop`, `move`, `allocate`, `deallocate`, the primitive operations, and the instructions for spawning processes have a straightforward translation as C macros. A function call to `f` is implemented by assigning `f` to `code` and transferring control to the interpretive loop. The C macro for `call` is defined below.

```
#define call(f) code = f;return;
```

The `return` instruction is implemented as `call(exec_return)`. The function `exec_return` pops the current stack top which points to a parameterless C function, the return address, and executes it.

This idea has been used in the Glasgow Haskell compiler [59], and seems to have originated as the "UUO handler" in Guy Steele's Rabbit compiler [69] for Scheme.

4.3.3 Examples

Now we consider a few example programs to illustrate the code generated by the compiler, deferring the details of code generation to a later section. Consider the source program for the non-tail recursive factorial function and its intermediate form

in Figure 26.

```
/* source program */

 fact n = if (n = 0) then 1
          else
             n*fact(n-1)
          endif;

/* the intermediate form */

 fact0 n = let t0 = (n = 0);
               t1 = if t0 then
                           let
                           in
                             1
                           end
                    else let t2 = n - 1;
                             t3 = fact(t2);
                             t4 = n*t3;
                         in
                           t4
                         end
           in
             t1
           end
```

Figure 26: Factorial Function and its Intermediate Form.

The intermediate representation of the factorial function can be thought of as a macro data flow graph as shown in Figure 27. Each node of the graph is a set of variable definitions, and an edge represents the flow of values from one node to another. Each node itself is a basic block which is a directed acylic graph with variables defined in the basic block as the vertices and their dependencies as the edges.

The code generation algorithm compiles each block into a C function, coalescing blocks if possible. Live variables are saved on the stack at each block exit. A function call or an if expression always terminates a block. The expressions `fact(t2)` and

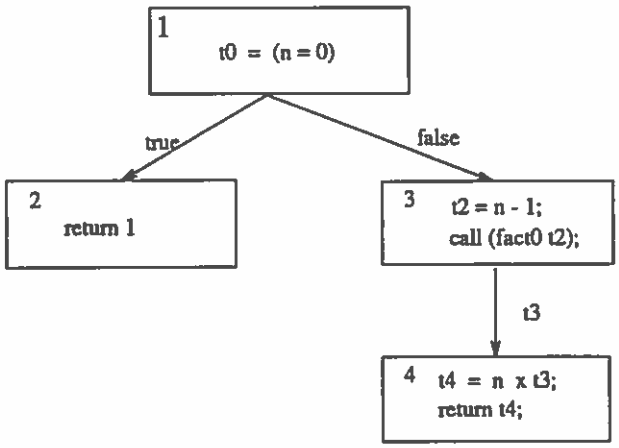

Figure 27: The Flow Graph of Factorial Function

`n*t3` cannot be compiled in the same C function because the control returns to the interpretive loop when a call is encountered. Before a function call its continuation is pushed onto the stack. Values passed from one block to another can be passed through registers or on the stack. A single value is always passed in argument register 0. Multiple values across blocks are passed on the stack.

Compiling a basic block as a C function involves three steps. We declare C variables for the live variables at the entry of the block, for the values passed from a previous block, and for the variables defined in the block. The values passed to a block are restored either from register 0 or from the stack, and the live variables are popped off the stack. The appropriate C code is generated for each variable definition in the block.

The C code generated by the compiler for the function `fact` is given in Figure 28.

In the factorial example, blocks 1, 2, and 3 of Figure 27 are compiled in the function `fact0`. Block 4 is compiled as the function `cont_1`. Recall that returning

```
void fact0()
{
 int t0;
 t0 = (r[0] == 0);          /* r[0] holds n */
 if (t0)
    {
     move(1,r[0]);          /* result returned in r[0] */
     call(exec_return);
    }
 else
    {
     int t2;
     t2 = r[0] - 1;
     push(r[0]);            /* save the live variable n */
     push(cont_1);          /* save the continuation */
     move(t2,r[0]);
     call(fact0);
    }
}


void cont_1()
{
 int t3,t4;
 t3 = r[0];      /* value passed to cont_1 moved to t3/
 pop(r[0]);      /* restore live variable from the stack */
 t4 = t3*r[0];
 move(t4,r[0]);
 call(exec_return);
}
```

Figure 28: Compiled C Code for Factorial Function.

from a function is compiled as a function call to `exec_return`. By convention, the result is always moved to the argument register `r[0]`. In the `else` part of `fact0` which corresponds to block 3, the register `r[0]`, *i.e.* `n`, and the function pointer `cont_1` which is the return address, are saved on the stack by `push` macros. The recursive call is executed by moving `t2` into `r[0]` and the macro `call`.

Block 4 is the function `cont_1`. The value `t3` passed to the block in `r[0]` is moved into the local C variable `t3`. The live variable `n` is popped off the stack into the register `r[0]`. The multiplication is perfomed and the result is moved to `r[0]` before executing `exec_return`.

Now we illustrate how we compile parallel function calls. Consider the fibonacci function and its intermediate form shown in Figure 29.

The data flow graph of the fibonacci function is shown in Figure 30. Notice that both the recursive calls can be executed in parallel. Parallel execution is achieved by spawning a process for one call and continuing the execution of the second call. The continuation of each call is a block which synchronizes the calls. After join synchronization, the values `t6` and `t7` are assumed to be available on the stack at a known offset. Blocks 1, 2, and 3 (see Figure 30) comprise the C function `fib0`. Block 4 and 5 are compiled as two separate C functions `join` and `cont_1` respectively. The C code for fibonacci is shown in Figure 31. The C macro `fork` initializes a counter on the stack. Processes synchronize by decrementing the counter. After join synchronization, one of the processes executes `cont_1`.

In the above code we assume that calls that can be executed in parallel are always executed in parallel. Sometimes it is worthwhile to execute two parallel function calls sequentially. The data flow graph in Figure 32 achieves the sequentialization of

```
fib n = if (n=0) or (n = 1) then 1
        else fib(n-1) + fib(n-2)
        endif;

/* the intermediate form */

fib0 n = let t0 = (n=0);
             t1 = (n=1);
             t2 = t0 or t1;
             t3 = if t2 then
                     let
                     in
                        1
                     end
                  else
                     let t4 = n-1;
                         t5 = n-2;
                         t6 = fib t4;
                         t7 = fib t5;
                         t8 = t6 + t7;
                     in
                        t8
                     end
          in
             t3
          end
```

Figure 29: Fibonacci Function and its Intermediate Form

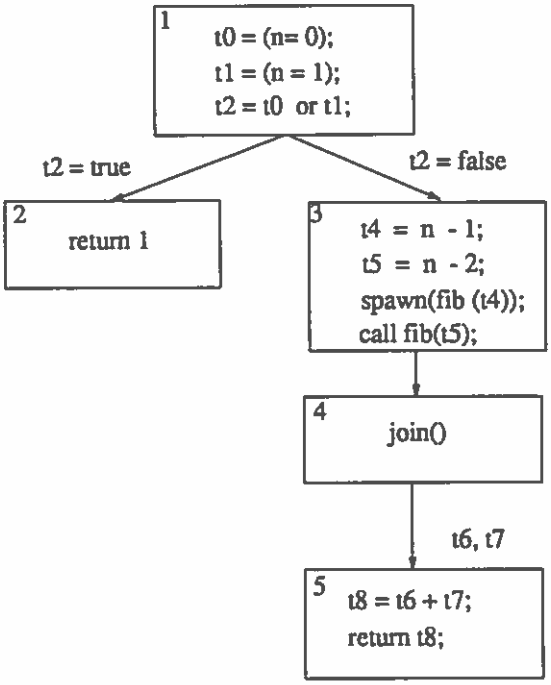

Figure 30: The Flow Graph of Fibonacci Function

```
void fib0()
{
 int t0,t1,t2;
 t0 = (r[0] = 0);
 t1 = (r[0] = 1);
 t2 = t0 || t2;
 if (t2)
   {
    r[0] = 1;
    call(exec_return);
   }
 else
   {
    int t4,t5;
    t4 = r[0] - 1;
    t5 = r[0] - 2;
    fork(2);            /* initialize a synch counter on the stack */
    push(join);         /* push the join continuation on the stack */
    spawn(fib0,t4);
    move(t5,r[0]);
    call(fib0);    /* execute the second call on the same processor */
   }
}

void join()
 {
  save_result();     /* save the result on the stack */
  synchronize();     /* parallel calls synchronize */
  call(cont_1);
 }

void cont_1()
{
 int t6,t7,t8;
 pop(t6);
 pop(t7);
 t8 = t6 + t7;
 move(t8,r[0]);
 call(exec_return);
}
```

Figure 31: Compiled C Code for Fibonacci Function

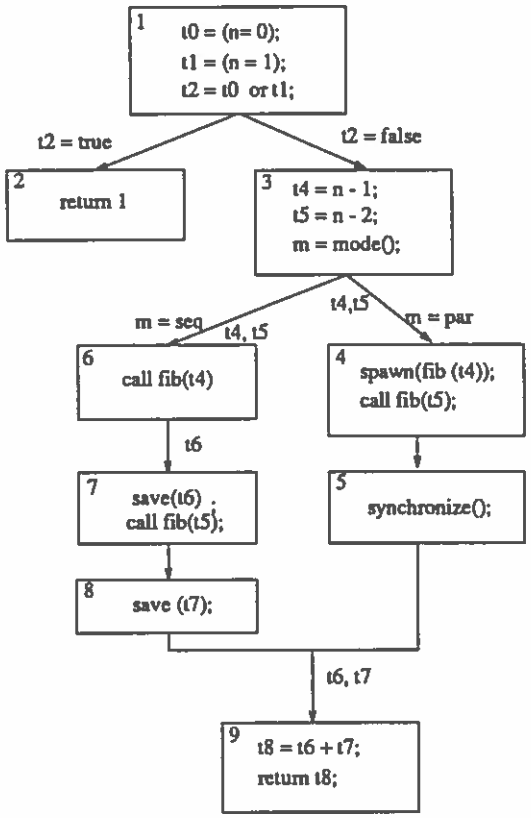

Figure 32: The Flow Graph of Fibonacci Function with Sequential/Parallel Evaluation

parallel calls based on the value of a flag `m`. This flag can be monitored by the runtime system.

4.3.4 The Code Generation Algorithm

The input to the code generator is the intermediate form of the source program with the dependence graph for each `let` expression. The output is a C program where every source function is compiled into a set of parameterless C functions.

The dependence graph of a `let` expression is converted into an ordered list of *blocks* by a function *make_let_blocks*. A block is a set of variable bindings of the form $t_i = e_i$ where the entry to a block implies execution of every variable binding in the block. It does not imply that every subexpression of e_i is executed because e_i could be a conditional expression. In this sense, our basic block is different from the definition of [3]. A `let` expression is converted to a *let_block* $\langle\{b_1, \ldots, b_n\}, r\rangle$ where b_i is a basic block and r is the result of the `let` expression. Two adjacent blocks b_i and b_{i+1} can belong to the C function if b_i does not contain any function calls or `if` expressions. The function *make_let_blocks* ensures that a binding consisting of a function call or a conditional and its successors do not occur in the same block.

We attempt to compile a *let_block* into one C function. A *let_block*, which is a unit of compilation is a tuple

$$\langle name, \langle blocks, r\rangle, lives, vars, cont\rangle$$

The corresponding C function is named by *name*; *vars* are the names of values passed to the *let block*; r is the result of the `let` expression; *lives* is the set of live variables of the block excluding *vars*; and *cont* comprising a name and a set of live variables represents a successor block to which control is transferred after the execution of the *let_block*.

Another unit of compilation is a join continuation, which is of the form

$\langle name, i, cont \rangle$

A join continuation is executed by the processes that are spawned as siblings. Each sibling synchronizes by executing the join continuation. After synchronization, one of the siblings continues to execute *cont*. The integer i represents the space used to pass arguments to the processes. This space is deallocated after join synchronization.

We use a datatype called `FunQ` that is a queue of units which are either *let_blocks* or *join_continuations*. The basic code generation algorithm for compiling a source function is

```
compile_function(f,e) =
    enqueue(FunQ,⟨f,make_basic_blocks(e), ∅, ∅, ⟨∅,exec_return⟩rangle);
    while FunQ not empty
        {
          c = dequeue(FunQ);
          compile_unit(c);
        }
```

The function *compile_function* converts the body of a source function into a list of blocks and adds it to `FunQ`, which is a queue of units to be compiled. A unit of compilation is removed from `FunQ` and is compiled using the function *compile_unit* whose definition is given below.

```
compile_unit(c) =
    if isjoin(c) then
      compile_join(c)
    else
      compile_let(c)
```

In all the functions that follow we assume we have a function *emit_code* that emits the C code. The emitted C code is shown in typewriter font. The function *compile_join* is

The function *compile_join* which compilation a join continuation is described below.

```
compile_join(⟨name,i, ⟨_,rest⟩ ⟩) =
    emit_code(void name (){);
    emit_code(save_result(););
    emit_code(join(););
    emit_code(dealloc_heap(););
    emit_code(call(rest);})
```

The function *compile_let* compiles a *let_block* as defined below.

```
compile_let(⟨name,lb,lives,vars, cont⟩) =
    emit_code(void name(){);
    declare(lives);
    declare(vars);
    restore_vars(vars);
    restore_lives(lives);
    compile_blocks(lb,cont);
    emit_code(};);
```

The function *declare* declares a list of variables as C variables. The function *restore_vars* restores the values passed to the let block into appropriate C variables. The function *restore_lives* restores the live variables from the stack into appropriate C variables. The body of the *let_block* is compiled by the function *compile_blocks* whose definition is

```
compile_blocks(⟨[b1,...,bn],r⟩,cont) =
    if n = 0 then
      emit_code(move(r,r[0]);)
    else if n = 1 then
          {
            compile_block(b1);
            if isprimop(last(b1)) then emit_code(move(r,r[0]););
          }
        else if isprimop(last(b1)) then
              {
                compile_block(b1,cont);
                compile_blocks(⟨[b2,...,bn],r⟩,cont)
              }
            else
              let new_fun = gen_name();
                  v = var(last(b1));
                  new_lives = fv_blocks([b2,...,bn],{v});
                  new_cont =⟨new_lives,new_fun ⟩;
              in
                  enqueue(FunQ, ⟨new_name,⟨[b2,...,bn],r⟩, new_lives,v,cont⟩);
                  compile_block(b1,new_cont)
              end
```

The function *compile_blocks* compiles the body $\langle[b_1,\ldots,b_n],r\rangle$ of a *let_block* whose continuation is *cont*. It checks if two blocks can be compiled in the same C function. The function *last* gives the last binding in a block. If the last binding of b_1 is a primitive operation, then the block is compiled using *compile_block* and the remaining blocks are compiled in the same function recursively. If the last operation of b_1 is a function call or a conditional, then the rest of the blocks has to be compiled as a separate function. This is done by generating a new name for the remaining blocks, and enquequing it for compilation as a new *let_block*. Its continuation is *cont*. The name of the temporary variable in the last binding (obtained by the function *var*) is the name of the value that will be passed to the new *let_block*. Its live variables are computed by the function *fv_block*.

The function *compile_block* compiles a *basic block* and is defined below.

```
compile_block(b,cont) =
  if calls(b) then
    compile_calls(b,cont)
  else compile_noncalls(b,cont)
```

There are two types of basic blocks, one consisting of only function calls and the one consisting of primitive operations or a conditional. The function *make_let_blocks* ensures that a conditional can only occur as the last binding in a basic block. Parallel function calls can be compiled as one basic block because all but the last call only involve creation of processes. The last call is implemented as a function call. However, in a sequential execution, each function call has to be compiled as a separate block. The definition of *compile_calls* is

```
compile_calls([t1=e1,...,tn=en],cont) =
  if n = 1 then compile_call(t1,e1,cont)
  else
      {
        enqueue(FunQ,⟨new_name,heap_size([t1=e1,...,tn=en]),cont⟩);
        compile_spawn_calls(b,⟨∅,new_name⟩);
      }
```

In the case of parallel function calls, a join continuation is enqueued in FunQ, and the calls are spawned using *compile_spawn_calls*. The definition of *compile_spawn_calls* is

```
compile_spawn_calls([t1=e1,...,tn=en],cont) =
  if n = 1 then
    compile_call(t1,e1,cont)
  else {
        compile_spawn(t1,e1,cont);
        compile_spawn_calls([t2=e2,...,tn=en],cont);
      }
```

The function *compile_spawn* generates the C code to spawn a process. Spawning a process involves allocating space on the argument heap which is another stack used for passing arguments to the newly created process, moving the parameters of the call to the heap, creating a process and making it available for scheduling. The function *compile_call* generates code to move the actual parameters of the call into

the argument registers; and the macro `call` to call the function. Before calling the function, the continuation, if it is not `exec_return` (which indicates a tail-recursive call), and its live variables are saved on the stack.

The function to *compile_noncalls* is defined below.

```
compile_non-calls([t1=e1,...,tn=en],cont) =
   let helper([t1=e1,...,tn=en]) =
      if n = 1 then
        if isif(e1) then compile_if(t1,e1,cont)
        else
          compile_primop(t1,e1)
      else {
             compile_prim(t1,e1);
             helper([t2=e2,...,tn=en);
           }
    in
      declare([t1,...,tn]);
      helper([t1=e1,...,tn=en]);
   end
```

A block of non-calls is compiled using *compile_primop* and *compile_if*. The C variables corresponding to the `let` variables of the block are declared by the function *declare*. For a binding t_i=e_i where e_i is use of a primitive operator, *compile_primop* generates the C assignment statement $t_i = e_i$. A conditional expression is compiled by *compile_if* which compiles the branches of the conditional, which are `let` expressions, using *compile_blocks* as defined below.

```
compile_if(if e0 then e1 else e2,cont) =
   emit_code(if (e0));
   emit_code({);
   emit_code(compile_blocks(make_let_blocks(e1),cont));
   emit_code(});
   emit_code({);
   emit_code(compile_blocks(make_let_blocks (e2),cont));
   emit_code(})
```

4.4 Update Analysis with Separate Compilation

Most modern programming languages provide module systems [52, 53, 81]. The programmer can decompose a program into several separate programs called modules: these modules usually correspond to the logical decomposition of the problem. Each module can be compiled separately without having any knowledge of the implementation of the other modules. A module can call functions defined in some other module. Such a communication among modules is facilitated by type signatures [52] or import and export declarations [81]. During the linking phase external references are resolved.

Our update analysis algorithm, a global dataflow algorithm, requires the entire source program for analysis. In this section we show how one can support update analysis with separate compilation. We do not know of any global dataflow analyses other than register allocation [79] that work with separate compilation.

For the discussion we assume that imported and exported functions of a module are declared by `import` and `export` declarations. While analyzing a module, we do not have any information about the imported functions. Depending on how we handle these functions, we get different scenarios for separate compilation.

4.4.1 Simple Separate Compilation

For update analysis, we can treat imported functions as opaque functions that do not propagate, select or update any argument. This implies that an explicit copying of the arguments to external functions is necessary. But copying is needed only across module boundaries. Such copying can further be minimized by explicit runtime tests, an approach currently taken by SISAL for arrays crossing function boundaries. This

method would work with existing linkers.

4.4.2 Separate Compilation with Smart Linking

We can avoid copying accross module boundaries at the expense of extra work at link time. Since the types of the imported functions are known to the module, we can make a conservative estimate of their propagation behavior. An imported function can propagate only those arguments whose types are the same as that of the result of the function. We also assume that an unknown function selects all its array arguments.

With this conservative estimate, we perform update analysis of each module. The object code of a module includes the information about propagation, aliasing, select-updates, and reference count analyses of each function defined in the module. It also includes aliasing of an external function caused by each external call site, and the reference count of the actual parameter of each external call site.

At link time, we can identify if each external call site introduces any additional aliasing of arguments of the function. If no aliasing is introduced, we know that the arguments need not be copied. For each call site, we can avoid copying an actual parameter if its reference count is 1 or if it is $\top$ but the function only reads that argument (note that this information is available at link time for each module). This scheme is likely to work well if mutually recursive functions do not live in multiple modules. We believe that mutually recursive functions are typically defined entirely within one module. Again, the modules can be compiled in any order but we need a sophisticated linker. For a more accurate update analysis, one can extend the module interface by allowing the user to specify the propagation behavior and selects-updates behavior of imported functions. Similar to signature matching, at link time

the propagation and selects-updates information specified by the user can be matched against the information computed by the analyzer and reported to the user in case of differences.

4.4.3 Dependency Based Separate Compilation

In the previous case we had conservatively assumed that an imported function propagates or reads all possible arguments. Can we do better? Consider determining the order of evaluation of two expressions `f(x)` and `g(x)`. Suppose `f` is an imported function that updates `x` and `g` is a function defined in the module that reads `x`. Since we assume that `f` reads `x`, our ordering algorithm chooses an arbitrary ordering for these two expressions. In the case of `f(x)` evaluated before `g(x)`, copying of `x` before the call to `f` cannot be avoided by the linker described previously. We could modify our ordering algorithm to evaluate calls to known functions first when the known function has no updates. But if both `f` and `g` are imported functions, the ordering algorithm has no way of fixing an evaluation order between `f` and `g`.

One might argue that if we had conservatively assumed that external functions update all their possible array arguments, then the ordering algorithm would have found the right order that enables the linker to avoid copying `x`. But the problem is that with this conservative estimate an update in the module may become non-destructive. This can easily be seen by considering `f` as the reader of `x` and `g(x)` the sole modifier of `x`. Again the ordering among these two expressions has to be arbitrary and in one case the update in the module becomes non-destructive. We chose a conservative estimate that does not interfere with the destructive updating of a known update.

If we compile the modules respecting dependencies, then the flow information

computed in one module can be used in compiling the subsequent modules that depend on it. Of course this assumes that there are no cyclic dependencies among modules. We still need the smart linking technique described in the previous section. Compiling modules in a dependency based order is similar to compiling several C files using the `make` facility. The benefits of each of these approaches need to be investigated by designing and implementing a complete functional language with modules, an important area for further research.

4.5 The Runtime System

Functional programming languages are implicitly parallel. The user is relieved of details such as creation of parallel processes, resource allocation such as process scheduling and storage management, and process synchronization. These tasks are handled by the runtime system. We have designed and implemented a runtime system for our language on a Sequent Symmetry S81, a bus-based shared memory multiprocessor.

4.5.1 Model of the Runtime System

The runtime system is based on the worker crew model [24]. A process is a schedulable unit of work. The granularity of a process is a function call. Each process has its own stack and argument heap. The stack is for sequential execution. The argument heap is used to store arguments to a new process. The argument heap of a process is used in a stack-like fashion. A worker allocates a stack and a heap for a process. In addition, there is a global data heap for allocating data objects such as arrays. A process can be classified as *new* or *continuing*. A continuing process already has a stack allocated on some processor. In our model, all processes that

have stacks allocated on some processor must be executed on that processor. Each worker has a pool of continuing processes that are waiting for operands.

Each worker in our model has two queues, a local queue for continuing processes ready for scheduling, and a process queue for new processes. Each worker looks for a continuing process in its continuation queue and executes it. If this queue is empty, it finds a new process and executes it. The scheduling policy of the runtime system determines how a worker acquires a new process. Our current sceduling policy is described in a later section. Each worker executes the following algorithm.

```
find_work() =
   if local_q not empty
     { c = get_cont();
       execute_cont(c);
     }
   else let p = get_newprocess();
        in
            execute_process(p)
        end
```

The function *execute_cont* extracts the continuation of the process from the pool of continuations and invokes it. The function *execute_process* allocates a new stack and heap for the new process and executes it. A new process is described by a C structure

```
struct Process {
 void (*code)();
 int arity;
 stackelement *frameptr;
 struct cont_descriptor c;
 }
```

Function pointer `code` holds a pointer to the called function; `arity` is its arity; `frameptr` is a pointer to the argument heap where the arguments of the process are stored; and `c` is a descriptor of the continuation of the process. A continuation descriptor is another C structure

```
struct cont_descriptor {
int offset;
int c_index;
int pe_no;
 }
```

It has a field pe_no that describes the worker to which the continuation belongs. c_index is the index for accessing the continuation from the pool of waiting continuations. The field offset is the offset from the stack at which the result of the process is stored.

The result of a computation is passed to its continuation by the C structure

```
struct  Result {
    stackelement value;
    int offset;
    int c_index;
   }
```

The variable value holds the value computed by a process; offset is the stack offset where the value is stored; and c_index is the index of the continuation in the pool of waiting continuations.

To execute a new process, a worker allocates a new stack and an argument heap on its physical stack and heap. The stack of a process is local to the worker whereas the argument heap is shared. The continuation descriptor of the process is pushed onto the stack. A pointer to the function exec_last_cont is also pushed onto the stack. Now the worker executes the function pointed to by the code field of the process. After its execution, the result of the process has to be passed to its continuation. This is accomplished by exec_last_cont. It pops the continuation descriptor off the stack and puts the result in the local continuation queue of the appropriate worker. It then deallocates the stack and heap of the current process.

4.5.2 Storage Management

When a worker gets a new process, it has to allocate a new stack and heap. In sequential execution, heap and stack management is much simpler. Typically, the stack grows downward and the heap grows upward thus utilizing the available space effectively. When multiple threads of execution exist concurrently, this simple memory model does not work.

In languages like Scheme and Standard ML of New Jersey, which have operations such as `call/cc`, stack based execution mandates that the captured stack be copied to the heap when `call/cc` instruction is executed [23]. Similarly, the stack has to be restored when a captured continuation is invoked. Moreover, in the presence of higher-order functions, variables have indefinite life times and a stack-like allocation of variable is not possible. Some implementations [4, 5] abandon the stack completely and represent continuations as a linked list of activation records on the heap, relying on the garbage collector for the reclamation of unused frames. Mixed strategies of representing continuations have been suggested in the literature [22].

Each worker has a single physical stack and argument heap. The stack is local to the worker whereas argument heap is shared and accessible to the other workers. One simple way to manage multiple stacks for concurrent execution is to divide the physical stack space into a fixed number of smaller stacks, each stack allocated to a new process. The entire stack of a processor is not available for a process that is executing sequentially. Our design goal was to retain the stack based execution (as we don't allow higher-order functions) but to have a flexible mechanism where each process potentially gets the entire available processor stack for execution. We achieve this by implementing the process stack as a doubly linked list of stack records, the

strategy proposed by [43]. Each stack record points to an ordinary stack. Execution within a stack is the usual stack execution. When a stack overflow occurs, the overflow handler moves to the stack of the next stack record. Stack underflow is handled similarly. One gets the advantage of stack based execution but some price is paid in moving along the linked list. The argument heap of a process is also implemented as a linked list of heap records. The stack record is described by the C structure

```
struct StackRecord {
 stackelement *stackpointer;
 stackelement *stacklimit;
 stackelement *stackbase;
 struct StackRecord *next, *prev;
  }
```

The stack of a process is shown in Figure 33.

When an executing process cannot be run any further because its operands are not yet computed, the worker process chooses another ready process by executing the basic algorithm *find_work*. If the entire available physical stack of the worker was allocated to the previous process, then we need to seal its stack in order to allocate a stack for a new process. Sealing a stack involves checking if `stack_limit` of the current stack record is the physical stack limit of the worker. If that is so, then `stack_limit` is changed to a new value, appropriately modifying the available physical stack of the worker for subsequent allocation. The argument heap is also sealed in the same fashion. In order to avoid frequent stack overflows it is desirable that each stack of a stack record have at least certain minimum size. The appropriate size has to be chosen experimentally.

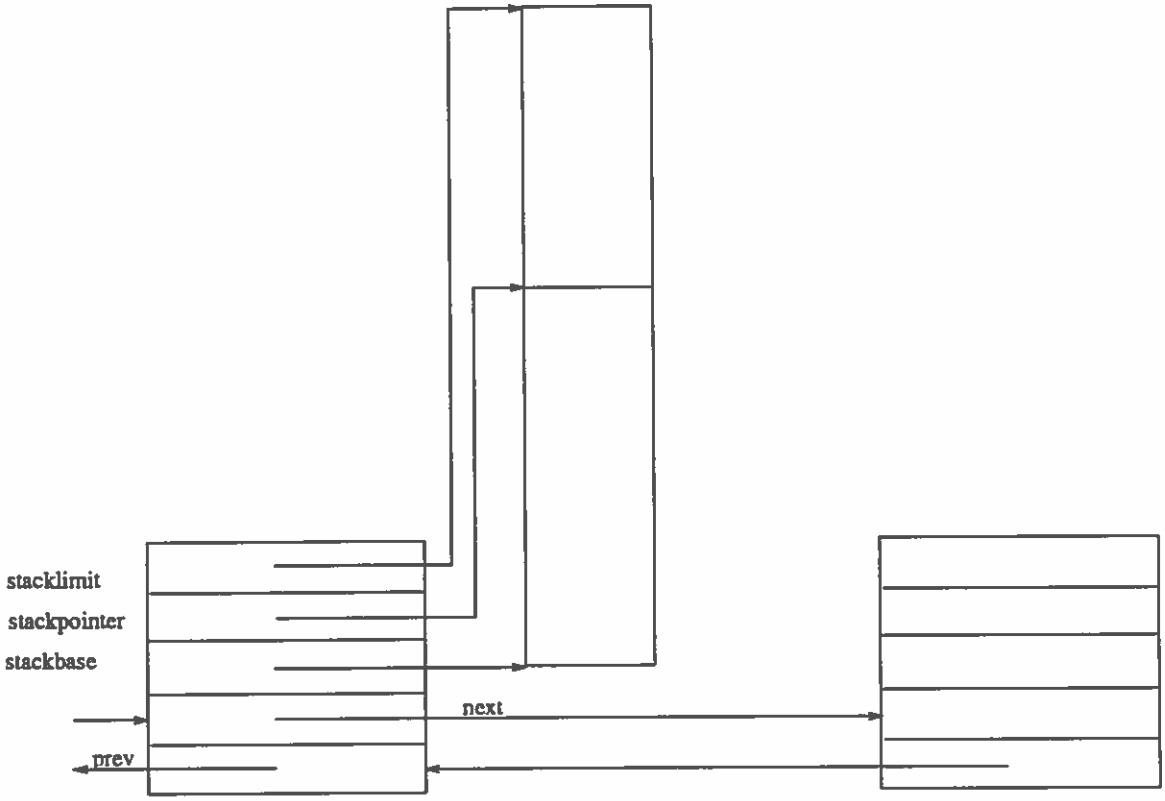

Figure 33: A Process Stack

4.5.3 Join Synchronization

Consider the parallel evaluation of function calls $t_1 = e_1, \ldots, t_n = e_n$. Space is allocated on the stack of the current process for the results of $e_1, \ldots, e_n$. The `fork` instruction initialises a counter to n on the stack. The join continuation of these expressions is pushed on the stack. The continuation of the parallel processes which is the current stack,heap pointers, and a pointer to the stack and heap record is saved in the pool of waiting continuations. The calls $e_1, \ldots, e_{n-1}$ are spawned as new processes. Each process gets a unique offset (from 1 to n) in its continuation descriptor. The expression e_n is implemented as a function call on the same processor.

When an expression e_i is evaluated completely, the result is returned to the Worker which had spawned e_i. The stack and the heap of the process are extracted from the pool using the `c_index` of the process. Recall that the first instruction in a join continuation is `save_result()`. The result of e_i is saved at $offset_i$ from the current stack pointer. The synchronization counter is decremented by 1. If the counter becomes 0, it means that the current process is the last process to synchronize. This process frees the slot used in the pool of waiting continuations and continues executing the rest of the code. If the counter is greater than 0, it implies that the current process cannot be executed any further. The worker relinquishes the current process by sealing its stack and heap if necessary and looks for new work by executing *find_work*. This mechanism is the same as described for multithreaded architectures [56].

4.5.4 Controlling Parallelism

One of the serious problems with implicitly parallel languages is that they tend to express too much parallelism. If the grain of parallelism is too fine then the overheads of spawning several fine processes, each one doing very little work, outweigh the gains of parallel execution. Moreover, too many parallel processes would also swamp the machine resources thereby precluding the execution of programs on larger data sizes. Our runtime system supports automatic constraining of parallelism with the following features.

1. A worker executes a process sequentially if its process queue is full.

2. A process executing sequentially can change to parallel execution upon request from starving workers.

3. An inherently sequential process does not pay too much overhead for execution on a parallel machine.

To constrain parallelism, each processor is equipped with a flag `mode` which determines whether the process should execute sequentially, or spawn more subprocesses for parallel execution. This flag can be set under various conditions.

A process does not spawn more processes if the process queue of the worker is full. Suppose that the maximum size of the process queue is q; then the maximum number of processes that exist simultaneously in the system is at most pq where p is the number of workers. When a process spawns parallel processes, it sets the flag to sequential mode if its process queue is full or its pool of waiting continuations is full.

In our current implementation we have chosen the size of the process queue and the pool of continuations as $2p$.

It is desirable that a sequentially executing process change to parallel execution upon demand for work from other starving workers. This is achieved by allowing a worker to set the `mode` flag of other workers. Function calls that can be executed in parallel are actually compiled for both sequential and parallel execution as discussed previously. The actual code executed at runtime is based on the `mode` flag of the worker. Thus spawning parallel calls incurs an overhead of checking a flag. Since the mode flags are checked only in the case of parallel function calls, inherently sequential code does not incur this overhead. There is some overhead for sequential execution, however, because a stack is now a list of stack records.

4.5.5 Process Scheduling

Our current scheduling policy is based on a virtual ring structure of the worker processes. Each worker has a left and right neighbor. Typically, a worker acquires work from its own process queue or from its left or right neighbour. This strategy is scalable but only achieves local load balancing. For global load balancing we use randomization. When a worker does not find work in either of its neighbors, it tries to pick work from the process queue of a random worker. If a worker does not find any work, then it sets the `mode` flag of its left neighbour to `par` asking its neighbour to spawn more work if possible and executes *find_work* again. Initially every worker starts with the `mode` flag set for parallel execution.

4.5.6 Implementation on Sequent Symmetry

Sequent Symmetry is a bus based shared memory multiprocessor. The S81 configuration has 20 processors that access global memory via shared bus. Each processor has a 128K instruction and data cache. Sequent Symmetry runs the Dynix 3.1 operating system, which is derived from the Berkeley 4.2 version of Unix. The Sequent Symmetry supports high level programming languages such as C, Fortran, and Pascal.

Parallel programming on Sequent is supported by a parallel programming library that provides calls for process creation, synchronization, declaring shared variables, and locking shared memory.

The runtime system is written in C using the parallel programming library. Each worker of the runtime system is implemented as a Dynix process. The continuation and process queues are implemented as shared circular queues. The stack space, argument heap space, and the global data heap are allocated as shared data.

No locking is required by a worker when acquiring a process from its local continuation queue or putting work in its process queue. The reason is that only one worker, the owner of these queues, ever acquires work from a local queue or puts work in its process queue. However, locking is required when putting work in a local continuation queue of another worker or getting work from a process queue. We expect the locking overhead to be negligible because each worker has only 2 neighbors, and a few more workers may be accessing the process queue of that worker because of randomization in our scheduling policy. No locking is required in any other part of the runtime system.

4.6 Preliminary Performance Results

We present the performance of the system on two benchmarks, the recursive fibonacci function and parallel vector addition. Table 4 presents the execution time of these two programs as a function of input arguments and the number of processors. The argument to the vector adddition function is the size of the vector. In Table 5, we compare the performance of the compiled program on one processor to the execution of a sequential C program. In all these examples, peak performance is achieved with 8 processors. For the vector addition example, the implicitly parallel vector addition is approximately 10 times slower than the corresponding C program. The tail recursive vector addition is approximately three times slower than the C program. We also studied the performance of vector addition with explicit sequentialization where the tail-recursive sequential vector addition is called when the vector size is below a particular threshold. In Table 4 the last row is the performance of vector addition when the threshold is 256. The threshold has to be determined by trial. This example shows that explicit sequentialization may be essential for achieving good performance. We are currently working on performing a post-pass of the assembly code to eliminate the overheads of C function calls.

If C compilers could be relied upon to handle tail-recursion properly, then the code generated by the functional language compiler would be just as fast as C in the sequential case, and would have the advantage of automatic parallelization on multiprocessors.

Table 4: Execution Time of Two Programs (in secs)

Programs	*Processors*				
	1	2	4	8	16
fib(25)	6.96	4.3	1.89	1.2	2.7
fib(30)	77.16	40.17	22.96	10.75	13.18
vadd(10000)	0.84	0.45	0.29	0.54	2.5
vadd(100000)	8.42	4.58	3.20	1.87	2.79
vadd(500000)	41.9	22.7	11.9	6.37	10.9
seqvec(500000,256)	6.32	3.19	1.67	1.13	7.5

Table 5: A Comparison with Sequential C Execution(in secs)

Programs	*unoptimized C*	*optimized C*	*ours*
fib(25)	1.76	1.32	6.96
fib(30)	19.56	14.68	77.16
vadd(500000)	2.76	2.68	41.9
seqvadd(500000,256)	2.76	2.68	6.32

CHAPTER V

THE NON-FLAT ARRAY UPDATE PROBLEM

5.1 Introduction

In Chapter II, we presented an efficient update analysis algorithm for flat arrays and sequential evaluation. We generalized our algorithm for parallel evaluation in Chapter III. In this chapter we consider another generalization : generalization to non-flat arrays. We discuss some of the problems while considering non-flat or nested arrays and present an extension of our algorithm described in Chapter II. Although our discussion in this chapter assumes sequential evaluation, one can consider parallel evaluation by using partition and combining operations on non-flat arrays analogously.

5.1.1 Non-flat Aggregates

No previous work on the aggregate update problem has considered non-flat aggregates, in which an aggregate x has an aggregate y as a proper subcomponent that can be extracted from x while continuing to share structure with x. Flat aggregates are general enough to include all Pascal-like types except pointers, references, and procedures. In this chapter we consider non-flat homogeneous arrays, but the algorithm we present here applies to all non-recursive Pascal-like types except procedures.

The restriction to non-recursive types is needed to ensure finiteness of our abstract domain of aggregates.

The selection and update operations on arrays are defined as follows: `sel(x,i)`

returns element `i` of the array `x`. `upd(x,i,y)` returns an array that is like `x` except that it has `y` at index `i`. The goal of update analysis is to implement updates destructively (by side-effect) instead of by creating a new array. As an illustration of update analysis, and to show how it is made more difficult by non-flat aggregates, consider the following program fragment:

```
f (x,y,i) = if i < 0 then x else upd (x,i,y);
g (x,y,i) = sel (upd (sel (f (x,y,i), i), i, i), 2*i) + sel (y,i)
```

Suppose `f` is called only from g, and for all calls to g the arguments to g are thoroughly dead when g returns. Then the update in the body of `f` can be performed destructively, because `f` has the only reference to `x`. The update in the body of g is harder to analyze. Its first argument, the value of `sel(f(x,y,i),i)`, may be the value of `y`, which is needed to compute `sel(y,i)`. Our update analysis algorithm will therefore arrange to compute `sel(y,i)` before the update and will perform the update destructively. To deduce that `y` is a possible result of `sel(f(x,y,i),i)`, however, the update analysis must keep track of which aggregates may share structure with other aggregates. This is more complex than the simple aliasing analysis that suffices for flat aggregates [63].

5.1.2 Copying To Avoid Copying

In the following example, neither update can be performed destructively:

```
f z w j  =  h (upd (sel (z, j), 0, 1), w)

g x y i  =  f (upd (x, i, y), x, i+1)
```

The update in g cannot be destructive because x is live. The update in f cannot be destructive because its first argument shares structure with w, which is live. Note that z and w are not aliases, and neither is a component of the other, but the fact that they share common subcomponents is enough to prevent the update in f from being performed by side effect.

But sharing is not semantically relevant in a functional language, so the argument in the preceding paragraph is based on an assumption about the implementation of the selection operation. If the selection operation were to return a copy of the selected element, then the update in f could be done destructively after all. A copying selection operation would never be slower than using flat arrays, and it would at times be faster because it would enable some updates to be performed in constant time.

We thus have an interesting tradeoff: The goal of update analysis is to avoid copying the first argument to the update operation. This goal can often be achieved at the expense of extra copying elsewhere, as in the selection operation. This tradeoff arises in connection with flat aggregates as well [41, 63], but non-flat arrays provide a broader range of options. To evaluate these options we must consider the motivations for non-flat aggregates and their typical patterns of use.

One motivation for non-flat aggregates is their potential for representing sparse matrices. Consider allocating storage for a triangular matrix. The most natural representation is an array of arrays, although one could embed a triangular matrix in a linear array. However the embedding becomes more complicated when one considers block diagonal matrices and other sparse matrices whose entries are themselves matrices. Such matrices arise in finite element methods. Sparse vectors can be represented as a non-flat array. A sparse matrix can then be represented as a sparse vector

of vectors. In Fortran a sparse vector is represented by two arrays: one for storing indices which have non-zero values, and the other for the corresponding values of the vector.

One of the problems with Fortran-77 like languages for sparse matrix computations is that space for *fill-ins* has to be preallocated by the programmer. Thus for each sparse matrix algorithm, the burden of preallocating additional storage, its initialization, and garbage collection is thrust upon the programmer [30, 83]. We believe that a functional language with dynamic storage allocation and efficient implementation of non-flat arrays would be a good choice for expressing sparse matrix algorithms.

Another motivation for non-flat aggregates is convenience. For example, it is easy to select a row from a two-dimensional non-flat array in order to compute its norm. Since syntactic sugar can be used to provide the same convenience for flat arrays, however, it seems that the real motivation for non-flat aggregates is efficiency. The programmer knows, or at least thinks, that it is faster to select a row from a two-dimensional non-flat array than to construct the corresponding row from a two-dimensional flat array. The compiler should trust the programmer's judgment in this. Nonetheless there are occasions when extra copying is justified to improve the effectiveness of update analysis. For example, suppose `A` is a two-dimensional non-flat array, and suppose `B` is the result of replacing row `i` by row `j` via the expression

```
upd (A, i, sel (A, j))
```

in a context where `A` is dead. This update can be performed destructively. Unfortu-

nately, it then becomes unlikely that updates such as

```
upd (sel (B, m), n, 0)
```

can be done destructively, because two rows of B share structure. It is easy for the compiler to warn that copying is required when rows of B are updated, but it would be more useful if the compiler were to advise that this copying could be avoided by changing the expression that yields B to

```
upd (A, i, dcopy (sel (A, j)))
```

where dcopy performs a deep copy of its argument.

The compiler is unlikely to have enough information to decide whether it is cheaper to perform a shallow copy of a row of B upon each update to a row, or to perform a deep copy of the third argument to the update operation that creates B. Only the programmer can decide this. We are led therefore to introduce both shallow and deep copy operations, scopy and dcopy, which have the semantics of identity functions but serve also as declarations. These explicit copy operations declare not only the programmer's awareness that copying will be required, but they also declare the places in the program where the programmer believes copying should occur in order to obtain the most efficient results. The compiler can of course omit any unnecessary copying that the programmer has declared, but this is not trivial as is shown by the above examples.

A shallow copy is generally appropriate for the first argument to an update operation, while deep copying is appropriate for the third argument to an update

operation or to eliminate sharing between formal parameters.

With explicit copy declarations and an effective optimizer, it becomes reasonable for the compiler to complain whenever copying is required that has not been anticipated by the programmer. If these complaints are treated as fatal errors, then the optimization we describe in this chapter becomes a limited kind of automatic program verification.

5.1.3 Local Optimizations

To be effective on non-flat aggregates, update analysis must deal specially with nested updates. Consider the Pascal assignment statement

```
A[i,j] := 0
```

for which our corresponding functional notation is

```
upd (A, i, upd (sel (A, i), j, 0))
```

in a context in which `A` is dead. Unfortunately, `A` is live in the context of the inner update, which updates a component of `A`, so it is not obvious that the inner update can be performed destructively. This idiom must be recognized as part of the optimization.

Row interchange is another idiom that is worth recognizing specially, though it is less important than nested updates.

5.2 Overview

In this chapter we present a simplified algorithm that corresponds to an implementation of the update operation in which the third argument is implicitly deep-copied unless the analysis determines that the copying is unnecessary. The general algorithm has the same structure as the simplified algorithm, but the general algorithm uses a more complicated abstract domain to express the potential sharing of structure between aggregates. We did consider the details of a general algorithm that does not assume deep copying the third argument to the update operator but could not find interesting examples where the simpler algorithm fails to avoid copying but the general algorithm suceeds. Therefore we present the simpler algorithm.

The simpler domain of sharing reflects the following invariant, which is made possible by copying the third argument of each update: No proper subcomponents of a single array share any structure with each other.

The algorithm consists of several phases, most of which correspond directly to the phases of our algorithm for flat arrays as described in Chapter II. Propagation analysis determines, for each expression, the arrays that may be the result of the expression. Sharing analysis detects all possible sharing between arrays; this replaces the aliasing analysis of Chapter II. The next phase, copy avoidance, is new: for each update operation, it attempts to prove that the results of the sharing analysis hold without copying the third argument to the update. Selects-and-updates analysis determines, for each expression, the sets of arrays that are selected or updated during evaluation of the expression.

The algorithm then constructs the data dependency graph and attempts to choose an order of evaluation in which, for each array, all selection operations precede

all update operations. Abstract reference count analysis then determines, for each update, whether the array being updated is live or dead. If the array is dead, then it is safe to perform the update destructively. Otherwise the result of the update must be a shallow copy of the array being updated, except for one change at the index of the update.

As in Chapter II, we assume that the optimization operates on an intermediate form in which all variables are distinct and a temporary name has been generated for each non-trivial expression. The expression associated with a temporary variable is given by the function *expr_of*. For simplicity we do not allow `let`-expressions in the source language, even though the analysis can easily be extended to `let` expressions as described in Chapter III.

5.3 Propagation Analysis

Given an expression, propagation analysis determines the arrays returned by the expression. Recall that our language has no recursive array types.

> *Definition*: The dimension d of an array of type `array of` ... B, where B is a scalar type (or at least a type that involves no array types), is the number of `array of` constructors that occur in the type. □

5.3.1 Abstract Domain of Arrays

For an array variable x of dimension d, $(x,0)$, $(x,1)$,...,(x,d) represent the arrays at levels 0, ..., d where $(x,0)$ is the entire array and (x,d) is a non-array value. The ordering between two elements (x,i) and (y,j) is defined as

$$(x,i) \sqsubseteq (y,j) \text{ iff } (x = y) \text{ and } j \leq i$$

Intuitively the ordering among the abstract values means that if we have a pointer to an array, then we also have pointers to all its sub arrays.

A *named* array is an array that is bound to a formal parameter of a function. An *anonymous* array is an array created by an update operation. The domain D of abstract arrays is the disjoint sum of the named arrays A and the anonymous arrays U (Figure 34).

5.3.2 Abstraction of Array Operators

The function $sel^{\#}$ defined on D maps an element $(x,i) \in A$ to $(x,i+1)$ and an element $u((x,i)) \in U$ to $(x,i+1)$. The function $upd^{\#}$ maps an element $(x,i) \in A$ to $u((x,i))$ and an element of U to itself. These functions reflect our simplifying assumption that the update operation always performs a deep copy of its third argument and shallow copy of its first argument.

The functions *Sel* and *Upd* are extensions of $sel^{\#}$ and $upd^{\#}$ to the Hoare power domain of D [65]:

$$\begin{aligned} Sel &: \mathcal{P}_H(D) \to \mathcal{P}_H(D) \\ Sel\ S &= \bigsqcup\{sel^{\#}(x) \mid x \in S\} \end{aligned}$$

$$\begin{aligned} Upd &: \mathcal{P}_H(D) \to \mathcal{P}_H(D) \\ Upd\ X &= \bigsqcup\{upd^{\#}(x) \mid x \in X\} \end{aligned}$$

The propagation function $\mathcal{H}$ computes the set of abstract arrays returned by an expression (Figure 35).

V	=	Program Variables	
F	=	Sser Defined Function Names	
N	=	$\{0, 1, 2, \ldots, d\}$	Domain of Levels (ordered by $\geq$)
A	=	$V \times N$	Named Aggregates
U	=	$\{u(x) \mid x \in A\}$	Anonymous Aggregates
D	=	$A + U$	Domain of Aggregates
$VEnv$	=	$V \to \mathcal{P}_H(D)$	Variable Environments
$FEnv$	=	$F \to \mathcal{P}_H(D)^n \to \mathcal{P}_H(D)$	Propagation Function Environments

Figure 34: Domains for Propagation Analysis

$$
\begin{array}{lcl}
\mathcal{H}[\![\mathrm{c}]\!]\sigma\rho & = & \emptyset \\
\mathcal{H}[\![x]\!]\sigma\rho & = & \sigma[x] \\
\mathcal{H}[\![t_i]\!]\sigma\,\rho & = & \mathcal{H}[\![expr\_of(t_i)]\!]\sigma\,\rho \\
\mathcal{H}[\![op(se_1, \ldots, se_n)]\!]\sigma\rho & = & \emptyset \\
\mathcal{H}[\![\mathtt{sel}(se_1, se_2)]\!]\sigma\rho & = & Sel(\mathcal{H}[\![se_1]\!]\sigma\rho) \\
\mathcal{H}[\![\mathrm{upd}(se_1, se_2, se_3)]\!]\sigma\rho & = & Upd(\mathcal{H}[\![se_1]\!]\sigma\rho) \\
\mathcal{H}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\sigma\rho & = & \mathcal{H}[\![e_1]\!]\sigma\rho \ \sqcup\ \mathcal{H}[\![e_2]\!]\sigma\rho \\
\mathcal{H}[\![f_k(se_1, \ldots, se_n)]\!]\sigma\rho & = & \rho[f_k](\mathcal{H}[\![se_1]\!]\sigma\rho, \ldots, \mathcal{H}[\![se_n]\!]\sigma\rho) \\
\mathcal{H}[\![\ \mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ \ t_i\ \mathtt{end}\ ]\!]\sigma\rho & = & \mathcal{H}[\![t_i]\!]\sigma\,\rho
\end{array}
$$

Figure 35: Propagation Function $\mathcal{H} : IExp \to VEnv \to FEnv \to \mathcal{P}_H(D)$

$$\mathcal{H}_p[\![pr]\!] \ = \ fix(\lambda\sigma.\, \sigma[f_i \mapsto \lambda y_1, \ldots, y_k.\, \mathcal{H}[\![e_i]\!][x_{1_i} \mapsto y_1, \ldots, x_{k_i} \mapsto y_k]\sigma])$$

Figure 36: The Function $\mathcal{H}_p : IProg \to FEnv$

5.3.3 Symbolic Transformations

The abstraction defined for propagation analysis results in a set of recursively defined abstract functions. Even though the array types are nonrecursive, a program can update an array recursively which may result in the infinite application of *Upd* operator. To ensure termination of our symbolic evaluation of these abstract functions, we use a rewrite system to compute normal forms at each iteration:

$$
\begin{aligned}
Upd(Upd(A)) &\Rightarrow Upd(A) \\
Upd(A \sqcup B) &\Rightarrow Upd(A) \sqcup Upd(B) \\
Sel(Upd(A)) &\Rightarrow Sel(A) \\
Sel(A \sqcup B) &\Rightarrow Sel(A) \sqcup Sel(B) \\
(A \sqcup B) \sqcup C &\Rightarrow A \sqcup (B \sqcup C)
\end{aligned}
$$

5.3.4 Complexity of Symbolic Transformations

The first concern about the rewriting system is whether it is Church-Rosser, *i.e.* the order of application of the rules is immaterial in computing normal forms. The above rules can be shown to be Church-Rosser by the Knuth-Bendix completion procedure [49].

To show that the rewriting process terminates, we divide the rules into three groups, the *Upd* rules consisting of the first two rules, the *Sel* rules comprising the next two rules, and the $\sqcup$ rule. Notice that the application of *Sel* rules does not create any redexes for the *Upd* rules. Similarly, the application of the $\sqcup$ rule does not create any redexes for either the *Sel* or the *Upd* rules. This immediately gives us an algorithm for computing normal forms. The simplification algorithm applies the

Upd rules until they are not applicable any more, followed by repeated application *Sel* rules. Lastly, the ⊔ rule is applied until it is not applicable any more.

Notice that the first *Upd* rule always decreases the size of the term, but the second *Upd* rule increases the size of term. The second *Upd* rule can be thought of as pushing an *Upd* symbol from the root of subtree toward its leaves. The size of new term after the removal of all the redexes of the second *Upd* rule is bounded by the number of leaves which is $O(n)$. Now by applying the first *Upd* rule all the *Upd* redexes can be removed.

By the same argument, repeated application the second *Sel* rule, equivalent to pushing a *Sel* symbol toward the leaves, also increases the size of a term by no more than a factor of 2. All the remaining *Sel* redexes can be removed in linear time. The ⊔ rule which only rearranges the terms in the tree can also be applied in linear time. Thus normal forms for the above system can be computed in linear time.

5.3.5 An Example

Consider the matrix multiplication program,

```
{
 matmul A B C i n =
  if (i = n) then C
  else
   matmul(A, B, upd(C,i,compute_row(A, B, sel(C,i),i,0,n)), i+1, n);

 compute_row  A B X i j n =
  if (j = n) then X
  else
   compute_row(A,B, upd(X,i, dot_product(sel(A,i),B,i,j,0,n,0)),i,j+1,n);

 dot_product A B i j k n sum =
  if k = n then sum
  else
   dot_product(A, B, i,j,k+1,n,sum +sel(A,k)*sel(sel(B,k),j));
}
```

The abstract functions for these definitions are

$$\begin{aligned}
matmul\ A\ B\ C\ i\ n &= C \sqcup matmul(A, B, Upd(C), \emptyset, n) \\
compute_row\ A\ B\ X\ i\ j\ n &= X \sqcup compute_row(A, B, Upd(X), i, \emptyset, n) \\
dot_product\ A\ B\ i\ j\ k\ n\ sum &= sum \sqcup dot_product(A, B, i, j, \emptyset, n, \emptyset)
\end{aligned}$$

The symbolic non-recursive solutions are

$$\begin{aligned}
matmul\ A\ B\ C\ i\ n &= C \sqcup Upd(C) \\
compute_row\ A\ B\ X\ i\ n &= X \sqcup Upd(X) \\
dot_product\ A\ B\ i\ n\ sum &= sum
\end{aligned}$$

We can conclude that matrix multiplication either returns its third argument or returns a new array obtained by updating the third argument.

5.4 Sharing Analysis

To compute liveness of a variable, we must compute sharing among program variables. Two formal parameters x and y of a function are shared if they can be bound to arrays that have common components. Recall that in the case of flat arrays the only form of sharing possible is aliasing.

5.4.1 The Sharing Domain

Sharing information between two arrays is specified by the array names and the corresponding level of one of the arrays at which sharing occurs. The sharing domain Sh is given in Figure 37.

The interpretation of the sharing information $(x, i) \in sh[y]$ is that (y, i) may share structure with x.

5.4.2 Auxiliary Sharing Function

The sharing function *compute_sharing* computes the level at which an aggregate shares with another aggregate. Notice that the sharing function is not symmetric in its arguments. For instance an array A could share with B at level 1, whereas B could share with A at level 0. Sharing between two abstract arrays is defined below:

$$
\begin{array}{ll}
\textit{compute\_sharing} & : \quad D \times D \times Sh \rightarrow N \\
\textit{compute\_sharing } a\ b\ sh & = \\
 & \quad \textbf{cases} \\
 & \quad a = (x,i), b = (x,j) \ : \\
 & \qquad \textit{if } i \leq j \textit{ then } \ j - i \textit{ else } 0 \\
 & \quad a = (x,i),\ b = (y,j),\ \langle y,n \rangle \in sh[x] \ : \\
 & \qquad \textit{if } i \leq n \textit{ then } \ n - i \textit{ else } 0 \\
 & \quad a \in A,\ b = u(w) \in U \ : \textit{compute\_sharing(a,w)} \\
 & \quad a = u(w) \in U,\ b = u(v) \in U \ : 1 + \textit{compute\_sharing}(w,v) \\
 & \quad \textit{end}
\end{array}
$$

5.4.3 The Sharing Function $\mathcal{SH}$

With our simplifying assumptions, the only expression that can cause sharing among named arrays is the function call. We determine the values propagated by the actual parameters using the information obtained from propagation analysis. For each pair of formal parameters of the function, the maximum possible sharing is computed using *compute_sharing*. The equations are given in Figures 38 and 39. The operator * is the downward closure operator on the elements of the power domain.

$$
\begin{array}{lcl}
Ds & = & V \times N \\
Sh & = & V \to \mathcal{P}_H(Ds) \quad \text{sharing domain}
\end{array}
$$

Figure 37: Domains for Sharing Analysis

$$
\begin{array}{lcl}
\mathcal{SH}[\![\mathsf{c}]\!]\sigma\ \rho\ sh & = & sh \\
\mathcal{SH}[\![x]\!]\sigma\ \rho\ sh & = & sh \\
\mathcal{SH}[\![t_i]\!]\sigma\ \rho\ sh & = & sh \\
\mathcal{SH}[\![\mathsf{op}(se_1,\ldots,se_n)]\!]\sigma\ \rho\ sh & = & sh \\
\mathcal{SH}[\![\mathsf{sel}(se_1,se_2)]\!]\sigma\ \rho\ sh & = & sh \\
\mathcal{SH}[\![\mathsf{upd}(se_1,se_2,\ se_3)]\!]\sigma\ \rho\ sh & = & sh \\
\mathcal{SH}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\sigma\ \rho\ sh & = & \mathcal{SH}[\![e_1]\!]\sigma\ \rho\ sh\ \sqcup\ \mathcal{SH}[\![e_2]\!]\sigma\ \rho\ sh \\
\mathcal{SH}[\![f_k(se_1,\ldots,se_n)]\!]\sigma\rho\ sh & = & \\
\quad let \quad v_1 & = & \mathcal{H}[\![se_1]\!]\sigma\ \rho \\
\quad\quad\ v_2 & = & \mathcal{H}[\![se_2]\!]\sigma\ \rho \\
\quad\quad\ \vdots & & \\
\quad\quad\ v_n & = & \mathcal{H}[\![se_n]\!]\sigma\ \rho \\
\quad\quad\ a_{ij} & = & \bigsqcup\{compute\_sharing(x,y,sh) | x \in v_i, y \in v_j, 1 \le i \ne j \le n\} \\
\quad in & & \\
\quad\quad sh \sqcup [x_{k_i} \mapsto \{(x_{k_j}, a_{ij})\}^*] & & \\
\quad end & &
\end{array}
$$

$$
\mathcal{SH}[\![\mathsf{let}\ [t_1 = e_1,\ldots,t_n = e_n]\ \mathsf{in}\ t_i\ \mathsf{end}]\!]\sigma\ \rho\ sh \ = \ \bigsqcup_{i=1}^{n}\{\mathcal{SH}[\![e_i]\!]\sigma\ \rho\ sh\}
$$

Figure 38: Sharing function $\mathcal{SH} : IExp \to VEnv \to FEnv \to Sh \to Sh$

$$
\begin{array}{l}
\mathcal{SH}_p[\![pr]\!] \ = \\
\quad let \quad \rho = \mathcal{H}_p[\![pr]\!] \\
\qquad\quad venv_0 = [,\ldots,x_{k_i} \mapsto \{(x_{k_i},0)\}^*,\ldots] \\
\quad in \\
\qquad\quad fix(\lambda\sigma.\ (\bigsqcup_{i=1}^{n}\mathcal{SH}[\![e_i]\!]venv_0\ \rho\ \sigma)) \\
\quad end
\end{array}
$$

Figure 39: $\mathcal{SH}_p : IProg \to Sh$

5.5 Copy Avoidance

To avoid internal sharing, we assumed an implementation of the update operator that always performs a deep copy of its third argument. We would like to eliminate this copying where possible.

Consider an expression `upd(a,i,x)`. Without any information, we have to deep copy `x`. If `x` is a new array that doesn't contain components of any other array, or `x` is a possibly updated version of `sel(a,i)`, then copying can be avoided. These cases are common in typical programs.

For copy avoidance, we describe an address propagation analysis similar to the one discussed in Chapter III. For detecting cases such as a being updated at index `i` with a value `sel(a,i)`, we have to carry around the index information. We define a sets of values $S_0, \ldots, S_d$ inductively as

$$\begin{aligned} \{c \mid c \textit{ is a constant}\} &\subseteq S_0 \\ \{x \mid \textit{dimension}(x) = i\} &\subseteq S_i,\ \forall i\ 0 \leq i \leq d \\ x \in S_i, y \in S_0 &\Rightarrow x.y \in S_{i-1} \end{aligned}$$

The domain of abstract values for copy avoidance is a flat domain of $\bigcup_{i=0}^{d} S_i$ as shown in Figure 40.

The abstraction of primitive operators is defined as

$$
\begin{array}{lcl}
D_{is} & = & \text{Flat Domain of } \bigcup_{i=0}^{d} S_i \\
IEnv & = & V \rightarrow D_{is} \\
IFnv & = & F \rightarrow {D_{is}}^n \rightarrow D_{is}
\end{array}
$$

Figure 40: Domains for Copy Avoidance

$$
\begin{array}{lcl}
\mathcal{I}[\![\mathrm{c}]\!]\sigma\rho & = & \mathcal{K}^1[\![\mathrm{c}]\!] \\
\mathcal{I}[\![x]\!]\sigma\rho & = & \sigma[x] \\
\mathcal{I}[\![t_i]\!]\sigma\rho & = & \mathcal{I}[\![expr\_of(t_i)]\!]\sigma\rho \\
\mathcal{I}[\![op]\!](se_1,\ldots,se_n)\sigma\rho & = & \mathcal{I}[\![op]\!](\mathcal{I}[\![se_1]\!]\sigma\rho,\ldots,\mathcal{I}[\![se_n]\!]\sigma\rho) \\
\mathcal{I}[\![\texttt{if } se_0 \texttt{ then } e_1 \texttt{ else } e_2]\!]\sigma\rho & = & \mathcal{I}[\![e_1]\!]\sigma\rho \ \sqcup \ \mathcal{I}[\![e_2]\!]\sigma\rho \\
\mathcal{I}[\![f_i(e_1,\ldots,e_n)]\!]\sigma\rho & = & \rho[f_i](\mathcal{I}[\![e_1]\!]\sigma\rho,\ldots,\mathcal{I}[\![e_n]\!]\sigma\rho) \\
\mathcal{I}[\![\texttt{let } [t_1 = e_1,\ldots,t_n = e_n] \texttt{ in } t_i \texttt{ end}]\!]\sigma\rho & = & \mathcal{I}[\![t_i]\!]\sigma\rho
\end{array}
$$

Figure 41: $\mathcal{I} : IExp \rightarrow IEnv \rightarrow IFnv \rightarrow D_{is}$

$$
\mathcal{I}_p[\![pr]\!] \ = \ fix(\lambda\sigma.\, \sigma[f_i \mapsto \lambda y_1,\ldots,y_k.\, \mathcal{I}[\![exp_i]\!][x_{1_i} \mapsto y_1,\ldots,x_{k_i} \mapsto y_k]\sigma]
$$

Figure 42: $\mathcal{I}_p : IProg \rightarrow IFnv$

$$
\begin{aligned}
\mathcal{I}[\![\texttt{array}]\!]x\ y &= \bot \\
\mathcal{I}[\![\texttt{sel}]\!]x\ y &= \mathit{if}\ (x = \bot)\ \mathit{then}\ \bot \\
&\quad\ \mathit{else\ if}\ (x = \top)\ \mathit{then}\ \top \\
&\quad\ \mathit{if}\ (y = \bot) \vee (y = \top)\ \mathit{then}\top \\
&\quad\ \mathit{else}\ x.y \\
\mathcal{I}[\![\texttt{upd}]\!]x\ y\ z &= \mathit{if}(x = \bot)\ \mathit{then}\ \bot\ \mathit{else}\ \top \\
\mathcal{I}[\![\texttt{upd!}]\!]x\ y\ z &= x \\
\mathcal{I}[\![\texttt{+}]\!]x\ y &= \top
\end{aligned}
$$

An anonymous array returns a new array which is abstracted to $\bot$. The `sel` operator is abstracted as a concatenation operator. An upd operation returns $\bot$ if its first argument is $\bot$. It is based on the observation that a non-destructive update of a named array always creates a shared array. A primitive operator such as + returns $\top$. Currently we do not keep track of values computed by the primitive operators. These values can be recorded using the temporary variable names they are bound to in the intermediate representation.

The criterion for copy avoidance can be formulated as follows. Given a program `pr` and an expression `upd(e_1,e_2,e_3)` and an identity variable environment I_d, let *v_1*, *v_2* and *v_3* be $\mathcal{I}[\![\texttt{e\_1}]\!]\ (\mathcal{I}_p[\![\texttt{pr}]\!])\ I_d$, $\mathcal{I}[\![\texttt{e\_2}]\!]\ (\mathcal{I}_p[\![\texttt{pr}]\!])\ I_d$, and $\mathcal{I}[\![\texttt{e\_3}]\!]\ (\mathcal{I}_p[\![\texttt{pr}]\!])\ I_d$ respectively. Deep copying of argument `e_3` can be avoided if *v_3* is $\bot$ or *v_2* is not $\top$ and $v\_1.v\_2 = v\_3$.

Our algorithm determines that all copies can be avoided in the matrix multiplication example.

5.6 Selects-and-Updates Analysis

This phase determines the set of arrays selected and updated by an expression. Instead of the Hoare power domain, we use the power set of D to represent arrays selected and updated because we must distinguish between an expression that only updates (x, i) and one that updates both (x, i) and $(x, i+1)$. We use a function *named* defined as

$$\begin{aligned} &named : \mathcal{P}_H(D) \rightarrow \mathcal{P}(D) \\ &named\ X \;=\; \{x \mid x \in X \cap A, \forall y \in X, x \sqsubseteq y \Rightarrow x = y\} \end{aligned}$$

The domains and the flow functions needed are described in Figures 43, 44, and 45.

5.7 Order of Evaluation

The choice of an order of evaluation is very much as described in Chapter II. Given the arrays selected and updated by each expression, the data dependency graph is augmented by *interference* edges that represent our hope to perform selections before updates for each array. Interference is determined by the *interference* predicate defined in Figure 46. A good (or at least reasonable) order of evaluation is obtained by attempting a topological sort of the resulting preorder.

5.8 Abstract Reference Count Analysis

5.8.1 Reference Count Domain

For abstracting the number of references to an array, we use a domain R with elements $\{\infty, 0, 1, 2, 3, \ldots, k\}$. The ordering on these elements is $\forall x \in R\ x \sqsubseteq \infty, \forall x \neq$

$$
\begin{array}{lcll}
D_{su} & = & \mathcal{P}(A) \times \mathcal{P}(A) & \text{Selected and Updated Named Arrays} \\
VEnv & = & V \to \mathcal{P}_H(D) & \text{Variable Environments} \\
FEnv & = & F \to \mathcal{P}_H(D)^n \to \mathcal{P}_H(D) & \text{Propagation Function Environments} \\
SEnv & = & F \to \mathcal{P}_H(D)^n \to D_{su} & \text{Selects-Updates Function Environments}
\end{array}
$$

Figure 43: Domains for Selects-and-Updates Analysis

$$
\begin{array}{lcl}
\mathcal{S} & : & IExp \to SEnv \to FEnv \to VEnv \to D_{su} \\
\mathcal{S}[\![\mathtt{c}]\!]\delta\ \rho\ \sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![x]\!]\delta\ \rho\ \sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![t_i]\!]\delta\ \rho\ \sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![\mathtt{op}(se_1, \ldots, se_n)]\!]\delta\ \rho\ \sigma & = & \langle \emptyset, \emptyset \rangle \\
\mathcal{S}[\![\mathtt{sel}(se_1,se_2)]\!]\delta\ \rho\ \sigma & = & \langle named(\mathcal{H}[\![se_1]\!]\sigma\ \rho), \emptyset \rangle \\
\mathcal{S}[\![\mathtt{upd}(se_1,se_2,\ se_3)]\!]\delta\ \rho\ \sigma & = & \langle \emptyset, named(\mathcal{H}[\![se_1]\!]\sigma\ \rho) \rangle \\
\mathcal{S}[\![\mathtt{if}\ se_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!]\delta\ \rho\ \sigma & = & \mathcal{S}[\![e_1]\!]\delta\ \rho\ \sigma\ \sqcup\ \mathcal{S}[\![e_2]\!]\delta\ \rho\ \sigma \\
\mathcal{S}[\![f_k(se_1, \ldots, se_n)]\!]\delta\ \rho\ \sigma & = & \\
\quad \delta[f_k](\mathcal{H}[\![se_1]\!]\sigma\ \rho, \ldots, \mathcal{H}[\![se_n]\!]\sigma\ \rho) & &
\end{array}
$$

$$
\mathcal{S}[\![\mathtt{let}\ [t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ t_i\ \mathtt{end}]\!]\delta\ \rho\ \sigma = \bigsqcup_{i=1}^{n} \mathcal{S}[\![e_i]\!]\delta\ \rho\ \sigma
$$

Figure 44: Selects-and-Updates function $\mathcal{S}$

$$
\begin{array}{l}
\mathcal{S}_p[\![pr]\!] = fix(\lambda\delta.\ \delta[f_i \mapsto \lambda y_1, \ldots, y_k.\mathcal{S}[\![e_i]\!]\ \delta\ (\mathcal{H}_p[\![pr]\!]) \\
\qquad\qquad [x_{1_i} \mapsto y_1, \ldots, x_{k_i} \mapsto y_{k_i}]])
\end{array}
$$

Figure 45: $\mathcal{S}_p : IProg \to SEnv$

$\infty\ x+1 \sqsubseteq x$.

The interpretation of each element of the domain is

1. ∞ represents an object with multiple references.

2. i represents an array that has but a single reference through level i. There may be multiple references to the components at level $i+1$ or lower levels.

3. k is the least element of R, and represents a scalar value. The value of k can be chosen as the maximum dimension of an array that appears in a particular program.

5.8.2 Live Variable Occurrences

As noted in Section 5.1.3, an array may be dead even though an array of which it is a proper component is live. Suppose `A` has dimension 3 or greater and consider a function call

```
f(sel (sel (A, i), j)
```

The array `sel (sel (A, i), j)` is dead after the call to `f` if the only future uses of `A` are updates of `A` at `i`, and of `sel(A,i)` at `j`. This information can often be obtained by first considering all syntactic occurrences of `A` that are live, and discarding those which satisfy the above criterion.

We define a variable occurrence as a pair, the variable and the program point at which it occurs. We define a function called *Live* which takes a set *Occ* of variable occurrences and a set of variables S and returns the set of live variables with respect to S.

$$
\begin{aligned}
&\mathit{Live\ Occ\ S}\ = \\
&\quad \{x|(x,\_) \in (\mathit{Occ} \setminus \{(v,t_2) \mid t_2 = \mathrm{upd}(v,y,z), a \in S, \\
&\quad a \text{ is a proper component of } \mathtt{sel}(v,y)\ \}
\end{aligned}
$$

In the above example, `sel (sel (A, i), j)` is a proper component of `A` and of `sel(A,i)`.

5.8.3 Auxiliary Functions for Reference Count Analysis

We define three auxiliary functions *incr_absrc* (Figure 48), *ref_agg* (Figure 49) and *Names* (Figure 50) for computing the reference count of an array. The function *ref_agg* computes the reference count of an abstract array given the set of live variables, the reference count environment, and the sharing information. *Names* determines the set of named arrays in an abstract value. We define *Ref* by extending *ref_agg* to a set of elements.

5.8.4 The function $\mathcal{R}$

The definition of $\mathcal{R}$ (see Figure 52) needs explanation only for the function call and the let-expression. For each argument se_i, we first determine the set of actually live variables $live_i$ with respect to the set $FV(se_i)$ using *Live*. If a variable $se_i \in live_i$, the abstract reference count of the se_i is ∞. Otherwise we determine abstract reference counts of the arrays propagated by each actual parameter of the function using *Ref*. The reference environment is modified using the abstract reference count so computed. For a let-expression, we first find the set of future references of the variables using *Vars_at*, a variation of *Vars* of Chapter II which also records the

temporary variable, and apply $\mathcal{R}$ to each e_i recursively. The function $\mathcal{R}_p$ (Figure 53) computes the abstract reference count of all variables (formal parameters). The environment $venv_0$ (see Figure nf-r-prog) maps every program variable x to the closure of a singleton $\{(x,0)\}$ (recall that $(x,0)$ is the array referred to by x).

Given $\mathcal{R}_p[\![pr]\!]$, it is easy to decide if an update can be performed destructively. Consider an update expression, $t_i = \mathtt{upd}(x,y,z)$. Suppose that $lset_i$ is the set of variables that are live at point i. This update cannot be made destructively if the abstract reference count of array propagated by x is equal to ∞. This condition is formally expressed as:

$$Ref(\mathcal{H}[\![x]\!]\ venv_0\ (\mathcal{H}_p[\![pr]\!]), lset_i, \mathcal{SH}_p[\![pr]\!], \mathcal{R}_p[\![pr]\!]) = \infty$$

We have not yet implemented the algorithm for non-flat arrays, and do not yet have a precise analysis of its polynomial time complexity or a rough analysis of its typical efficiency on programs with non-flat arrays. The algorithm is similar to our previous algorithm, which leads us to believe it will straightforward to implement and reasonably efficient. One of our future research goals is to implement the algorithm and to study its effectiveness on sparse matrix algorithms.

$$
\begin{array}{lll}
interferes\ e_i\ e_j\ pr & = & let \\
 & & \quad fenv = \mathcal{H}_p[\![pr]\!] \\
 & & \quad sh = \mathcal{SH}_p[\![pr]\!] \\
 & & \quad venv_0 = [\ldots, x_{k_i} \mapsto \{x_{k_i}\}^*, \ldots] \\
 & & \quad suenv = \mathcal{S}_p[\![pr]\!] \\
 & & \quad < s_i, u_i >= \mathcal{S}[\![e_i]\!] suenv\ fenv\ venv_0 \\
 & & \quad < s_j, u_j >= \mathcal{S}[\![e_j]\!] suenv\ fenv\ venv_0 \\
 & & in \\
 & & \quad ((u_j \cup \{(y, d-1) | (x, m) \in u_j, (y, k) \in sh[x], k \leq m\}) \cap \\
 & & (u_i \cup s_i)^*) \neq \emptyset \\
 & & end
\end{array}
$$

Figure 46: The Interference Predicate : $IExp \rightarrow IExp \rightarrow IProg \rightarrow Bool$

$$
\begin{array}{l}
incr\_absrc \quad : \quad R \rightarrow N \rightarrow R \\
incr\_absrc\ r\ i \ = \\
\quad if \ \ r = \infty\ then\ \infty \\
\quad else \ \ if \ \ r - i > 0\ then\ r - i \\
\quad else\ \infty
\end{array}
$$

Figure 48: The Function $incr\_absrc : R \rightarrow N \rightarrow R$

$$
\begin{array}{l}
ref\_agg : D \rightarrow REnv \rightarrow \mathcal{P}_H(A) \rightarrow Sh \rightarrow R \\
ref\_agg\ x\ renv\ L\ sh \ = \\
\quad \textbf{case} \ \ x \ \textbf{of} \\
\quad isA(x)(x = (b, i)) \ : if \ x \in L\ then\ \infty \\
\qquad\qquad else \ \sqcup \{compute\_sharing(x, y, sh) \mid y \in L\} \sqcup incr\_absrc(renv[b], i) \\
\quad isU(x)(x = u(y)) : if \ ref\_agg(y) = \infty\ then\ 0\ else\ 1 + ref\_agg(y) \\
\quad end
\end{array}
$$

Figure 49: The Function $ref\_agg : D \rightarrow REnv \rightarrow \mathcal{P}_H(A) \rightarrow Sh \rightarrow R$

$$
\begin{array}{l}
Names \quad : \quad D \rightarrow \mathcal{P}_H(A) \\
Names\ a \ = \\
\quad cases \\
\qquad a \in A \ : \ \{a\}^* \\
\qquad a \in U \text{ and } a = u((v, i)) \ : \ \{(v, i+1)\}^* \\
\quad end
\end{array}
$$

Figure 50: The Function $Names : D \rightarrow \mathcal{P}_H(A)$

$FEnv$	$=$	$F \to \mathcal{P}_H(D)^n \to \mathcal{P}_H(D)$	Propagation Function Environments
Sh	$=$	$V \to \mathcal{P}_H(Ds)$	Sharing Domain
R	$=$	$\{\infty, 0, 1, 2, \ldots, k\}$	Abstract Reference Counts
$REnv$	$=$	$V \to R$	Reference Count Environments
$FOcc$	$=$	$\mathcal{P}((V \cup T) \times T)$	Variable Occurrences

Figure 51: Domains for Reference Count Analysis

$$
\begin{array}{l}
\mathcal{R}[\![\mathtt{c}]\!] renv\ venv\ fenv\ sh\ fset = renv \\
\mathcal{R}[\![x]\!] renv\ venv\ fenv\ sh\ fset = renv \\
\mathcal{R}[\![t_i]\!] renv\ venv\ fenv\ sh\ fset = renv \\
\mathcal{R}[\![\mathtt{op}(se_1, \ldots, se_n)]\!] renv\ venv\ fenv\ sh\ fset = renv \\
\mathcal{R}[\![\mathtt{sel}(se_1, se_2)]\!] renv\ venv\ fenv\ sh\ fset = renv \\
\mathcal{R}[\![\mathtt{upd}(se_1, se_2, se_3)]\!] renv\ venv\ fenv\ sh\ fset = renv \\
\mathcal{R}[\![\mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ ]\!] renv\ venv\ fenv\ sh\ fset = \\
\quad \mathcal{R}[\![e_1]\!] renv\ venv\ fenv\ sh\ fset \sqcup \mathcal{R}[\![e_2]\!] renv\ venv\ fenv\ sh\ fset \\
\mathcal{R}[\![f_k(se_1, \ldots, se_n)]\!] renv\ venv\ fenv\ sh\ fset = \\
\quad let\ r_1 = if\ se_1 \in Live(fset, FV(se_1))\ then\ \infty \\
\qquad\qquad else\ Ref(\mathcal{H}[\![se_1]\!] venv\ fenv, renv, \\
\qquad\qquad \bigsqcup\{Names(x) \mid x \in \mathcal{H}[\![t]\!] venv fenv, t \in Live(fset, FV(se_1)\}, sh))\} \\
\qquad \vdots \\
\qquad r_n = if\ se_n \in Live(fset, FV(se_n))\ then\ \infty \\
\qquad\qquad else\ Ref(\mathcal{H}[\![se_n]\!] venv\ fenv, renv, \\
\qquad\qquad \bigsqcup\{Names(x) \mid x \in \mathcal{H}[\![t]\!] venv fenv, t \in Live(fset, FV(se_n)\}, sh))\} \\
\quad in \\
\qquad renv \sqcup [x_{k_1} \mapsto r_1, \ldots, x_{k_n} \mapsto r_n] \\
\quad end \\
\mathcal{R}[\![\mathtt{let}[t_1 = e_1, \ldots, t_n = e_n]\ \mathtt{in}\ t_i\ \mathtt{end}]\!] renv\ venv\ fenv\ sh\ fset = \\
\quad let\ fset_i = fset \cup \{\bigcup_{j=i+1}^{n} Vars\_at(t_j, e_j)\} \\
\qquad \cup\{(x, t_k) \mid 1 \le l < i, i < k \le n, x \in FV(expr\_of(t_k))\} \\
\quad in \\
\qquad \bigsqcup_{i=1}^{n} \mathcal{R}[\![e_i]\!] renv\ venv\ fenv\ sh\ fset_i \\
\quad end
\end{array}
$$

Figure 52: $\mathcal{R} : IExp \to REnv \to VEnv \to FEnv \to Sh \to FOcc \to REnv$

$$
\begin{array}{lll}
\mathcal{R}_p[\![pr]\!] = & & \\
\quad let & fenv & = \mathcal{H}_p[\![pr]\!] \\
& sh & = \mathcal{SH}_p[\![pr]\!] \\
& venv_0 & = [, \ldots x_{k_i} \mapsto \{(x_{k_i}, 0)\}^*,] \\
\quad in & & \\
& fix(\lambda\sigma. \bigsqcup_{i=1}^{n} \mathcal{R}[\![e_i]\!]\ \sigma\ venv_0\ fenv\ sh\ \emptyset) & \\
\quad end & &
\end{array}
$$

Figure 53: $\mathcal{R}_p : IProg \rightarrow REnv$

CHAPTER VI

RELATED WORK

6.1 An Overview

Copy avoidance is one of the most important problems to be solved for the efficient implementation of functional languages. The side-effect-free semantics requires that any operation that modifies a data structure be implemented by copying. For aggregate structures such as arrays it is a serious problem because the complexity of an algorithm can suffer by orders of magnitude. However, if an update operation occurs in a context in which it is guaranteed that there is no other use of the old data structure, then the modification can be implemented in-place without sacrificing functional semantics.

One approach to this problem is to detect cases where an update can be made in-place. Another is to choose an implementation of the data structure in which it is relatively easy to maintain multiple versions created by incremental modifications. In the next two sections we discuss the runtime and compile time techniques for copy avoidance.

6.2 Runtime Techniques

6.2.1 Reference Counting

In this scheme, every object has an extra field indicating the number of references to that object. When a new reference is to the object is created, the count is incremented. When a reference is used the count is decremented. When the count becomes 0, the storage for the object can be reclaimed. An update can be performed destructively if the reference count of the object is 1.

There are some disadvantages with reference counting. Every object needs an extra field for storing the reference count. The program incurs the overhead of maintaining these reference counts. In the case of parallel execution these reference counts have to be incremented and decremented atomically which incurs additional overheads. Examples of reference counting technique are [19, 82].

6.2.2 Persistent Arrays

Non-destructive updates can be less expensive if multiple versions of the data structure can be maintained without too much overhead. For instance, an array of size n can be represented as a balanced binary search tree of n leaves, with array indices as search keys. The cost of a functional update is $O(logn)$ instead of $O(n)$. However, now reading an element of the array takes logarithmic time.

The tree based array representation assumes that all versions of the array are equally likely to be accessed. For applications in which arrays are used in a single threaded manner, only the most recently updated array is accessed or updated. An array with trailers [12] provides constant time access and update to the most recent version of the array and penalizes the access of an older version proportional to the

number of updates.

Baker observes [8] that arrays with trailers are equivalent to the implementation of functional arrays using the shallow binding technique [7]. The basic idea is to represent updates on an array as a tree where each node represents one version of the array obtained by an update operation to its parent. The implementation of the array operations maintains the following invariant: the root of the tree is the version of the array read or updated most recently. For single threaded access and updates, $O(1)$ complexity is achieved. Multithreaded access or an update requires rerooting the tree if that version being operated upon is not the root. Rerooting can be done in time proportional to the number of updates. If there are a sufficiently large number of reads on the same version, all but the first read take $O(1)$ time. In his PhD thesis, Chuang [20] extends the shallow binding technique to *fragmented shallow binding* where the cost of an update is $O(1)$ and the amortized cost of a read operation, in a sufficiently long sequence of reads on arrays where the arrays being read form a path in the tree, is also $O(1)$.

There is one problem with the runtime approach. Even if array accesses and updates are made in a single threaded manner, there is an extra overhead involved in maintaining these data structures. One can get the performance of imperative arrays only if the updates are known to be destructive at compile time.

6.3 Compile Time Techniques

In this section we discuss the related work on storage optimization which subsumes update analysis and other optimizations such as life time analysis and compile time storage deallocation. In general there are three approaches to compile-time detection of destructive updates. In the *verification* approach the programmer asserts

that it is safe to destructively update certain data objects leaving it for the compiler to verify the assertion by using an appropriate type system [40, 77]. Another closely related approach requires the programmer to write programs in a restricted style that guarantees that all updates can be performed be side-effect.

The *optimization* approach leaves it to the compiler to detect updates that can be implemented by side effect. Optimization is more flexible than verification.

6.3.1 Optimization Techniques

The earliest work on storage optimization found a linear order of evaluation for the nodes of a labeled dag where the labels represent identifiers, nodes represent assignment statements, and the edges represent data dependencies; it was formalized by Sethi as a pebble game on graphs with labels [68]. Sethi's work applies to basic blocks with only primitive operators. We assume arbitrary functions as operators, which necessitates our interprocedural analysis. After deriving the interprocedural information, we derive an order locally in essentially the same way as Sethi.

The other research work in the area of storage optimization is globalization of formal parameters of a functions. Schmidt [64] gave syntactic criteria for converting the store argument of the direct semantics of an imperative language into a global variable. This work was generalized as the globalization of function parameters by Sestoft [67, 36]. It also assumes a fixed order of evaluation of expressions. Fradet [31] gave a simple syntactic criterion, based on the types of the variables, for detecting single threadedness in programs written in continuation passing style.

The initial work on update analysis of a call-by-value functional languages was Hudak's abstract reference counting technique for a first-order language with flat aggregates [45]. We compared this work to ours in Chapter II. Our analysis is more

effective, a great deal more efficient, and derives a good order of evaluation instead of assuming a fixed order as in [45].

Gopinath [37] considers copy elimination in the single assignment language SAL, which has constructs for specifying *for* loops. His work involves computing the target address of an object returned by an expression using a syntactic index analysis and assuming the liveness analysis of [45]. Again, this work also does not consider reordering expressions.

Bloss [12, 13] extended the work on update analysis to first-order lazy functional languages, which are more difficult to analyze because the order of evaluation of expressions cannot be completely determined at compile-time. She defines a non-standard semantics called path semantics [12, 14] which gives information about all possible orders of evaluation. Path semantics is used to check whether an update can be performed destructively. Computing the abstract path semantics is very expensive because of the size of the abstract domain of paths [12]. This work also assumes a fixed order of evaluation for strict operators. Draghicescu's [29] work on update analysis for lazy languages improves the abstract complexity but is still exponential.

Deutch [27] describes an analysis based on abstract interpretation for determining the lifetime and aliasing information for higher-order languages. This analysis is based on abstracting the operational semantics of a very low level intermediate language. Our work differs from Deutsch's in three ways. Since we do not associate objects with expression labels generating them, we do not introduce spurious aliasing. Deutch does not address the complexity of his analysis so it is not clear whether it would be efficient if restricted to the first-order case. Deutch also assumes a fixed order of evaluation of expressions.

Hicks [42] derives the lifetime information of objects in Id, a parallel single assignment language developed at MIT [55]. Lifetime information is used for validating deallocation instructions in the program or automatically inserting deallocation instructions for reclaiming the storage. This work does not address the aggregate update problem directly because Id does not provide update as a language construct (although update operations could be defined in Id). Furthermore the algorithm appears to be exponential.

Cann [16] reports that, for a set of 8 benchmarks, a linear-time analysis in the SISAL compiler is able to make 99–100% of the updates destructive. The SISAL compiler in question performs no interprocedural update analysis, however, relying on "run time reference counting to identify update-in-place opportunities" for arrays that cross procedure boundaries [17]. We surmise that the effectiveness of the SISAL compiler's update analysis on these benchmarks is an artifact of their origin as Fortran 77 programs, which contain no recursion and probably contain relatively few procedure calls.

There has been no previous work on the specific problem of update analysis for non-flat aggregates. The most closely related work is by Deutsch, who has considered aliasing and liveness in higher-order strict languages [27] and has more recently developed a general model of aliasing for non-flat aggregates [28]. Also related is the analysis of recursive types by Wang and Hilfinger, who have expressed hope that their analysis can be extended to gather information for update analysis [80]. These papers do not address the issues of copy introduction and avoidance, local optimization, and choice of order of evaluation that we believe must be addressed as part of an effective optimization-based approach to update analysis of non-flat aggregates. Our symbolic

calculation of propagation information is an application of [21].

Deutsch's model of aliasing for non-flat aggregates is suitable for separate compilation [28], but the efficiency of these algorithms is otherwise unknown.

6.3.2 Verification Techniques

The verification approach to solving the update problem typically imposes a type discipline that can express the intended use of a variable as well as the domain of its values. Update analysis then reduces to type checking. Examples of this approach include Wadler's work on linear types [77] and Guzman's single threaded lambda-calculus [41, 40]. Other similar ideas include monads[76], mutable abstract datatypes[46], syntactic criteria for single threading [64, 31], and the Imperative Lambda Calculus, which introduces assignment in a functional language with a stratified type system without sacrificing confluence [72]. These systems typically require the programmer to impose a sequential order of evaluation that guarantees single threading. None of this work has yet confronted the problems of non-flat arrays or other aggregates or parallel evaluation. Guzman's thesis recognized the importance of copy introduction for flat arrays [41].

6.4 On Parallel Execution

There hasn't been much work on update analysis in parallel functional languages. The only reported work is that of Gopinath [37, 38] and SISAL [16, 15]. SISAL does not handle recursion. Gopinath's analysis has a worst case exponential complexity. Our analysis is simplified by the partition and combine operations. We believe that our algorithm is the first practical algorithm for update analysis of parallel functional programs. P. Wadler has proposed monads as an approach to de-

structive updating [76]. Monads sequentialize the execution to achieve destructive updating and therefore are not suitable for parallel execution. We have taken an orthogonal approach: instead of sequentializing all updates, we divide the array into semantically different arrays by the `partition` operator allowing the updates to be done in parallel. Guzman [41] and Swarup *et al* [72] also assume sequential evaluation.

Monolithic arrays were proposed because of the difficulty of expressing parallel updates [6, 44]. In Chapter III, we gave a generalization of the incremental update to express a collection of updates on an array. Monolithic arrays are a special case of this operator. Wadler's new monolithic array construct [75] needs additional data structures for performing combining operations, whereas in our approach combining is done at the array itself. Another relevant work in the context of specifying a collection of operations is the xapping data structure of Connection Machine Lisp [70], which is based on the SIMD model of computing. The programming language Id [55], a non-strict language, provides accumulators as an extension of arrays. An accumulator is allocated as a new array with initial values and all accumulations are performed atomically by an accumulating operator. The accumulators of Id appear to have been derived from the monolithic array operator, whereas we have generalized the incremental update operator.

The most related work in the area of runtime systems for mostly functional languages is that of QLisp[1, 33] and the implementation of MulT, a parallel dialect of Scheme with futures [51]. QLisp is a queue-based multiprocessing implementation of a parallel dialect of Lisp with explicit coarse grain parallelism. When a program executes a statement expressing parallelism, it adds new processes to the queue of tasks to be processed. After finishing the current task the processor picks a new

task for execution from the task queue. The decision to spawn a process is explicitly specified by the programmer as a predicate. In the dynamic partitioning scheme [57], spawning of parallel processes is also based on the state of the runtime system.

The implementation of Mul-T is based on the technique of lazy task creation [54]. In this scheme each processor begins execution sequentially. A starving processor steals work from the stack of an executing process. Task stealing requires locking the stack of another process that runs on a different processor. Therefore the runtime system for lazy task creation is based on a shared memory model and is not clear how one can extend it to distributed memory machines. In our model the stacks are local to the processor, therefore the runtime system can be easily modified for a distributed memory implementation. The data objects such as arrays are globally shared, therefore we do need schemes for allocation of distributed arrays — an important area for further research. We hope to borrow the techniues of data allocation used in high performance Fortran [50].

CHAPTER VII

CONCLUSIONS

In this chapter we summarize the contributions of this dissertation and outline the directions for future research.

7.1 Contributions

This thesis presents the first practical interprocedural update analysis algorithm for strict first order functional languages with arrays of scalars. The algorithm makes all updates destructive in several common numerical algorithms written in a functional style. The analysis runs in polynomial time in the worst case and in linear time for typical programs. All previous algorithms of this kind require exponential time in the worst case. The algorithm does not assume a fixed evaluation order and derives a good order that maximizes opportunities for destructive updating. The simplicity of the algorithm makes it adaptable to separate compilation.

We extended the algorithm to parallel functional languages by devising new array operations for expressing divide and conquer algorithms on arrays. Our analysis is shown to have polynomial time complexity even for parallel evaluation. To the best of our knowledge it is the first practical algorithm for update analysis of parallel functional programs. Our analysis is so effective that it removes all copies in many common numerical algorithms written as parallel functional programs. In cases where a copy cannot be eliminated, the compiler can advise the user about the source of the

problem. We also considered the implications of our algorithm for language design, and explained why we believe programmers will be able to write efficient programs that rely on update optimization. We described a new update operation for specifying a collection of updates on an array which subsumes monolithic arrays available in most functional langauges.

The update optimization has been implemented in a compiler for our parallel functional language. We described the runtime system and its implementation on a Sequent Symmetry, a bus based shared memory multiprocessor, with some preliminary performance results.

We also considered the problem of non-flat or nested but non-recursive arrays and some of the difficulties introduced by non-flatness. We presented an extension of our algorithm to handle non-flat arrays.

7.2 Future Directions

There are two possible directions for further research. One is the extension of the analysis for more powerful functional languages. The other is an efficient implementation of the runtime system on different kinds of multiprocessors.

7.2.1 Language Extensions

Update analysis of higher order functional languages is one of the most important problems for future research. Our analysis can easily be adapted to languages with a restricted kind of higher-order functions — those which do not capture any array variables. Such an extension, though limited, is quite interesting because the higher order functions of C belong to this class. For handling general higher-order functions, the notion of propagation of a variable has to be generalized because a

variable captured in a closure gets propagated. I would first like to study the use of higher-order functions in typical programs using arrays and explore whether the algorithm can be extended to handle these common cases. We know that in the worst case, higher order functions can be handled by copying the arguments when a closure is formed. An analysis to determine if a closure is singly referenced may be helpful to minimize copying in the default case.

Another possible extension is to consider languages with multiple values. In our language we have one operation, the partition operation that returns multiple values. We would like to consider arbitrary multiple values. If we add a simple syntactic restriction that a single variable is not returned as two values of an expression, then I believe that the analysis can be extended to languages with multiple values without any technical problems.

Another direction is the extension of the analysis for languages with recursive types such as lists. An interesting question is whether type information can be utilized to lower the complexity of the analysis.

I would like to investigate if our analysis is applicable to non-strict functional programs or to strict portions of a non-strict program.

7.2.2 Parallel Implementation

There are several possibities for improving the performance of the runtime sytem. One of the most important problems is increasing the granularity of parallelism in an implicitly parallel functional language. I would like to investigate the efficacy of different techniques for constraining parallelism and enhancing the performance of an implicitly parallel program. There are three approaches to this problem. One is to develop runtime mechanisms to control parallelism based on resources such

as the number of processors, average load of a processor etc. In the current implementation, parallelism is controlled by the queue sizes. Parallelism in a process can further be constrained by choosing sequential execution beyond a particular depth of the execution tree of that process.

Another approach would be to use static analysis to determine the arguments whose "size" should have a dynamic influence on the decision to evaluate in parallel. The information obtained from update analysis can be used as a heuristic for identifying these arguments. The sizes of arrays that determine the complexity of a function can also be estimated using the techniques of [60, 47].

I believe that it is necessary to introduce constructs for explicit sequentialization to improve the efficiency of a parallel program, thus leaving the important decision of sequentialization to the user. It is a viable alternative to explicit parallelism. One problem with explicit sequentialization, as with explicit parallelism, is that the performance of a program may suffer when ported to a different parallel machine. To this end, one can design a language with annotations for identifying variables that determine the complexity of a function. A predicate with a parameter which is a threshold for parallelism, expressed in terms of the granularity variables, can be used to specify explicit sequentialization. When a program is ported to a different machine, the appropriate value of the parameter for each predicate can be determined through a few trial runs on that machine.

I would also like to build a portable implementation of a parallel functional language for distributed memory multiprocessors. The important problems to be solved are distributed data allocation for arrays [50] and heuristics for process and data migration. I believe that the data distribution declarations of High Performance

Fortan can be used for allocation of distributed arrays in a functional language.

BIBLIOGRAPHY

[1] *Parallel Lisp: Languages and Systems.* Lecture Notes in Computer Science, Springer-Verlag, 1989. Volume 441.

[2] A. Aho, J. Hopcroft, and J.D. Ullman. *Design and Analysis of Computer Algorithms.* Addison Wesley Publishing Company, 1974.

[3] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley Publishing Company, 1986.

[4] Andrew. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25:275–279, 1987.

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

[6] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures : Data structures for parallel computing. In *Graph Reduction*, pages 336–369. Lecture Notes in Computer Science, Springer-Verlag, 1986. Volume 279.

[7] H. G. Baker. Shallow binding in lisp 1.5. *CACM*, 21(7):565–569, 1991.

[8] H. G. Baker. Shallow bindings makes functional arrays fast. *ACM SIGPLAN Notices*, pages 145–147, 1991.

[9] H.P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* Amsterdam: North-Holland, 1984.

[10] R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Englewood Cliffs NY: Prentice Hall, 1988.

[11] G. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[12] A Bloss. *Path Analysis and Optimization of Non-strict Functional Languages.* PhD thesis, Yale University, Dept. of Computer Science, 1989.

[13] A. Bloss. Update analysis and efficient implementation of functional aggregates. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, 1989.

[14] A. Bloss and P. Hudak. Path semantics. In *Third Workshop On Mathematical Foundations of Programming Language Semantics*, pages 476–489, 1988.

[15] D. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Dept. of Computer Science, 1989.

[16] D. Cann. Retire Fortran? a debate rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[17] D. Cann. Personal communication by electronic mail, February 1993.

[18] E. Chailloux. An efficient way of compiling ML to C. In Peter Lee, editor, *ACM SIGPLAN Workshop on ML and its applications*, pages 37–51. Carnegie Mellon University, 1993. published as Technical Report CMU-CS-93-105.

[19] T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. In *International Conference on Logic Programming*, pages 276–293, 1987.

[20] T. Chuang. *New Techniques for the Analysis and Implementation of Functional Programs.* PhD thesis, New York University, Dept. of Computer Science, 1993.

[21] T. Chuang and B. Goldberg. A syntactic approach to fixed point computation on finite domains. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 109–118, 1992.

[22] W.D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Strategies for implementing continuations. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 124–131, 1988.

[23] W.D. Clinger and J. Rees. Revised^4 report on the algorithmic language scheme. Technical Report CIS-TR-91-25, University of Oregon Dept. of Computer Science, 1992.

[24] M. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation.* The MIT Press, 1989.

[25] R. Cridlig. An optimizing ML to C compiler. In Peter Lee, editor, *ACM SIGPLAN Workshop on ML and its applications*, pages 28–35. Carnegie Mellon University, 1993. published as Technical Report CMU-CS-93-105.

[26] A.J.T. Davie. *An Introduction to Functional Programming Systems using Haskell.* Cambridge University Press, 1992.

[27] A. Deutch. On determining the lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *ACM Symposium on Principles of Programming Languages*, pages 157–168, 1990.

[28] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *1992 International Conference on Computer Languages*, pages 2–13, 1992.

[29] M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation-order and its applications. In *ACM Conference on Lisp and Functional Programming*, pages 242–249, 1990.

[30] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1986.

[31] P. Fradet. Syntactic detection of single-threading using continuations. In *Functional Programming Languages and Computer Architecture*, pages 241–258. Lecture Notes in Computer Science, Springer-Verlag, 1991. Volume 523.

[32] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[33] R. Goldman and R.P. Gabriel. Preliminary results with the initial implementation of qlisp. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, July 1988.

[34] G. H. Golub. *Matrix Computations.* The Johns Hopkins University Press, 1988.

[35] G. H. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, Inc., 1993.

[36] C.K. Gomard and P. Sestoft. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantic Based Program Manipulation (PEPM)*, pages 166–176, 1991.

[37] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Stanford University, Computer Systems Laboratory, 1988.

[38] K. Gopinath. Copy elimination in functional languages. In *ACM Symposium on Principles of Programming Languages*, 1989.

[39] D. Gudeman, Koenraad De Bosschere, and Debray S.K. jc : An efficient and portable sequential implementation of janus. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 399–413, 1992.

[40] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, 1990.

[41] J.C. Guzman. *On Expressing the Mutation of State in a Functional Programming Language.* PhD thesis, Yale University, Dept. of Computer Science, 1993.

[42] J. Hicks Jr. *Compiler Directed Storage Reclamation using Object Lifetime Analysis.* PhD thesis, Electrical Engineering and Computer Science, MIT, 1992.

[43] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[44] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: A functional perspective. In J.H. Fasel and R.M. Keller, editors, *Graph Reduction*, pages 312–327. Lecture Notes in Computer Science, Springer-Verlag, 1986. Volume 279.

[45] P. Hudak. A semantic model of reference counting and its abstraction. In *ACM Conference on Lisp and Functional Programming*, pages 351–363, 1986.

[46] Paul Hudak. Mutable abstract datatypes or how to have your state and munge it too. Technical report, Department of Computer Science, Yale University, 1993.

[47] L Huelsbergen, James R. Larus, and Alexander Aiken. Using run-time sizes of data structures to guide parallel thread creation. In *Proceedings of ACM Conference on Lisp and Functional Programming*, 1994. (to appear).

[48] T. Johnsson. Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture, Nancy, France.* Lecture Notes in Computer Science, Springer-Verlag, September 1985. Volume 201.

[49] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 2–108. Oxford University Press, 1992.

[50] K. Knobe, J.D. Lukas, and G.L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[51] D. Kranz, R. Halstead, Jr, and Eric Mohr. Mul-t: A high-performance parallel lisp. In T Ito and R. Halstead,Jr, editors, *Proceedings of US/Japan Workshop on Parallel Lisp : Languages and Systems*, pages 306–311. Lecture Notes in Computer Science, Springer-Verlag, June 1989. Volume 441.

[52] D. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2):1–35, 1985.

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

[54] E. Mohr, D. Kranz, and R.H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[55] R.S. Nikhil. Id (version 90.0) reference manual. Technical Report CSG Memo 284-1, MIT, 545 Technology Square, Cambridge, MA 02139, August 1990.

[56] R.S. Nikhil, Papadopoulos G.M., and Arvind. *t: A multithreaded massively parallel architecture. In *The 19th Annual International Symposium on Computer Architecture*, pages 156–167. IEEE, 1992.

[57] J.D. Pehoushek and J.S. Weening. Low-cost process creation and dynamic partitioning in qlisp. In *Proceedings of US/Japan Workshop on Parallel Lisp: Languages and Systems*, pages 182–199. Lecture Notes in Computer Science, Springer-Verlag, 1989. Volume 441.

[58] Simon Peyton Jones. *Implementation of Functional Programming Languages.* Englewood Cliffs NY: Prentice Hall, 1987.

[59] Simon L. Peyton Jones and *et al.* The glasgow haskell compiler: A technical overview. In *Proceedings of the UK Joint Framework for Information Technology Technical Conference, Keele*, 1993.

[60] V. Sarkar. *Partitioning and Scheduling of Programs for Multiprocessors.* The MIT Press, 1989.

[61] A.V.S. Sastry and W.D. Clinger. The non-flat array update problem. Unpublished manuscript, July 1993.

[62] A.V.S. Sastry and W.D. Clinger. Parallel destructive updating in strict functional languages. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, 1994. (to appear).

[63] A.V.S. Sastry, W.D. Clinger, and Z.M. Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275, 1993.

[64] D. Schmidt. Detecting global variables in denotational specifications. *ACM TOPLAS*, 7(2):299–310, 1985.

[65] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Boston: Allyn and Bacon, 1986.

[66] P. Sestoft. Replacing function parameters with global variables. Master's thesis, DIKU, University of Copenhagen, October 1988.

[67] P. Sestoft. Replacing function parameters by global variables. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.

[68] R. Sethi. Pebble games for studying storage sharing. *Theoretical Computer Science*, 19(1):69–84, July 1982.

[69] G. Steele. Rabbit: A compiler for scheme. Master's thesis, MIT, 1978.

[70] G. L Steele Jr. and W.D. Hillis. Connection machine lisp: fine grained parallel symbolic processing. In *ACM Symposium on Lisp and Functional Programming*, pages 279–297, 1986.

[71] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[72] V. Swarup, U.S. Reddy, and E. Ireland. Assignments for applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 192–214. Lecture Notes in Computer Science, Springer-Verlag, 1991. Volume 523.

[73] D. Tarditi, P. Lee, and A. Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.

[74] Kenneth R. Traub. *Implementation of Non-strict Functional Programming Languages.* London : Pitman ; Cambridge, Mass. : MIT Press, 1991.

[75] P. Wadler. A new array operation. In J.H. Fasel and R.M. Keller, editors, *Graph Reduction*, pages 328–335. Lecture Notes in Computer Science, Springer-Verlag, 1986. Volume 279.

[76] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[77] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990. Presented at IFIP TC2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, April 1990.

[78] P. Wadler. The essence of functional programming. In *ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[79] David Wall. Global register allocation at link-time. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, 1986. Published as SIGPLAN Notices 21 (7).

[80] E. Wang and P. Hilfinger. Analysis of recursive types in lisp-like languages. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 216–225, 1992.

[81] N. Wirth. *Programming in Modula-2.* Berlin: Springer-Verlag, 1982.

[82] David S. Wise. Design for a multiprocessing heap with on-board reference counting. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–304, 1985.

[83] Z Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic, 1991.