

Implementing Compiler Optimizations Using Parallel Graph Reduction

by

Joanna L. Kulik

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

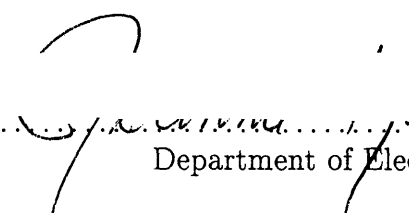
Master of Science


at the

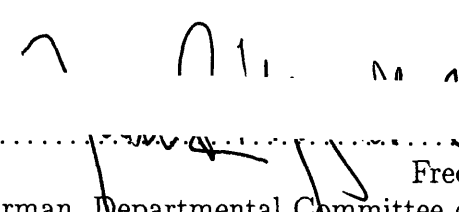
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

© Massachusetts Institute of Technology 1995. All rights reserved.

Author.....
Department of Electrical Engineering and Computer Science
January 13, 1995

Certified by.....
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by.....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Eng.

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 13 1995

LIBRARIES

Implementing Compiler Optimizations Using Parallel Graph Reduction

by

Joanna L. Kulik

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The Id-in-Id compiler is an Id program that compiles Id programs into dataflow graphs. The output of this compiler currently runs on a graph simulator called GITA, but with minor adaptations it will also run on the Monsoon dataflow machine, and with more extensive adaptations the output can run on other parallel machines. Since Id is an implicitly parallel language, the Id-in-Id compiler will itself run on a parallel machine – providing perhaps the most challenging application with unstructured parallelism.

This thesis investigates the use of parallel graph reduction techniques in compiler optimizations for lambda calculus based languages. The Id-in-Id compiler employs such techniques for machine independent optimizations. Among the innovations of the compiler is DeBruijn chaining, a novel representation of the lambda calculus used in implementing free variables of lambda terms. Another novel technique, the Memobook, facilitates efficient and parallel graph traversal. Memobooks make use of M-structures, an imperative feature of the Id language. Disjoint-set data-structures that can be unified in parallel have also been developed. These data-structures are useful for performing mutations on a program graph. The thesis includes both an analysis of the effectiveness of parallel graph-reduction in compiler optimization and a discussion of their future potential in software development.

Thesis Supervisor: Arvind

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Shail Aditya, YuLi Zhou, and Satyan Coorg for generously contributing their help and advice during the various stages of this project. I would also like to thank Professor Arvind for the genuine interest he took in the project and for guiding the structure and emphasis of this thesis. Finally, I would like to give thanks to my parents, who saw me through every revision. This thesis would not have been possible without their unending support and encouragement.

In memory of Estelle Kulik

Contents

1	Introduction	11
1.1	Overview of the Area	12
1.1.1	Improving Efficiency of Functional Language Implementations	12
1.1.2	Compile-time Graph Reduction in the Id-in-Id Compiler	15
1.2	Organization of this Thesis	16
2	Background and Problem	18
2.1	The Programming Language Id	18
2.2	Optimization of Id	19
2.2.1	The Intermediate Language KId	21
2.2.2	Kid Reduction Rules	22
2.3	Problems and Purpose	24
3	Representing Programs as Graphs	25
3.1	Kid Graphs	25
3.2	The Problem with Lambdas and Free-Variables	27
3.3	DeBruijn Chains	29
4	Data-structures for Performing Reductions	44
4.1	Indirection Nodes	45
4.2	Unions of Disjoint-Sets	46
4.3	Comparing Indirection-nodes and Disjoint-Set Unions	49
5	Keeping Track of Reductions	54
5.1	Simple Graphs Tied Into Complicated Knots	54
5.2	Memobooks	55
5.3	Memobooks in Parallel Graph Algorithms	57
5.3.1	Copying Functional Graphs	57
5.3.2	Constant-folding Cyclic Graphs	58

5.3.3	Summing the Values in a Mutable Graph	59
6	Optimization Strategies	67
6.1	Strategies for Optimizing with a Single Reduction Rule	67
6.2	Adding Reduction Rules	69
6.3	Lenient Reduction in the Id-in-Id Optimizer	70
7	Results	74
7.1	The Id-in-Id Compiler Running on a Sun-Workstation	75
7.2	Memoized Fibonacci and Factorial on GITA	76
7.3	Comparing Functional Reductions to Disjoint-Set Unions on GITA	78
8	Conclusions and Future Work	82
A	Memobook Implementation	88
B	Code for Reduction Tests	91

List of Figures

2-1	Grammar for the kernel language KId.	21
2-2	The CSE optimization rule.	23
2-3	Optimizing reduction rules for algebraic-identity optimizations.	23
3-1	Grammar for Kid Graph nodes.	26
3-2	A KId Graph block	27
3-3	Delimiting the extent of lambda-bodies.	29
3-4	A lambda-calculus graph and its supercombinatorial equivalent.	30
3-5	A program and its lambda-lifted counterpart.	30
3-6	The graphical representation of a program using lists to denote the extent of lambdas.	31
3-7	Reduction of a program using lists to denote the extent of lambdas.	32
3-8	The DeBruijn-chain representation of a program graph.	33
3-9	Reduction of an expression using Debruijn chains.	36
3-10	A program represented without special support for free-variables.	37
3-11	An expression represented using Debruijn numbers	37
3-12	An expression represented using DeBruijn chains	38
3-13	Beta-substitution algorithm for programs represented using DeBruijn numbers.	39
3-14	Reduction of a program represented using Debruijn numbers.	40
3-15	Beta-Substitution algorithm for programs represented using DeBruijn chains.	40
3-16	Reduction of a program represented using Debruijn chains.	41
3-17	The difference between DeBruijn chains and DeBruijn numbers.	42
3-18	Grammar for Kid Graph nodes.	43
4-1	Changing a functional graph using selective copying.	45
4-2	Beta-reduction using copying.	46
4-3	Beta-reduction using indirection-nodes.	47
4-4	Multiple beta-reductions can lead to chains of indirection-nodes.	48
4-5	Operations on disjoint-set forests.	49

4-6	An example of path-compression in disjoint-set forests.	50
4-7	The operations UNION and REPRESENTATIVE for disjoint-set forests.	50
4-8	The modified REPRESENTATIVE operation, using path-compression, for disjoint-set forests.	51
4-9	Using disjoint-sets to implement beta-reduction.	52
4-10	Revised UNION and REPRESENTATIVE operations, using ∇ nodes to represent members of sets.	53
4-11	A comparison of the indirection-node and disjoint-set unions approaches.	53
5-1	Possible outcomes of parallel reduction.	61
5-2	Producer-consumer interaction with memobooks.	62
5-3	The operations PRODUCE_MEMO and READ_MEMO.	63
5-4	The COPY_TREE and COPY_GRAPH operations for copying trees and graphs.	63
5-5	Two cyclic graphs.	64
5-6	The CONSTANT_FOLD operation for constant-folding graphs.	64
5-7	A modified CONSTANT_FOLD operation that uses paths to detect cycles.	65
5-8	Two functions, SUM_GRAPH and PIPELINE_SUM_GRAPH, for summing values stored on a graph.	66
6-1	Optimization of Kid programs using an interpreter.	68
6-2	An example of too much compile-time interpretation.	68
6-3	Algorithm for performing a single pass.	71
6-4	Algorithm for performing two single-passes.	72
7-1	Un-memoized factorial, fact1 , and memoized factorial, fact2	77
7-2	Un-memoized fibonacci, fib1 , and memoized fibonacci, fib2	78
7-3	A graph before and after it has been reduced.	80

List of Tables

7.1	Performance of programs compiled by Id-in-Id and Id-in-Lisp compilers.	76
7.2	Performance results of fact1 and fact2	77
7.3	Performance results of fib1 and fib2	78
7.4	Ten input graphs and their characteristics.	80
7.5	Performance results of different transformation schemes.	81

Chapter 1

Introduction

The Id-in-Id compiler is an Id program that compiles Id programs into dataflow graphs. The output of this compiler currently runs on a graph simulator called GITA, but with minor adaptations it will also run on Monsoon dataflow machines, and with more extensive adaptations the output can run on other parallel machines. Since Id is an implicitly parallel language, the Id-in-Id compiler will itself run on a parallel machine—providing perhaps the most challenging application with unstructured parallelism.

Efficiency is the great challenge facing any developer of a compiler for functional languages. Functional programs typically run slowly, and they have a reputation for inefficiency. The compilation process usually proceeds in three stages, and the compiler can try to improve, or optimize, the efficiency of the output code at all three stages of compilation. First, a compiler can improve the programmer's source code. Second and more important, it can remove inefficiencies created during earlier translation phases of compilation. This kind of optimization is especially important for functional language compilers because such compilers translate languages especially often. Finally, the compiler can improve the low-level code it creates, so that the code can run as fast as possible on the target machine. Conventional compilers that try to optimize their code in each of these areas often produce code that executes 2 to 3 times faster than compiler code produced without optimizations. Functional language compilers can produce code that executes up to 20 times faster than code produced without optimizations.

The challenge faced in designing the Id-in-Id compiler was to incorporate these features into a compiler written to run on a parallel machine. The thesis describes an approach for meeting this challenge in the second stage of Id-in-Id compilation, where machine-independent optimizations occur. It focuses on methods for carrying out well-known functional program optimizations, rather than on the development of new optimizations. The approach presented here builds on two important ways of improving the efficiency of functional language implementations (a) compile-time reduction

of programs and (b) use of graph reduction. This introductory chapter gives an overview of the research area and also provides an outline for the rest of the thesis.

1.1 Overview of the Area

Although the field of functional languages is still in its infancy, the strong points of functional languages are well-known [9]. Most computer scientists agree that functional programs are easier to write than conventional imperative programs. This programmer-friendliness does not come from what functional programs lack—such features as imperative assignment statements, side effects, and control of flow—but it stems rather from what functional programs uniquely possess. They alone have higher order composition mechanisms that help programmers to “program in the large,” or to think big.

Computer scientists also agree about another important feature of functional programs. Functional programs are easier than conventional programs to reason about. Statements as simple as the ones used in basic arithmetic describe the meaning and execution of functional programs. It is no wonder therefore that functional programming appeals to theorists as much as it does to application programmers.

These advantages would be enough to justify further work on functional languages, but there is an additional reason to be concerned with functional programming today. Functional languages seem especially suited to parallel machines. This is because the functional languages are based on the language of mathematics, which does not include notions of time or state. Functional languages, like mathematical formulas, are implicitly parallel. With functional languages, programmers do not have to deal with state-notions or synchronize operations among processors. They can instead concentrate on the big picture of program design and leave the smaller but mind-numbing details for automatic solution by the language implementation.

The merits of functional programming used to be offset by one serious disadvantage. Functional programs ran very slowly. Programs written in the non-strict functional languages, for example, sometimes ran as much as 100 times slower than comparable programs written in procedural languages such as C [19]. The slowness of functional languages undoubtedly affected their acceptance. Why write an important program in a functional language if it will run at one one-hundredth the speed of a program written in a procedural language?

1.1.1 Improving Efficiency of Functional Language Implementations

Methods for speeding up the implementation of functional languages therefore continues to be an active research area today. The current research falls into two major categories, the design of effective compilers and the design of appropriate execution vehicles. This thesis deals with the former area

and presents techniques for carrying out optimizations during compilation of functional programs. Two important ways of improving the efficiency of functional languages, in both compilation and execution, serve as the foundations for these techniques. One of these is compile-time reduction, in which the compiler reduces the program as much as possible at compile-time to cut down on work at run-time. The other important approach is graph reduction, in which identical lambda-calculus expressions are represented by simple pointers rather than copies of the original expressions to cut down on redundant computations.

Reducing Programs at Compile-Time

Compilers receive high-level source programs as input and produce low-level machine code as output. It is usual therefore to think of compilation as having three phases: a beginning phase of input, a middle phase of transformation, and a final phase of output. At each phase, the compiler represents the program using different data-structures. The job of each phase then is to transform the program from one representation to another.

Compilers can also be thought of simply as translators that connect programmers to machines. They translate programs that are understood by programmers to programs that can be understood by machines. When viewed in this way, compilation does not look like a series of data-structure transformations, but rather it looks like a series of language translations. Compilation ends with the final translation of the program into machine language.

Ariola and Arvind used this conception of a compiler to create an abstract scheme for compiling functional languages [27]. Their scheme posits a series of programming languages: The source language is the first of these, and a machine-level language is the last. The scheme also contains reduction rules for translating from one language in the series to the next. The reduction rules do not give exact computational algorithms for translations. Instead, they give the characteristics of the program before and after each rule is applied. Their reduction system is implementation-independent and general enough for use in proofs about properties of program translations, but the system is also specific enough to provide a blueprint for compiler writers to construct a working compiler.

Arvind and Ariola's reduction system includes rules for the machine-independent optimizations that occur at compile-time before the final translation of the program into low-level code. It is important to note that the Arvind and Ariola's optimizing rules do not translate programs from one language to another. Instead, they improve, or optimize, the program's use of the language. An essential component of their optimizing reduction rules is that their use of a reduction technique called graph-reduction.

Optimizing programs using reduction at compile-time is not a new idea. The same idea is the basis of compile-time interpretation, a method that has been successfully included in the optimization

phases of many functional language compilers. Interpretation at compile-time can yield important compilation information that might otherwise be obtained only at run-time. Compile-time interpretation is particularly appealing because the inefficiency of interpretation at compile-time is offset by the fact that it is performed only once for a program, and costs are therefore amortized by long-term run-time savings.

There primary difference between compile-time interpretation and Arvind and Ariola's system is that compile-time interpretation uses only one or two rules to reduce functional programs at compile-time, whereas Arvind and Ariola's system uses many. Arvind and Ariola's reduction rules cover many different optimizations, including CSE, code-hoisting, loop-peeling and unrolling, and inline-substitution. Though the two approaches are different, it is illuminating to note the both methods rely on the same idea, applying reduction rules at compile-time to optimize programs. Compile-time interpretation is an accepted tool for optimizing functional programs. A system that uses more reduction rules, like Ariola and Arvind's, should be equally successful as compile-time interpretation, if not more, at optimizing programs. Again, the cost of applying additional rules at compile-time is amortized by long-term run-time savings.

Graph Reduction

Although functional languages differ in their syntactic styles, they are fundamentally all alike. The basic building block of all functional languages is the function application, and the language underlying most functional languages is the lambda calculus. When the syntactic dressing is removed from functional languages, what is usually left is the lambda calculus. Landin therefore refers to functional languages as lambda calculus plus syntactic sugar [11].

Nonetheless, most writers on the subject divide the functional languages into types. The most common way of categorizing functional languages is by their method of evaluating functions. Strict languages are those that require the complete evaluation of all arguments before a function call. Non-strict languages require evaluation of arguments only when they are needed. Today, this is the primary distinction between these two types of languages. It used to be the case that these languages were also separated by another characteristic: programs written in non-strict languages executed much, much slower than those written in strict languages.

Graph-reduction, a technique originally developed by Wadsworth [25], helped to narrow the gap between non-strict and strict languages. The difficulty with executing non-strict programs is delaying the execution of argument expressions without making multiple copies of the expressions. In graph-reduction, computations and their interdependencies are represented using pointers. With such sharing of computations, wasteful copying and duplication of computations can be avoided. Wadsworth realized that, with graph reduction, an interpreter that evaluates arguments only when the arguments are needed can be as efficient as any other kind of interpreter.

Unfortunately, Wadsworth limited his discussion of graph reduction to lambda-calculus interpreters. Most modern language implementations, however, do not use interpreters to execute programs. Instead, compilers compile programs down to low-level code so that they can be executed directly on conventional Von Neumann machines. Furthermore, Wadsworth assumed that the cost of manipulating and copying large graph data-structures would be negligible. Those with more hands-on experience with program implementations knew that this assumption was untenable. Nonetheless, some were able to see in Wadsworth's invention of graph reduction an invaluable technique for speeding up the slow execution speeds of programs written in non-strict languages.

Turner was the first to show that it was indeed possible to use graph-reduction in a workable implementation for non-strict languages [24]. Turner's idea was to perform reductions on programs represented as combinator-graphs, which seemed to be more manageable than Wadsworth's lambda-calculus graph-representations. Johnsson followed up on this work by using Turner's ideas along with compilation techniques that were state-of-the-art at the time [10]. In Johnsson's scheme, a compiler translated input programs to super-combinator graph data-structures for execution on an abstract graph-reduction machine called the G-machine. Johnsson's G-machine served as the model for most of today's efficient implementations of non-strict languages. In fact, the term "graph-reduction" is today used almost exclusively to describe G-machine-like implementations.

Why does graph-reduction, a technique that has had such a great impact in the execution of functional programs, have any bearing on the optimization of programs at compile-time? The closer that a program representation can accurately reflect the constraints of the program's execution model, the better the representation is for program optimization. Most functional program implementations represent programs as graphs at run-time so optimizers should too. When optimizing reduction rules are applied to programs represented as graphs, the result is graph-reduction. Ariola and Arvind have shown that graph-reduction is not only beneficial, but necessary for optimization using their reduction rules. Reduction rules for some optimizations, such as CSE, cannot be formulated without the use of graph-reduction.

1.1.2 Compile-time Graph Reduction in the Id-in-Id Compiler

At its core, Id is very much like other functional languages. Its primary building blocks are functions and function applications. Like other functional languages, it also supports higher-order and curried functions. It includes a rich set of data-types and static type-checking for program security. Finally, it provides a single-assignment syntax and other syntactic conveniences for writing programs. Nikhil provides a complete specification for the language Id in [13].

Unlike most functional languages, Id also provides fine-grained implicit parallelism. Functional programming advocates sometimes say that all functional programs are implicitly parallel. Most implementations of functional languages, however, restrict language semantics so that programs can

be efficiently compiled for a sequential machine. Programs written in these languages are implicitly parallel only at a coarse level. Id does not place sequential restrictions on its semantics. In Id, the parallelism that purportedly exists in all functional languages remains unspoiled [23].

The Computation Structures Group’s Id-in-Id compiler follows the compilation scheme outlined by Arvind and Ariola in [27]. Id-in-Id compilation proceeds as a series of phases, each phase translating and optimizing programs represented using different languages using reduction rules. The heart of the compiler is its optimization phase. To represent programs for optimizations, the Id-in-Id compiler turns redundant computations into shared dependencies in a graph. It performs the actual optimizations by applying Ariola and Arvind’s reduction rules to these program graphs. Because the compiler is written in Id, it performs the graph-reductions in parallel.

1.2 Organization of this Thesis

The Computation Structures Group has produced the first compiler to carry out graph reductions in parallel at compile-time. This thesis describes the system of graph-reduction that makes compiler optimization in parallel possible. It examines existing methods for graph reduction, identifies obstacles to using these methods in parallel compilation, proposes ways to overcome the obstacles, and describes the use of the proposed solutions in the Id-in-Id compiler.

Chapter 2 describes the foundations of this thesis in previous work. This work includes the development of the Id parallel-programming language and the Kid reduction system for optimizing programs written in Id.

Chapter 3 presents Kid Graphs, a graphical representation of Kid programs for optimizations. Kid Graphs use novel technique that involves building chains of pointers, called DeBruijn chains, to efficiently represent free-variables inside a graph.

Chapter 4 addresses the problem of performing reduction transformations on graphs. It examines several possible approaches to program transformation, including the use of functional data-structures and indirection-nodes to represent programs. It settles on one solution, the use of parallel disjoint-set union operations to perform program transformations, for use in the Id-in-Id optimizer.

Chapter 5 describes the problem of synchronizing multiple transformations on graphs. It presents a solution to this problem, called memobooks. Memobooks are general purpose utilities for synchronizing operations on any group of objects. This chapter describes how memobooks can be used in a variety of parallel graph applications and also how they are used in the Id-in-Id optimizer.

Chapter 6 addresses the important question of what reduction strategy is best for optimization. Researchers in classical program reduction and researchers in compiler development differ in their solutions to this question. This chapter also describes the overall strategy used by the Id-in-Id compiler to optimize programs. It also shows how DeBruijn chains, disjoint-set unions, and memobooks

fit into this strategy.

Chapter 7 summarizes the performance of the Id-in-Id optimizer on several platforms. It shows how optimization performed by the Id-in-Id compiler compare to those performed by another compiler, the Id-in-Lisp compiler. It also assesses the performance of memobooks and disjoint-set unions, when used in several applications.

Chapter 8 summarizes the results of this research and presents suggestions for further work in the area. Although the Id-in-Id compiler advances developments in parallel graph reduction substantially, more work in the areas of compiler technology and parallel implementation is needed to make its concepts become state-of-the-art in design of compilers for functional languages.

Chapter 2

Background and Problem

Work done during the past decade by the Computation Structures Group provided the foundation for my work on the Id-in-Id compiler. Among the contributions of this group were (a) the development of the Id language; (b) the development of Kid, a related intermediate language more suited to compiler optimization; and (c) the development of a reduction system for examining the correctness of optimizations made on Kid programs. This chapter describes each of these contributions and then gives an overview of the problems addressed in the rest of the thesis.

2.1 The Programming Language Id

The input language for the compiler and the language in which the compiler is written is Id. Whiting and Pascoe provide a succinct description of the development of this language in their history of dataflow languages [26]. Nikhil provides a complete specification for the language in [13].

Like other functional languages, Id supports function application, the basic building block of all functional programs. For example, a function that adds its argument to itself may be defined in Id as:

```
def DOUBLE x = (x + x)
```

(DOUBLE 3) is then an Id program to double the value of three, and (DOUBLE (DOUBLE 3)) a program to quadruple it.

Higher order functions can be built from existing abstractions in Id. For example, TWICE is a function that takes two arguments, *f* and *x*. It applies the function *f* to *x*, then applies *f* to that result. TWICE is therefore defined in Id as follows:

```
def TWICE f x = f (f x) ;
```

TWICE can also be used to define another function, QUADRUPLE, that quadruples the value of its

argument. This example shows Id's support for currying, a technique introduced by [21] that is useful for writing programs succinctly:

```
def QUADRUPLE x = TWICE DOUBLE x ;  
or  
QUADRUPLE = TWICE DOUBLE ;
```

More complex expressions can be formed in Id by binding names to expressions and collecting the bindings together in a let block like the following:

```
{ a = 4 * 15 ;  
  b = 3 + 2 ;  
  In (a + b) / (a * c)  
}
```

The binding statements in this block should not be read like assignment statements in imperative programs. They do not specify a sequence of events. Their sole purpose is to specify the dependencies in the input program. Besides let blocks, Id also includes other features common to functional languages. These include support for polymorphism, static type-checking, and data-structures such as arrays, tuples, and algebraic-types.

A program written in a functional language can be represented as a set of operator nodes interconnected by a set of data carrying arcs. In such a representation, there is no sequential control, and each operator is free to execute when its data arrive. This is true for all functional programs. However, most functional languages impose a sequential ordering on the execution of operations. In Id, these operators are free to execute in parallel.

Because Id programmers do not specify a parallel ordering in their programs, this sort of parallelism is often called *implicit parallelism*. Other parallel languages require programmers to explicitly manage control and data in a parallel machine. These languages are often tailored for writing programs of a specific type, such as scientific applications, to make the task of explicit management easier. Id, on the other hand, is a general-purpose programming language. Because of its fine-grained parallelism and expressive power, it is both an appropriate language in which to write a parallel optimizer and an appropriate target languages for compiler optimization. Traub has already demonstrated the dramatic effect that optimizations have on programs written in Id. The effectiveness of an Id optimizer of Id programs, however, remains to be demonstrated.

2.2 Optimization of Id

Id is a high-level language whose features are designed to help programmers express complex ideas simply and succinctly. Before programs written in high-level languages can be executed, they must be translated into machine-language statements, and before this translation can take place, the

high-level statements must be broken down into smaller statements that correspond to the machine-language statements. An Id program at this phase of compilation is at a level intermediate between its high-level and machine-language form. Optimization of a program at this intermediate phase can pay rich dividends.

Traub has concluded that the intermediate language used in a functional compiler should be grounded mathematically in an abstract reduction system [23]. He believes, however, that an adequate kernel language would meet several other criteria:

1. The language would be non-strict and thus serve as a model of both lenient and lazy evaluation.
2. It would have primitive constructs for features ordinarily treated as primitive by functional language implementations.
3. It would be a minimal language without “syntactic sugar.”
4. Its operational semantics would accurately model the behavior of realistic functional language implementations.
5. The language and its operational semantics would have a formal structure that makes it easy to study the relation between the individual computations that comprise the execution of a program.

The core language for all functional languages is the lambda-calculus, a small language with a precise semantics that can be described using reduction rules. The lambda-calculus is therefore an obvious candidate for an intermediate, or kernel, language in a functional compiler. Traub has concluded, however, that the lambda calculus cannot serve this function. The language fails very obviously on Traub’s Criterion 2. Its only primitive constructs are functions and function applications; other primitive features (e.g., numbers) must be simulated in the lambda calculus through the use of functions.

Furthermore, it is not possible to express some fundamental optimizations as transformations on lambda-calculus programs. Common-subexpression-elimination, or CSE, is one example. The idea behind CSE is to eliminate redundant computations by sharing them. If a sub-expression C is common to two different expressions A and B , then A and B can share C ’s computation. Conventional compilers perform optimizations like CSE because modern functional language implementations must share computation wherever possible. Sharing of computations is critical for the efficiency of these implementations.

The lambda-calculus is a powerful language that captures many ideas about computations, but unaided it cannot represent even the simplest notions about sharing of computations. It is not possible, for example, to formulate the idea “let A and B share C ’s computation” in the pure lambda-calculus. Formulating an optimization rule to express this same idea is likewise impossible.

Compile-time detection of sharing is critical in optimization, and the lambda-calculus therefore also fails on Traub's Criterion 4.

2.2.1 The Intermediate Language KId

Designers of functional compilers have therefore used other intermediate languages, such as IF1 [22], FLIC [17] and Term Graph Rewriting [4] systems, to model functional language implementations. The Computation Structures Group have devised three intermediate languages based on their experience with Id programs. The first to be devised was Traub's Functional Quads [23], which provides a "three address" syntax for program graphs. Next came Ariola and Arvind's P-PTAC, or Parallel Three-Address Code [1]. The most recent of these languages is Kid, or Kernel Id [27].

Kid also maintains the features of dataflow graphs and representing sharing of expressions in programs. Kid is not quite as low-level as these other languages, however, making optimization of Kid programs easier. Its grammar is given in Figure 2-1. KId programs are similar to Id programs in most respects, but the Kid language lacks some of the extraneous syntactic notation found in Id. The name KId, in fact, stands for "Kernel Id" and is meant to suggest that Kid programs contain the significant part, the kernel, of Id programs.

PF_1	::= Allocate
PF_2	::= + - ... Equal? Select Apply
SE	::= <i>Variable</i> <i>Constant</i>
E	::= SE $PF_n(SE, \dots, SE)$ Bool_Case (Se, E, E) <i>Block</i> $\lambda(x_1, \dots, x_n).E$
<i>Block</i>	::= $\{ [Statement;]^* \text{ In } SE \}$
<i>Statement</i>	::= <i>Binding</i> <i>Command</i>
<i>Binding</i>	::= <i>Variable</i> = E
<i>Command</i>	::= P_store (SE, SE, SE) Store_error \top_s

Figure 2-1: Grammar for the kernel language KId.

The most important feature to note in the Kid grammar is the use of block notation to express shared computations in programs. In Kid, a block is a collection of statements that bind meta-variables to expressions. The meta-variables indicate sharing of expressions. Expressing shared computation makes it possible to: a) perform important optimizations b) maintain side-effects correctly. For example, to represent the idea that the expressions $(+ (* a b) 3)$ and $(- 1 (* a b))$ can share the computation of $(* a b)$, one could add a binding of the form $c = (* a b)$ to the

program and then transform the original expressions to $(+ c 3)$ and $(- 1 c)$.

The command construct given in Figure 2-1 is used to represent side-effecting commands in Kid programs. Side-effects destroy referential-transparency, a property of programs that ensures that an expression and a copy of the expression are indistinguishable from each other. In Id, an expression and a copy of the expression *are* distinguishable from each other. For example, if two expressions, A and B , depend upon a side-effect expression C , this dependency must be represented by using pointers to the shared-expression. If A and B contained copies of the expression C , the side-effect would occur twice at run-time. Blocks make it possible to create and maintain shared dependencies, and therefore side-effects, correctly.

2.2.2 Kid Reduction Rules

The real proof of the power of Kid is in its reduction rules: Each Id optimization mentioned by Traub in [23] can be expressed in terms of Kid reduction rules. The optimizations are: inline substitution, partial evaluation, algebraic rules, eliminating circulating variables and constants, loop peeling and unrolling, common subexpression elimination, lift free expressions (loop invariants), loop variable induction analysis, and dead-code elimination. The full set of optimizing Kid reduction rules is given in [27].

Figure 2-2 (a) shows how to formulate a CSE optimization rule for Kid programs. The top part of the rule specifies the precondition of the rule. The bottom part specifies the reduction to be performed when this precondition is met. The rule given in Figure 2-2 specifies that if some meta-variable X is bound to an expression E and meta-variable Y is not bound to X , then any binding of the form $Y = E$ should be reduced, or rewritten, to the binding $Y = X$. Figure 2-2 (b) shows the reduction of a Kid program, with blocks, using the CSE rule. The program has been improved because the new program computes $(+ b c)$ once, whereas the old program computed it twice. Reductions like this are called *graph-reductions*. Wadsworth coined the term to refer to reduction of programs as graphs. Figure 2-2 (c) shows how the reduction given in (b) is actually a reduction on the dependencies of the program when represented as a graph.

There are several advantages to formulating optimizations as reduction rules. Reduction rules provide a precise notation for reasoning about program transformations. Ariola has used the abstract reduction rules of the Kid language, for example, to show that most Kid optimization rules are confluent. A confluent optimization rule is one that is guaranteed to reduce a program to its normal form after repeated application. The order in which the rule is applied to the program does not matter. This means that optimization rules can be applied to programs in a variety of orders, including parallel order, to reach optimal forms. Partial-evaluation and inline-substitution are the only rules that are not guaranteed to reach the normal-form of a program if it exists.

Ariola has also used the Kid reduction rules to prove the correctness of certain well-known op-

$$(a) \frac{X = E \wedge Y \neq X}{Y = E \rightarrow Y = X}$$

(b) { a = (+ b c) ;
d = (+ b c) ;
In
(* a d) } ;

{ a = (+ b c) ;
d = a ;
In
(* a d) } ;

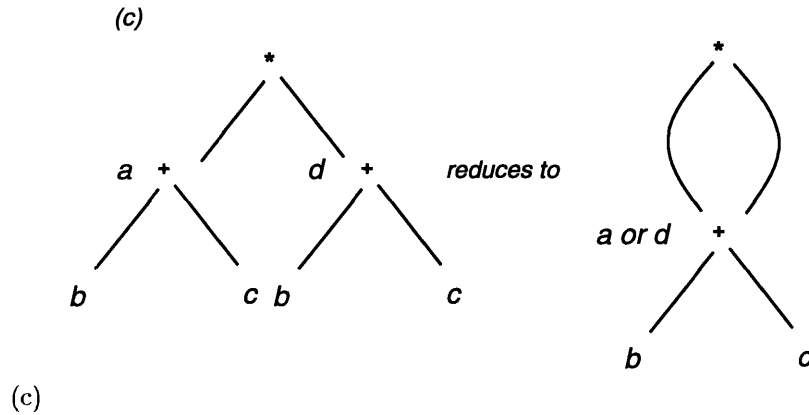


Figure 2-2: The CSE optimization rule for Kid programs (a) and its application to a Kid program (b). CSE reductions can also be depicted pictorially, as in (c).

timizations. An optimization is said to be correct if optimized and non-optimized versions of the program produce the same result. Both correct and partially-correct optimizations reduce expressions that have a normal form to other expressions that have the same normal form, but correct and partially correct optimizations differ in their effect on expressions that do not have a normal form, or Ω expressions. Correct optimizations reduce Ω expressions to Ω expressions, but partially correct optimizations may reduce Ω expressions to expressions that have a normal-form. Several algebraic-identity optimizations are shown in Figure 2-3. These optimizations reduce the number of arithmetic operations in a program by following the identity laws of algebra. The first two of these optimizations are correct. The third is only partially correct because it would reduce the expression $(+ * \Omega 0)$ to 0.

$$\begin{aligned} (+ E 0) &\rightarrow E \text{ correct} \\ (* E 1) &\rightarrow E \text{ correct} \\ (* E 0) &\rightarrow 0 \text{ partially-correct} \end{aligned}$$

Figure 2-3: Optimizing reduction rules for algebraic-identity optimizations. Optimizing reduction rules for algebraic-identity optimizations. The third rule is only partially correct because it would reduce the expression $(+ * \Omega 0)$ to 0.

Finally, it is possible to describe optimization precisely and cleanly using reduction rules. This makes reduction rules appealing not only for use in mathematical proofs. It also makes reduction very appealing for use in the design of real optimization systems. Reduction rules can represent the essential features of optimizations without being biased toward a particular implementation.

2.3 Problems and Purpose

Arvind and Ariola have set the stage for the development of a parallel optimizer for Id. They have described an intermediate language that captures the essentials of Id programs. They have also given precise rules for performing compiler transformations. They have proven that correct programs can be generated by applying these optimization rules to Kid programs in parallel. Still, to draw a complete picture of a parallel optimizer, we need to solve some additional problems.

First, we need to develop a way of representing Kid programs as graphs. A Kid program is a textual representation of a program graph. A compiler needs to manipulate actual program graph data-structures, however, not text. With simple programs (such as the one given in Figure 2-2), translation from a program into graphical form is straightforward. But with complex programs, difficult questions arise. What happens to variable names in graphs? How are blocks to be represented?

When programs are represented as graphs, a second set of problems arise. Graph-reduction involves the repeated transformation of graphs. Again, the simplicity of transforming simple programs as text or pictures is misleading. How can these transformations be carried out efficiently using real program memory? Functional language implementations use a technique called indirection nodes to perform transformations efficiently during graph reduction, but is this the best approach?

A third set of problems arise in synchronizing reductions in a parallel environment. It is necessary to keep some record of actions that take place concurrently in a parallel environment to prevent collisions. In the Id language, I-structures are the logical place for keeping such records. But how can I-structures be used in this job?

Finally, where do optimizations stop? To answer this question, we must have some sort of conception of what an optimized program looks like, and we must also have a strategy for optimization. Previous research in program reduction and compiler techniques differ in this matter, which should apply to a program reduction compiler?

The next four chapters of the thesis deal with these practical issues.

Chapter 3

Representing Programs as Graphs

The basis for the Id-in-Id optimizer is the Kid optimization system described by Arvind and Ariola. The rules in this system apply to program code in the Kid language. The actual Id-in-Id optimizer optimizes programs that are represented as graph data-structures called Kid Graphs. This chapter discusses the design of Kid Graphs. Section 3.1 gives an overview of Kid Graphs and describes their correspondence to Kid programs. Section 3.2 discusses a major obstacle to representing Kid programs as graphs: the representation of lambda bodies and the variables that they use. Section 3.3 presents a solution to this problem, a novel technique called DeBruijn chains, and discusses how DeBruijn chains can be used in Kid Graphs.

3.1 Kid Graphs

A Kid Graph is a uni-directional, distributed graph representation of a Kid program. Kid graphs are uni-directional because all the arrows in the graphs point in only one direction. If parent nodes in a uni-directional graph keep pointers to their children, then the children do not keep pointers to their parents. Kid graphs are distributed representations because the nodes are not kept in one central location. A distributed graph is similar to a list data-structure: the only way to obtain all of its elements is to traverse it from a single node, called the graph's *root*. The main advantage in using distributed graph representations is that it simplifies garbage collection. If a node in a distributed graph is changed so that it is no longer referred to by a parent node, it can be automatically reclaimed for re-use by a garbage collector. In a centralized graph representation, explicit garbage collection is needed since one reference to every node in the graph remains through all changes. The question of introducing cycles into Kid Graphs is addressed in a later chapter.

Each node in a Kid Graph is a record. All nodes contain tags that indicate node-type and usually some additional data, such as pointers to other nodes. These pointers form the dependencies of the graph. Figure 3-1 gives the grammar for Kid Graph node-tags. Although the Kid Graph tags form

a grammar, they can not be used to write syntactic programs. In order to illustrate the practical matters of manipulating Kid Graphs, additional notation is needed. Throughout the rest of the thesis, pictures will be used to illustrate Kid Graph programs. Figure 3-2 shows a Kid program and a picture of its corresponding Kid Graph. The circles in the picture correspond to node records and the contents of each circle indicates the node's tag. If node a contains a pointer to node b , it will be depicted as a line originating from the bottom of a 's node to the top of b 's node. The highest node in the graph is the root of the graph.

PF_1	$::=$	Allocate
PF_2	$::=$	$+$ $-$ \dots Equal?
Exp	$::=$	<i>Variable</i> <i>Constant</i> $PF_n(Exp, \dots, Exp)$ Bool.Case (<i>Exp</i> , <i>Lambda</i> , <i>Lambda</i>)
$Lambda$	$::=$	$\lambda(x_1, \dots, x_n).Block$
$Block$	$::=$	$\{[Root;]^* \text{ In } Exp\}$
$Root$	$::=$	<i>Exp</i> <i>Command</i>
$Command$	$::=$	P_store (<i>Exp</i> , <i>Exp</i> , <i>Exp</i>) Store_error \top_s

Figure 3-1: Grammar for Kid Graph nodes.

One difference between Kid Graphs and Kid is in the use of blocks. Figure 3-2 shows a KID program that contains a block and a Kid Graph of that program. Two kinds of information contained in the program block are not represented in the Kid graph. First, the Kid block contains bindings that tie variables to expressions. For example, the binding $d = (x + b)$ links the variable d to the expression $x + b$. The compiler uses these links to create graphs; it would be superfluous to note the bindings explicitly on graphs. Second, Kid blocks contain scoping information since each block represents a separate lexical scope. It is also unnecessary to keep track of such scoping information in Kid graphs. Long before a compiler converts a program to a graph, it analyzes lexical scopes and changes variables to reflect this analysis.

Because blocks are no longer required for denoting lexical scopes, blocks exist in Kid Graphs only to collect computations for lambdas. They collect information about potentially side-effecting commands, such as $A[x] = c$ and $(g \ x)$. When such commands appear in a block, they are called the *roots* of the block. Second, a block collects information about the expression to which the lambda evaluates, called the *result* of the block. Kid graphs must keep pointers to these expressions. Pictorially, if a block does not contain any roots, the block node itself will be omitted from the picture and replaced with a pointer that points directly to its result, as in Figure 3-3.

This use of blocks in Kid graphs pays a handsome dividend: It simplifies dead-code elimination.

When a side-effect-free expression is no longer referenced in a program, the expression can be eliminated from the program. Because side-effect free expressions are not kept in a central location in Kid graphs, no pointers to the expressions remain in the graph when they are no longer referenced in a program. Id's garbage-collection mechanism automatically eliminates this dead-code from the graph corresponding to the expression. If side-effect-free expressions were kept on blocks in a graph, the nodes corresponding to these expressions would have to be eliminated explicitly as dead-code. Furthermore, because side-effecting expressions are always kept on blocks, the nodes corresponding to these expressions are never eliminated as dead-code. This is desirable because side-effecting expressions should never be removed from a program.

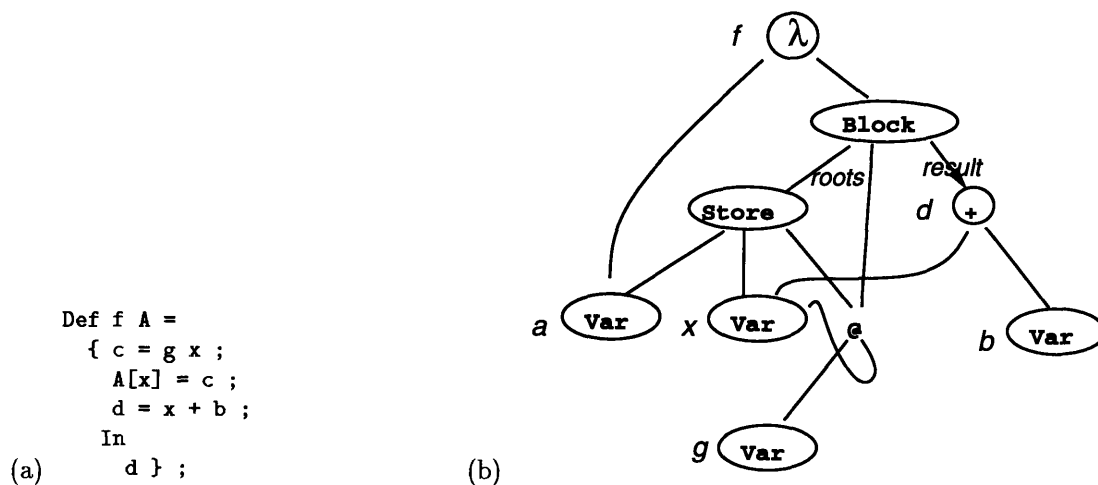


Figure 3-2: A Kid program containing a block (a) and the Kid graph corresponding to the program (b).

3.2 The Problem with Lambdas and Free-Variables

The biggest difficulty that arose in the development of Kid Graphs was finding a way to represent programs as graphs that could be copied efficiently during reduction. Wadsworth noted the centrality of the graph-copying problem in his original paper on graph reduction of the lambda calculus, and the problem has continued to trouble developers of compilers for lambda-calculus-based languages since that time. The problem is simply this. Whenever a lambda expression is reduced, the body of the lambda has to be copied. Such copying is simple when lambdas use only those variables which they define, but it is difficult when lambdas also contain free expressions, or expressions that are bound outside their scopes, because only the bound expressions are copied in graph reduction. For accurate copying it is necessary therefore to represent graphs in such a way that free expressions are distinguished from bound expressions.

There are two lambdas in the Kid Graph depicted in Figure 3-3. In this figure, those nodes that are bound by each lambda, called the *extent* of the lambda, are enclosed in a shaded region. According to Wadsworth, in order to perform the reduction of the expression $((\lambda z. (+ y (* z x))) x)$ those nodes and only those nodes that fall within the extent of the lambda $(\lambda z. (+ y (* z x)))$ should be copied. If you started copying the lambda from the node marked #, however, it would be difficult to tell where to stop copying. The difficulty with graphs is that the nodes that fall within the extent of a lambda can not be easily distinguished from the nodes that are *reachable* from a lambda.

Hughes developed an influential solution to the problem of distinguishing between free and bound variables in lambda abstractions. His solution involves the transformation of a graph into supercombinator expressions before graph-reduction occurs. When a lambda-calculus program is transformed to a supercombinator-program, each lambda in the program is replaced by a special constant, called a supercombinator. A single reduction-rule is then added to the reduction system, corresponding to that constant. When this rule is applied, it will have the same effect as the application of the original lambda to its arguments. For the purposes of this discussion, it is sufficient to think of a supercombinator as a special kind of lambda. What makes a super-combinator lambda special is that it contains only references to its own bound-variables or to other super-combinators. By eliminating all free expressions from lambdas, supercombinator-transformation also eliminates the problem of distinguishing between nodes reachable from lambdas and nodes that fall within the extent of the lambdas. Hughes's approach is called lambda-lifting. It results in "flat" supercombinator programs in which all nested lambdas are found in the outermost level. Figure 3-4 illustrates a program graph before and after it has been lambda-lifted.

Supercombinator-transformed graphs are very useful at execution-time, but they are not an ideal graphic representation of programs for compile-time optimization. A major difficulty is that important optimizations cannot be applied directly to supercombinator-transformed graphs. Take, for example, the constant-propagation optimization. In a program in which the variable x is bound to the value 3, as in Figure 3-5 (a), the constant propagation rule replaces all uses of x in the program with the value 3. With programs represented in the pure lambda-calculus, the optimizer can perform this operation by scanning the programs for all occurrences of x . With programs represented as supercombinators, the optimizer cannot use the constant propagation rule because x can only occur within one scope, the scope in which it is defined. Uses of x in other lambdas have been transformed so that x is an argument to these lambdas. After transformation of a program to supercombinators, it is extremely difficult to reconstruct which arguments of each lambda correspond to x in the original, un-lambda-lifted program. Figure 3-5 (b) shows the lambda-lifted counterpart to the program depicted in (a). It takes extra work to figure out that 3 can substitute for x_1 in the second program.

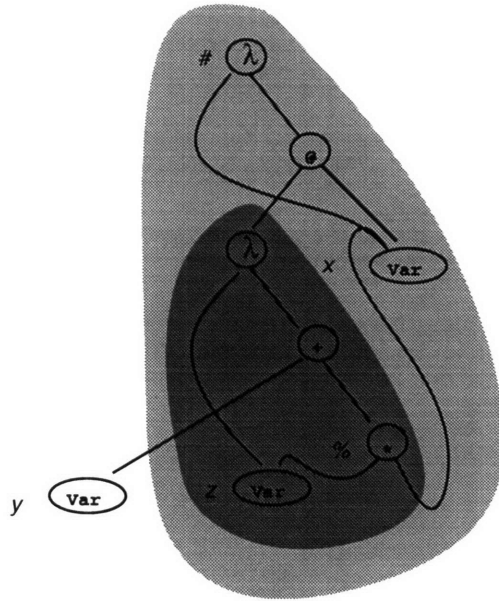
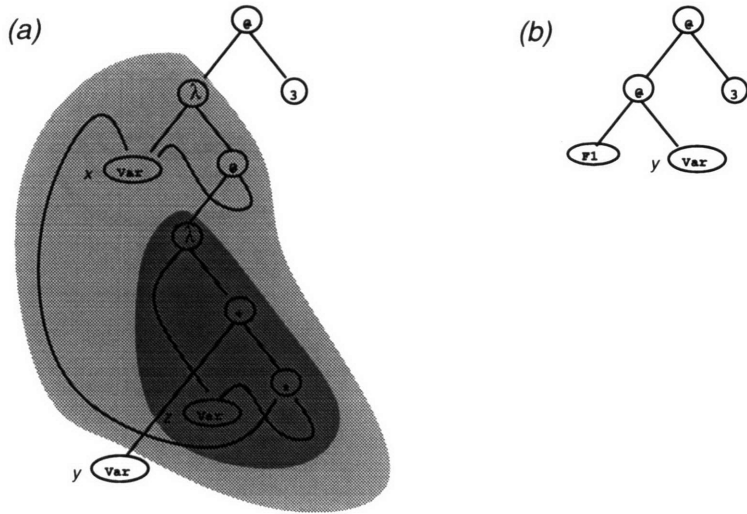


Figure 3-3: Graph representing the expression $(\lambda x. ((\lambda z. (+ y (* z x))) x))$ (a). The extent of each lambda is indicated by differently shaded regions. (b) shows the same graph, except with shared variables represented using shared nodes.

3.3 DeBruijn Chains

Is it possible to determine the extent of lambdas efficiently without first flattening the programs through lambda-lifting? The shaded regions in Figure 3-4 circumscribe the extent of each lambda in a program graph. This program could also be represented as in Figure 3-6. In this figure, each node in the graph has been given a name, which appears in italics next to the node. Each lambda in the graph is annotated with a list, and each list contains the names of the nodes that fall within the extent of that lambda. In a real implementation, these names would correspond to the addresses of the nodes. The original graph has only been annotated, not transformed, to achieve this representation. Constant-propagation on this program is as easy as it would be using the original program. Most importantly, when programs are represented this way, one can quickly determine the extent of any lambda by looking at its list.

Unfortunately, reductions that occur within the extent of a lambda can change the extent of the lambda. In Figure 3-7, a single reduction in a lambda adds new nodes and removes old nodes from the extent of the lambda. If the list is not updated, then the extent of the lambda would be listed incorrectly. One could improve this technique by reducing the number of nodes that are kept in the list of every lambda. For instance, each lambda list might contain only the nodes that fall on the boundaries of the lambda's extent. The lists would still have to be updated, however, whenever boundary nodes were affected by a reduction. These solutions would be particularly cumbersome to



(c)
$$\begin{aligned} ((F1 X) Y) &\rightarrow (((F2 X) X) Y) \\ (((F2 X) Y) Z) &\rightarrow (((+ Y (* X Z)))) \end{aligned}$$

Figure 3-4: Graph for the expression $((\lambda x. ((\lambda z. (+ y (* x z))) x)) 3)$ (a) and its supercombinator representation (b). Both the lambdas in the original graph (a) have been transformed to supercombinator constants and two new reduction rules added to the reduction system (c).

<pre>(a) def f y = { x = 3 ; def g z = (+ x z) ; In (g y) } ;</pre>	<pre>(b) def f y = { x = 3 ; def g x1 z = (+ x1 z) ; In (g x y) } ;</pre>
---	---

Figure 3-5: A program (a) and its lambda-lifted counterpart (b).

implement in parallel because they involve repeatedly checking and updating lists.

Another solution to this problem is to place special nodes on all arcs that leave the extent of a lambda. I will call these special nodes *free-nodes* because they reside on arcs that lead from the expressions defined in a lambda to expressions that are free in that lambda. With this solution, it would not be necessary to maintain centralized lists of free-nodes. The extent of a lambda could be determined by recursively searching for all nodes that are reachable from the lambda, stopping whenever a free-node is reached. A graph that uses this approach is given in Figure 3-8. The extent of each lambda is enclosed in a shaded region and free-nodes are marked with the tag **Free**. Note that all arcs crossing these shaded regions are marked with free-nodes. Note also that the free-nodes are *chained* together. Thus, if lambda *B* is nested inside lambda *A* and if *B* uses a free-variable that is also free to *A*, then *B*'s free-node points to *A*'s free-node. Variable *y* in Figure 3-8 has been chained twice.

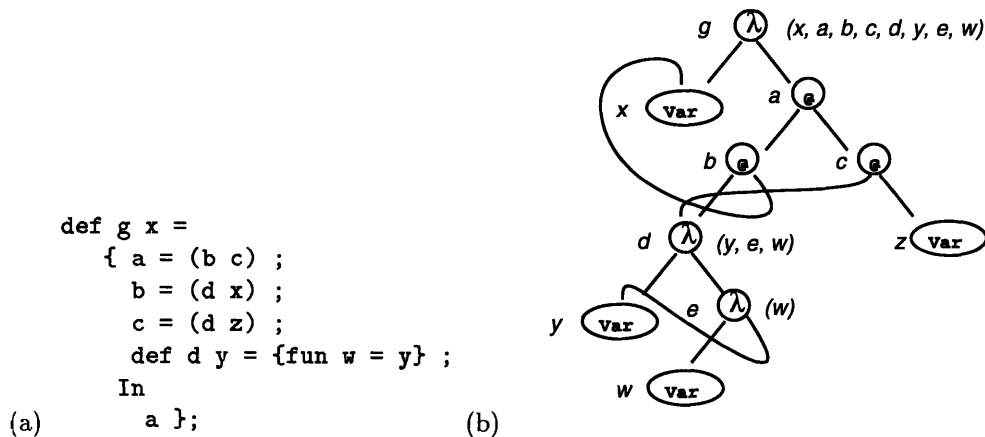


Figure 3-6: A simple program (a) and its graphical representation using lists to denote the extent of lambdas (b).

For an optimizer developer, this program-representation is more useful than a supercombinator-representation would be. Note that by following a free-node chain, it is possible to find the bound version of any free-variable. This information is useful in performing optimizations. For example, it can be used to perform constant-propagation. To propagate a constant 3 for every occurrence of x in a program, replace all free-nodes in x 's chain with the constant 3.

This technique is called DeBruijn chaining because of its similarity to a technique originally developed by August DeBruijn. The goal of DeBruijn's approach was to improve upon the conventional method of naming variables in lambda-calculus programs. In the conventional approach, symbolic names represent variables. DeBruijn's approach uses integers, called *DeBruijn numbers*, to represent variables. A DeBruijn number denotes the number of lambdas that come between the definition of a variable and its use, sometimes called its nesting-depth. Once every variable is assigned a proper DeBruijn number, the original name of the variable can be discarded. For instance, the conventional lambda-calculus program $(\lambda x. (\lambda y. x) x)$ becomes $(\lambda. (\lambda. 1) 0)$ in DeBruijn notation. In these examples DeBruijn numbers are italicized so that they can be distinguished from integer constants. The original DeBruijn notation was designed for representations in which lambdas were restricted to only one argument, but the notation can be extended easily to representations in which lambdas have multiple arguments. Several researchers have developed successful implementations that use this approach [12] [16]. An overview of DeBruijn's representation and its use in program transformation can be found in [15].

Figure 3-10 through 3-12 depict three possible representations of a program as graphs. The first graph in the Figure 3-10 does not use any special system for representing variables within the graph. Each variable is given a single node, and every occurrence of the node in the program is represented by a pointer to this shared node. Again, shaded regions delineate the extent of each lambda.

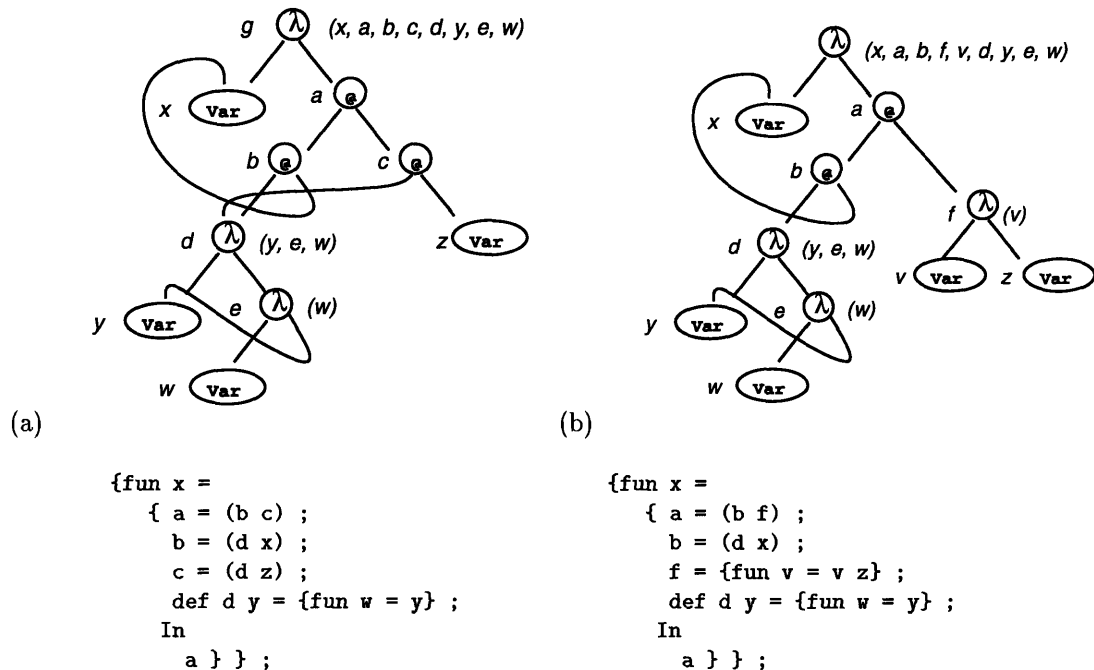


Figure 3-7: Reduction of a program (a) and reduction of a corresponding graph that uses lists to denote the extent of each lambda (b).

The second graph in Figure 3-11 depicts the same program, this time using DeBruijn numbers rather than names. Note that the sharing across nested lambdas that was present in Figure 3-10 has been broken by the numbered renaming in Figure 3-11. Only variables with the same name and the same scope are shared. Also, the number of each variable corresponds to the number of extents the edge to the variable once crossed. Under a DeBruijn number representation, variables in different scopes can easily be distinguished from each other. All the nodes that fall within the extent of a given lambda are marked with the number 0.

The third representation is a hybrid of the first and second. Like the second, only variables within the same lambda can be shared, so that variables in different lambdas can be distinguished from each other. However, chains of pointers have been added to link the use of a variable within each scope. For instance, the use of the variable Y in the expression $(\lambda F.(\dots) Y)$ is linked to its use in expression $(\lambda X.(\dots) Y)$ and its definition in $(\lambda Y.\dots)$. The length of these chains is the same as the DeBruijn number used in the second graph. This third graph illustrates the use of DeBruijn chains.

There is one clear disadvantage in using either DeBruijn names or DeBruijn chains in graph reduc-

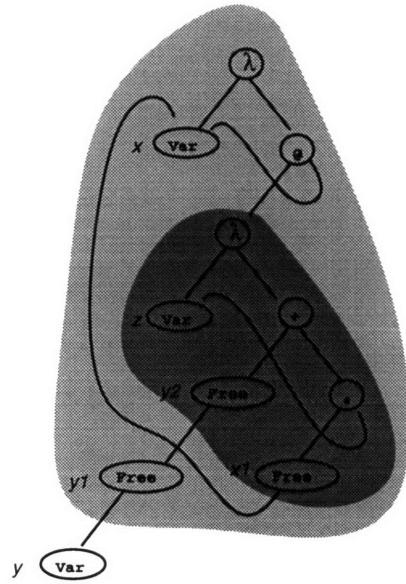


Figure 3-8: Graph representing the expression $(\lambda x.n ((\lambda z. (+ y (* z x))) x))$ using DeBruijn chains. The extent of each lambda is delimited by a shaded region. No arcs in the graph cross the boundary of a shaded region without passing through a free-node.

tion. DeBruijn representations require more overhead during reduction than do super-combinator, lambda-lifted representations. This is because DeBruijn representations use relative offsets to expressions to represent variables names. If an expression moves, then the relative offset of a variable may change. Take, for example, the following reduction of a textual program:

(a) `def g x =` (b) `def g x =(+ 1 x) ;`
 `{ def f a = (+ a x) ;`
 `In (f 1) } ;`

When the application `(f 1)` is reduced, the variable that was previously free in the body of `f` in (a) is no longer free in the copy of `f` in (b). It has moved into the scope of the enclosing lambda. If these programs had been represented using DeBruijn numbers, the use of the variable `x` would be represented using the DeBruijn number `1` in (a) and `0` in (b). Similarly, if this program had been represented using DeBruijn chains, the use of `x` in the first program would have a chain of length one in the first program and would not have any chain at all in the second program.

DeBruijn's original scheme included a set of rules for maintaining DeBruijn numbers during reductions [8]. Figure 3-13 gives the algorithm for performing a beta reduction on a DeBruijn expression¹. The expression $((\lambda.E_1) E_2)$ can be reduced by calling the function `BETA 0 E2 E1`.

¹Neither this algorithm nor the one for reducing DeBruijn chains given in Figure 3-15 includes steps for maintaining sharing within the reduced graphs. It is assumed that some mechanism for maintaining shared dependencies exists. The subject of maintaining sharing correctly will be addressed in Chapter 5.

The main function for performing the beta-reduction is the recursive function BETA which, besides E_1 and E_2 , takes an integer argument i , indicating the current nesting-depth of the recursion. This extra argument is required because beta-substitution of an expression $((\lambda \mathbf{x}. E_1) E_2)$ requires replacing all occurrences of \mathbf{x} in E_1 with occurrences of E_2 . If E_1 contains nested lambdas, then different occurrences of \mathbf{x} in these lambdas appear as different DeBruijn numbers, each reflecting a different occurrence of \mathbf{x} . BETA uses i to keep track of the nesting-depths so it can correctly determine which DeBruijn number corresponds to \mathbf{x} .

The expression E_2 may also contain expressions that are bound outside of E_2 and therefore also outside of E_1 . These indices will change when E_2 is substituted for \mathbf{x} . The amount of change in an index depends on the nesting-depth of the \mathbf{x} . It is therefore necessary to make *different copies* of E_2 for each substitution of E_2 for \mathbf{x} . This is accomplished by the SHIFT function in the DeBruijn algorithm given in Figure 3-13. Besides E , the expression to be shifted, SHIFT takes in two other integer arguments i and d . The integer i indicates the nesting-depth of \mathbf{x} , the variable that will be replaced by E . Because SHIFT is itself recursive, it also needs to keep track of the nesting-depth of its own recursions, using d . SHIFT makes a copy of E , shifting the indices of E depending on i and d . Figure 3-14 depicts the reduction of a program using DeBruijn numbers. Note that the expression corresponding to (a \mathbf{z}) occurs twice in the final program.

The algorithm for reducing DeBruijn numbers can be used to reduce DeBruijn chains, except that the rules for altering numbers must be altered to apply to chain lengths. Figure 3-15 gives the algorithm for performing a beta-reduction of the expression $((\lambda x. E_1) E_2)$ using DeBruijn chains. Again, the main beta-reduction function BETA takes an argument i that keeps track of the current nesting-depth of the recursion, and the initial reduction would start with a call to BETA $0 E_2 x E_1$. Note that changing a variable in the outermost depth of nesting has the effect of changing an entire chain of variables in a DeBruijn chain representation. By removing one length of a DeBruijn chain in the outermost scope, the optimizer reduces by one the distance of all nodes in nested scopes to the end of the chain. By substituting one expression in a DeBruijn chain in the outermost scope, the substitution can be seen by all nodes in nested scopes to the end of the chain. The function BETA therefore only makes changes in the graph when i , the nesting depth, equals 0 . Figure 3-16 depicts the reduction of a program using DeBruijn chains. What is interesting about this example is that, in order to replace the variable \mathbf{y} with the variable \mathbf{z} in the program, the BETA function has to enter a nesting-depth of depth 1 and then exit to nesting depth 0 after encountering the free-node for \mathbf{y} .

In terms of the Id-in-Id compiler, the crucial difference between DeBruijn numbers and DeBruijn chains is that DeBruijn chains work well in graph-reduction and DeBruijn numbers do not. The problem with DeBruijn numbers should be apparent in Step 1b in the DeBruijn number algorithm given in Figure 3-13. The algorithm requires making separate copies of E_2 for each occurrence of

x . The primary motivation for using graph-reduction is to avoid making separate copies of E_2 for every occurrence of x . DeBruijn chains use pointers to represent relationships between program elements. It is not necessary to make separate copies of E_2 when performing a substitution on a program represented with DeBruijn chains. In addition, it must be reiterated that it is not possible for each occurrence of x to point directly to E_2 with DeBruijn numbers. This is because the DeBruijn numbers within E_2 represent the nesting-depth of variables in E_2 's environment, not x . Figure 3-17 depicts an example of what would happen if occurrences of x were allowed to point directly to E_2 in the DeBruijn number representation.

To summarize, Wadsworth identified a difficult obstacle to the use of graph reduction in compilers in his original article describing the invention of the technique. He pointed out that one must be able to find the extent of lambda bodies in order to perform graph reduction correctly, and finding the extent of lambda bodies is not always easy. One technique that has been used to overcome this obstacle is the super-combinator approach to graph representation. Although this technique has worked well with interpreters, it is not useful with optimizers. The Id-in-Id compiler solves the problem of finding lambda bodies by employing a unique approach to naming: DeBruijn chains. These chains are similar to DeBruijn numbers in some respects, but unlike DeBruijn numbers, they are suitable for use in graph reduction. A revised grammar for Kid Graphs, using DeBruijn chains, is given in Figure 3-18.

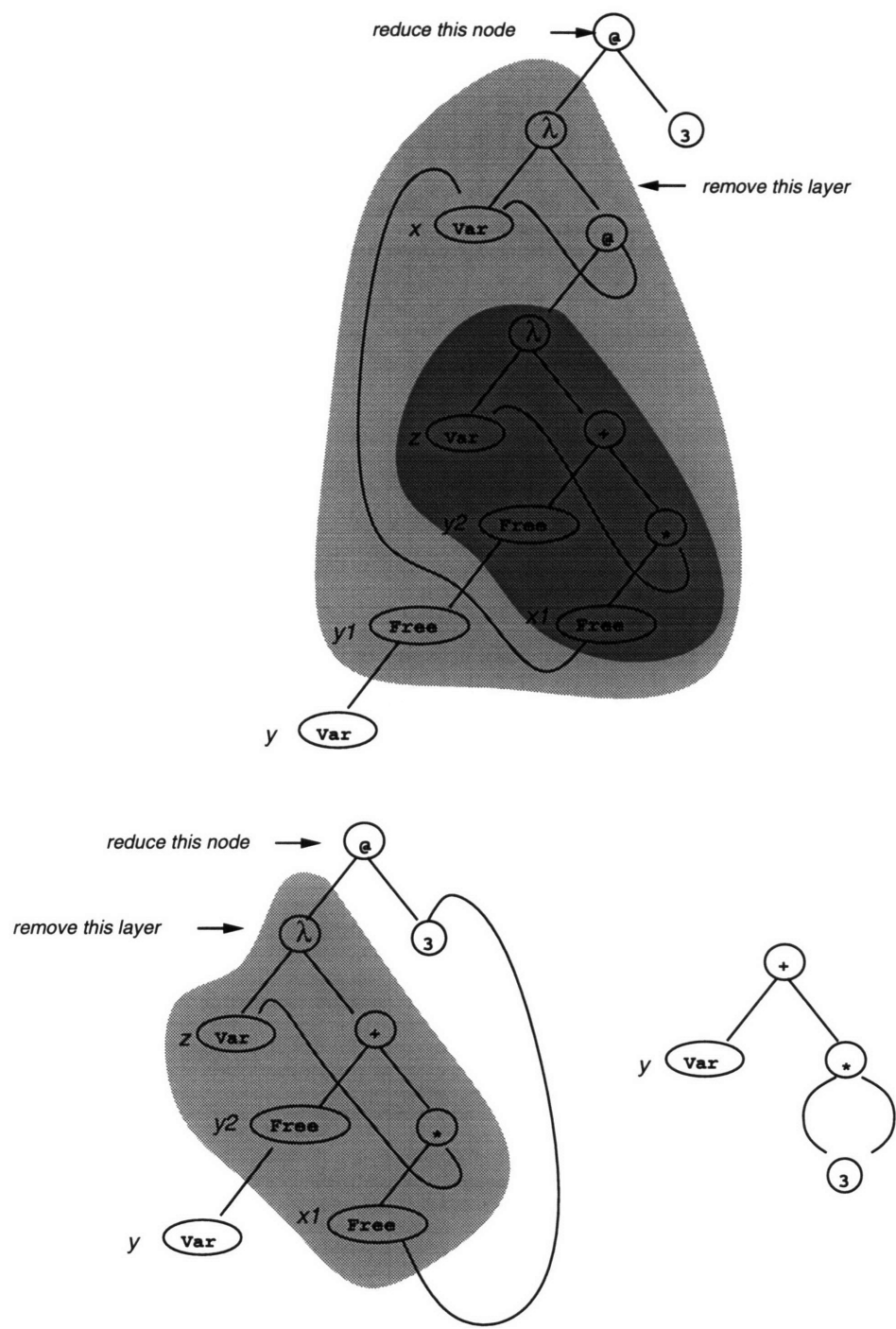


Figure 3-9: Reduction of the expression $((\lambda x. ((\lambda z. (+ y (* z x))) x)) 3)$ using DeBruijn chains. Each time an application is reduced, a layer of indirection is removed from each lambda body.

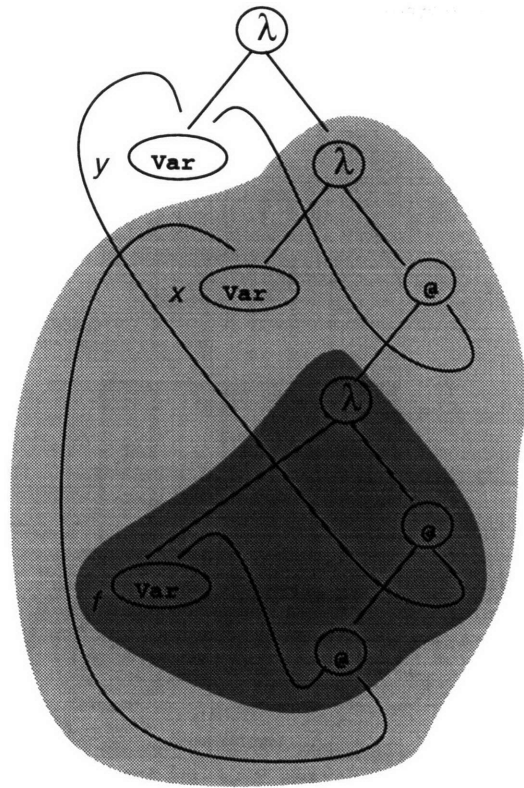


Figure 3-10: The program $(\lambda y. \lambda x. ((\lambda F. (F X) Y) Y))$ represented as a graph.

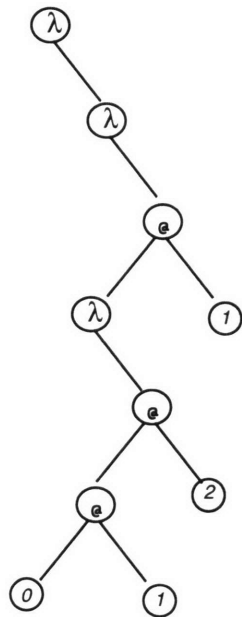


Figure 3-11: The representation of $(\lambda y. \lambda x. ((\lambda F. (F X) Y) Y))$ using DeBruijn numbers.

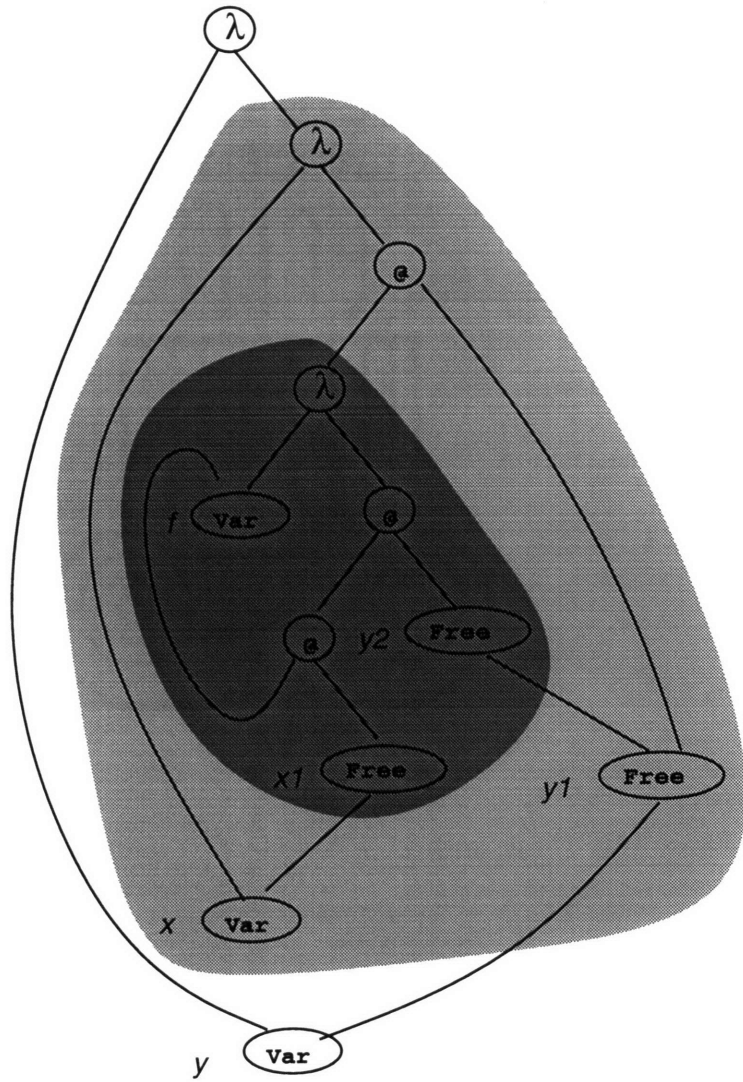


Figure 3-12: The representation of $(\lambda y. \lambda x. ((\lambda F. (F X) Y) Y))$ using DeBruijn chains.

BETA $i E_2 E_1$:

1. If E_1 is the DeBruijn number j :
 - (a) If $j < i$ then j is a variable bound within E_1 , but not by E_1 . Return DeBruijn number j .
 - (b) If $j = i$, then j is the variable bound by E_1 and should be substituted with E_2 . Make a copy of E_2 , E'_2 by calling SHIFT with arguments i 0 and E_2 . Return E'_2 .
 - (c) Otherwise, j is a DeBruijn number representing a free-variable of E_1 . Return DeBruijn number $j - 1$.
2. If E_1 is not a DeBruijn number, then copy the children, $c_1 \dots c_n$, of E_1 to $c'_1 \dots c'_n$:
 - (a) If E_1 is a lambda with child c_1 , copy c_1 by calling BETA $i + 1 E_2 c_1$.
 - (b) Otherwise, copy each child c_i by recursively calling BETA $i E_2 c_i$.
3. Use $c'_1 \dots c'_n$ to create a copy of E_1 , E'_1 . Return E'_1 .

SHIFT $i d E$:

1. If i is zero, then the indices of E do not need to be shifted. Return E .
2. If i is not zero and E is DeBruijn number j , where $j \geq d$, then j must be shifted. Return DeBruijn number $j + d$. Otherwise, return DeBruijn number j .
3. If i is not zero and is not a DeBruijn number, then E and its sub-expressions must be shifted. First shift the children, or sub-expressions, $c_1 \dots c_n$ of E to $c'_1 \dots c'_n$:
 - (a) If E is a lambda with body c_1 , then shift c_1 by calling SHIFT $i d + 1 c_1$
 - (b) Otherwise, copy each child c_i by calling SHIFT $i d c_i$.
4. Use $c'_1 \dots c'_n$ to create a copy of E , E' . Return E' .

Figure 3-13: Beta-substitution algorithm for programs represented using DeBruijn numbers.

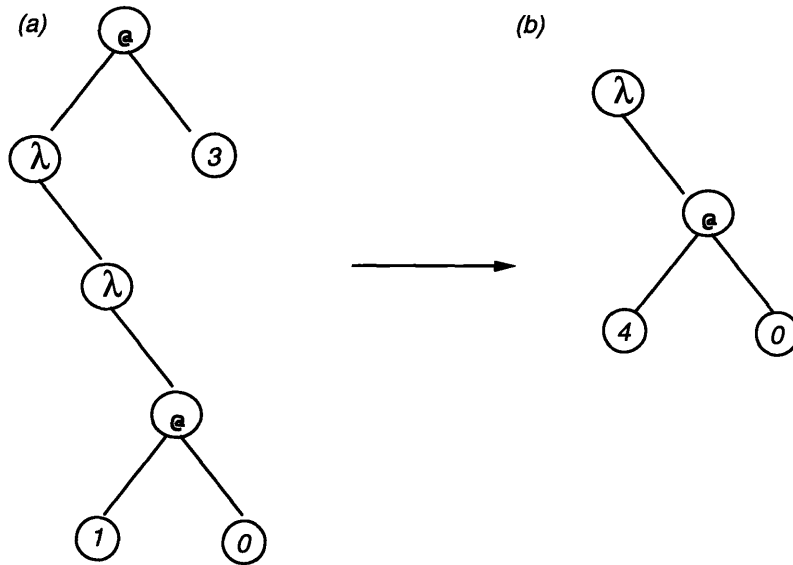


Figure 3-14: Reduction using DeBruijn numbers of expression $((\lambda y. (\lambda x. (y x))) z)$, where the use of variable z is three lambdas away from its definition.

BETA $i E_2 x E_1$:

1. If E_1 is the variable x and i is 0, then return E_2 .
2. If E_1 is a free-node pointing to node c and i is 0, then return c .
3. If E_1 does not satisfy either of the above two conditions, then copy of the children, $c_1 \dots c_n$, of E_1 to $c'_1 \dots c'_n$:
 - (a) If E_1 is a free-node with child c_1 , copy c_1 by calling BETA $i - 1 E_2 x c_1$.
 - (b) If E_1 is a lambda with child c_1 , copy c_1 by recursively calling BETA $i + 1 E_2 x c_1$.
 - (c) Otherwise, copy each child c_i by recursively calling BETA $i E_2 x c_i$.
4. Use $c'_1 \dots c'_n$ to create a copy of E_1 , E'_1 . Return E'_1 .

Figure 3-15: Beta-Substitution algorithm for programs represented using DeBruijn chains.

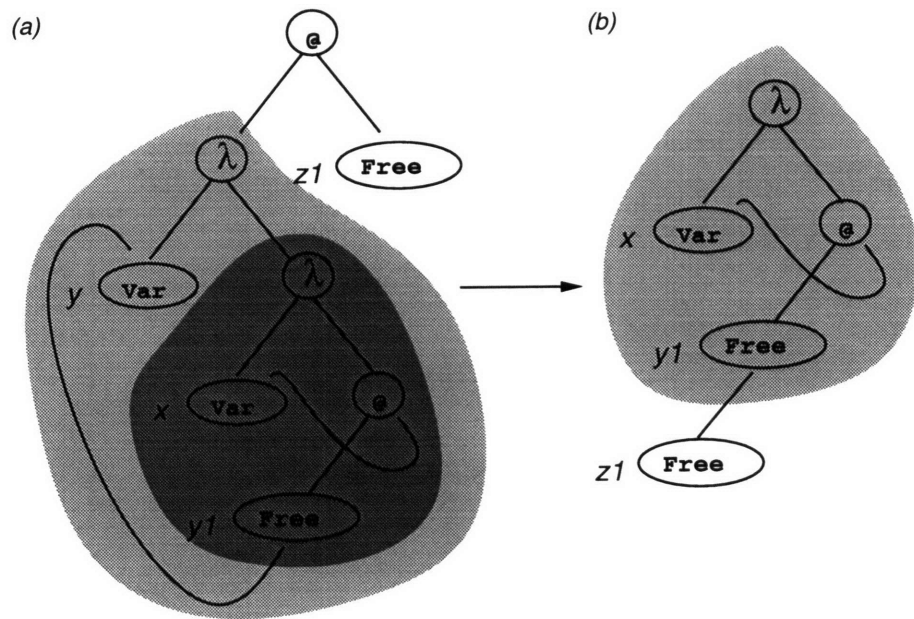
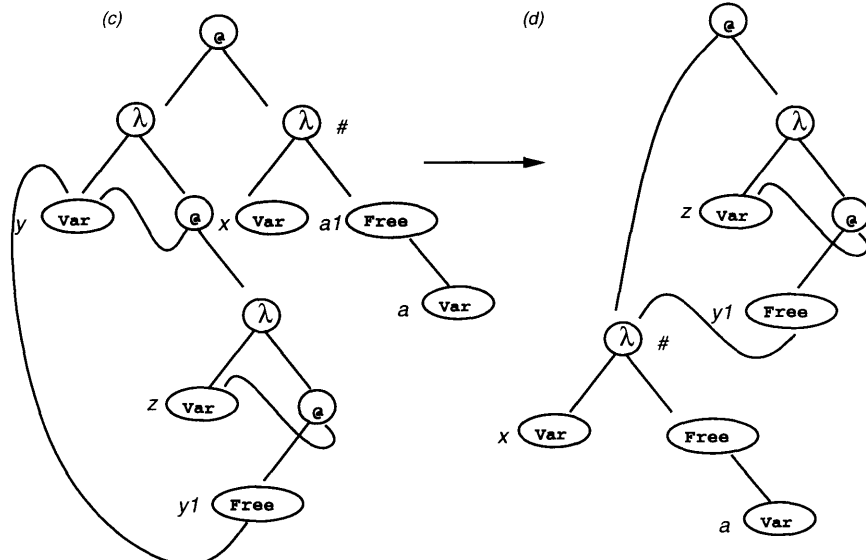
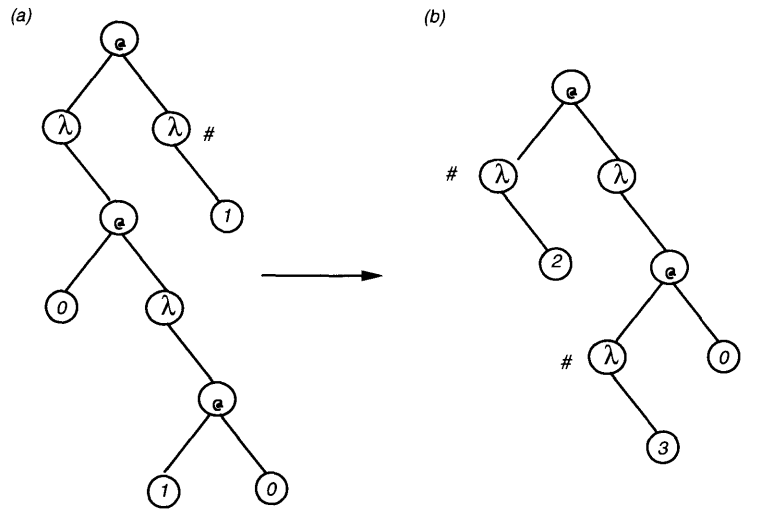


Figure 3-16: Reduction using DeBruijn chains of expression $((\lambda y.(\lambda x.(y x))) z)$, where the use of variable z is three lambdas away from its definition.



```

{ def g x = a ;
  def f y =
    { def h z = (y z) ;
      In (y h) } ;
  In
  (f g) } ;

{ def g x = a ;
  def h1 z = (g z) ;
  In
  (g h1) } ;

```

Figure 3-17: Graph-reduction of a program, one represented using DeBruijn chains (a) and the other represented using DeBruijn numbers (b). Notice that the node marked # appears twice in the reduced DeBruijn number graph (a) and only once in the reduced DeBruijn chain graph.

<i>PF₁</i>	::=	Allocate
<i>PF₂</i>	::=	+ - ... Equal?
<i>Exp</i>	::=	<i>Variable</i> <i>Constant</i> <i>PF_n</i> (<i>Exp</i> , ..., <i>Exp</i>) Free <i>Exp</i> Bool_Case (<i>Exp</i> , <i>Lambda</i> , <i>Lambda</i>)
<i>Lambda</i>	::=	$\lambda(x_1, \dots, x_n).Block$
<i>Block</i>	::=	{ [<i>Root</i> ;]* In <i>Exp</i> }
<i>Root</i>	::=	<i>Exp</i> <i>Command</i>
<i>Command</i>	::=	P_store (<i>Exp</i> , <i>Exp</i> , <i>Exp</i>) Store_error \top_s

Figure 3-18: Grammar for Kid Graph nodes.

Chapter 4

Data-structures for Performing Reductions

The Id-in-Id optimizer optimizes programs by applying reduction rules to them. Practically, this means that the optimizer changes, or transforms, program graphs. In a parallel optimizer, these transformations are made in parallel, so the graphs that represent the programs must be amenable to parallel transformation.

At its core, Id is a functional language and supports a wide variety of functional data-structures such as lists, arrays, records, and union-types. In theory, any of these data-structures can be used to represent program graphs. One advantage of using functional data-structures is their expressivity. It is easy to reason about functional data-structures in parallel programs because, once created, they can not change.

The disadvantage of using functional data-structures in programs is in memory management. Because functional data-structures cannot be mutated, it is necessary to copy them in order to achieve the effect of mutation. For instance, to change one element of functional array A , it is necessary to create a new array A' that contains the new element and copies of all the other elements of A . The implication is that in order to perform a single reduction, a reducer might have to copy an entire program graph. This approach requires both time and memory space.

Along with compiler optimization and run-time garbage collection, careful programming can often reduce these inefficiencies. Figure 4-1 shows a picture of a small graph. If one were to change node A in graph G certain nodes in G could be taken directly from G to form the new graph G' . These nodes include all of the descendents of A and all of the siblings of A . However, all the nodes that are direct ancestors of A must be copied to form the new graph G' . Using this technique, a reducer would have to check at every step in reduction to determine whether the descendants of a node have changed and whether or not the node must be copied. Though this technique reduces the

amount of memory that is required to perform reductions, it increases the amount of time required to perform them.

In light of this, it seems worthwhile to explore the possibility of using mutable graphs for program transformation. The appeal of using mutable data-structures is that a change in a graph can be performed directly on the graph. As soon as such a mutation is made, the change is visible to the rest of the graph. Mutation does not require copying. Mutation also seems worthwhile because, as Barth, Nikhil, and Arvind have noted [20], the use of mutable data-structures in functional programming can sometimes lead to programs that exhibit even more parallelism than programs that do not.

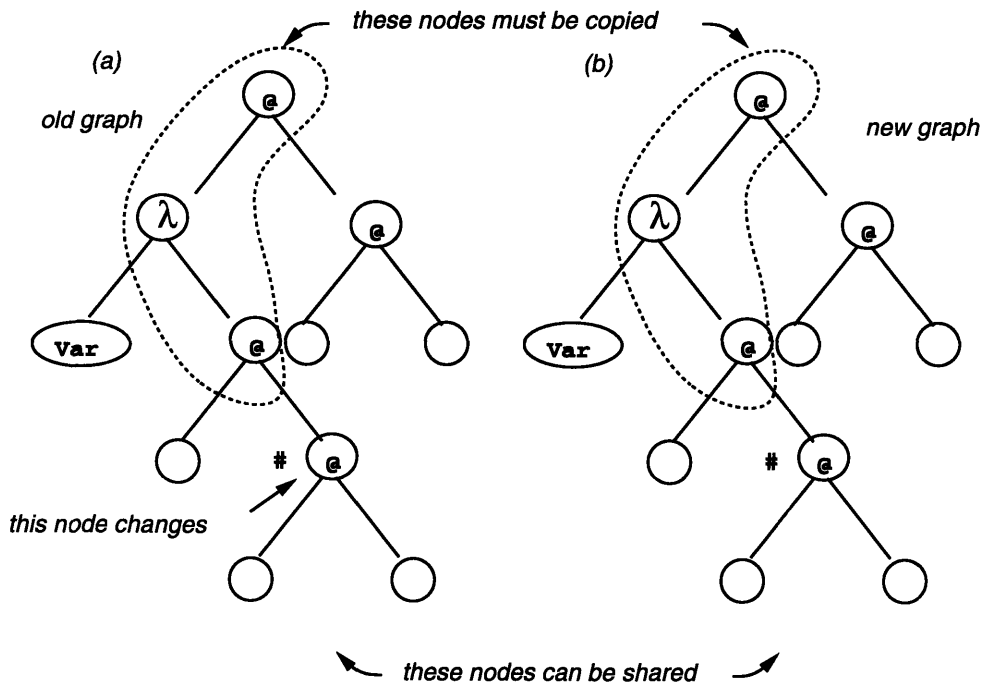


Figure 4-1: In order to change a node in a functional graph, only ancestors of the node have to be copied to create the new graph. The graph in (a), above, needs to be changed at the node marked #. The changed version of the graph in (b), shares nodes with (a) that are descendants and cousins of the node marked #.

4.1 Indirection Nodes

Many sequential implementations of graph-reduction add special nodes to graphs, called indirection-nodes, to aid in the transformation of mutable graphs. The idea of using indirection-nodes was introduced by Wadsworth and a complete description of indirection-nodes is given in [18]. Peyton-Jones illustrates the importance of indirection-nodes by posing the following question: What does it mean, in terms of the mutation of pointers and memory locations, to perform a single reduction

from x to y in a graph? Peyton-Jones cites two possibilities, illustrated in Figure 4-2 and Figure 4-3:

1. We could construct a copy of node y at location x . All previous references to a would now see a node that is identical to y , so the desired effect would be achieved. This kind of transformation is shown in Figure 4-2.
2. We could construct a special indirection-node at the memory location marked by x . The indirection node would contain a pointer to the memory location for y . All previous references to x would now see the indirection node. By dereferencing the indirection-nodes, the expression y could be obtained. This kind of transformation is shown in Figure 4-3. The indirection node is represented with the symbol ∇ .

The problem with the first solution is that, in the example given in Figure 4-2, a single application of f to 6 becomes two applications. Duplicating applications decreases program efficiency. In addition, if the expression $(f\ 6)$ contains side-effects duplicating applications may lead to incorrect program execution. The indirection-node solution does not duplicate expressions. Indirection-nodes do take time to dereference, however. Peyton-Jones suggests that this weakness can be overcome by *short-circuiting* indirection-nodes after they have been dereferenced. Figure 4-4 shows how a graph containing multiple indirection-nodes can be short-circuited. Although this approach works satisfactorily in a sequential environment, one would have to solve additional problems before using indirection nodes in a parallel environment.

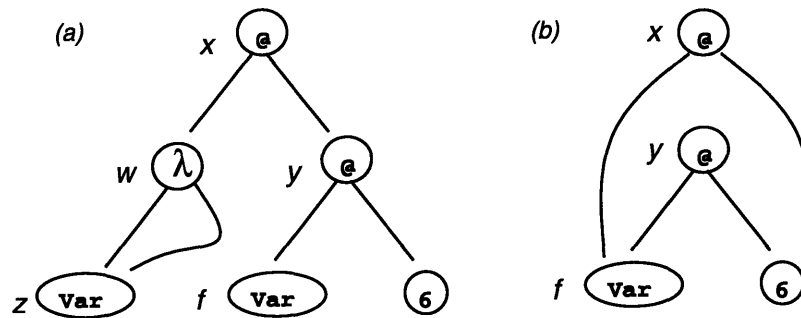


Figure 4-2: Beta-reduction of the program $((\lambda z.z) y)$, where $y = (f\ 6)$, using copying. As a result of copying, a single application of f to 6 in (a) becomes two applications in (b). This is problematic if y is referred to by other expressions.

4.2 Unions of Disjoint-Sets

Another approach is to look at the problem at a higher level. The purpose of reduction is to equate one expression with another. When we reduce the expression $(+ 1\ 2)$ to the expression 3 , we perform the reduction because we know that the expressions are equivalent, or interchangeable. More

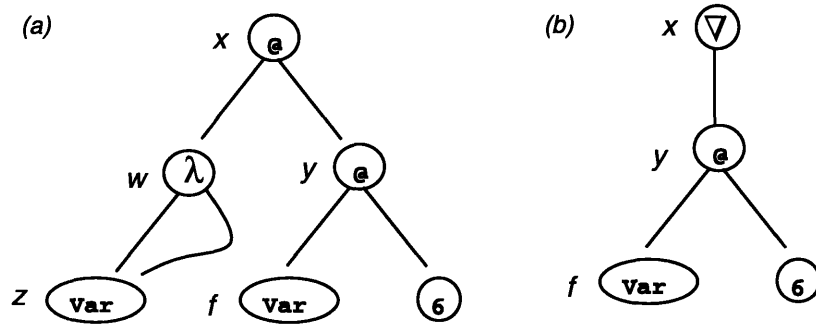


Figure 4-3: Beta-reduction using indirection-nodes. As a result of indirection nodes, all references to a in (b) must be dereferences to obtain the expression (f 6).

generally, when we reduce expression X to expression Y , we are saying that we believe expression X is equivalent to expression Y and furthermore that everything that is currently equivalent to X or will ever be equivalent to X is also equivalent to Y . We are also saying that everything that is currently equivalent to Y or ever will be equivalent to Y is equivalent to X and all its equivalents. The advantage of looking at reduction in this way is that it does not focus on single, individual steps but on the overall effect that reduction tries to achieve.

Another advantage of looking at reduction in this way is that it highlights the similarity between performing reductions on expressions and performing unifications on disjoint-sets. A disjoint-set is a collection of objects. Disjoint-sets can be joined, or unified, together and queried to see whether two objects are members of the same set. Disjoint-sets have proved to be a useful tool in a wide variety of programming applications, and researchers in programming algorithms have long been interested in their efficient implementation. Solutions for finding the connected components of a graph, for instance, make heavy use of disjoint-sets. In these solutions, each connected component is represented by a disjoint-set. When edges connecting two components are discovered, the sets corresponding to the components are unified.

Sophisticated data-structures for representing disjoint-sets and algorithms for performing operations on them have been invented to fulfill this need. It would be nice if these same data-structures and algorithms could be used for reducing graphs. In a typical disjoint-set data-structure, each set is identified by some representative, which is a member of the set. The member of the set that gets to be the representative may be chosen by some prespecified rule, such as the smallest member of the set. Two operations can be used to manipulate disjoint-set data-structures, `REPRESENTATIVE` and `UNION`. Given an element of a set, the operation `REPRESENTATIVE` finds the representative of the set that contains that element. `REPRESENTATIVE` can therefore be used to tell whether two items belong to the same set. Given two elements of two sets, the `UNION` operation unifies the sets to which the elements belong. It follows that the representative of two sets that have been unified is the same. A summary of disjoint-set data-structures and algorithms for implementing them are

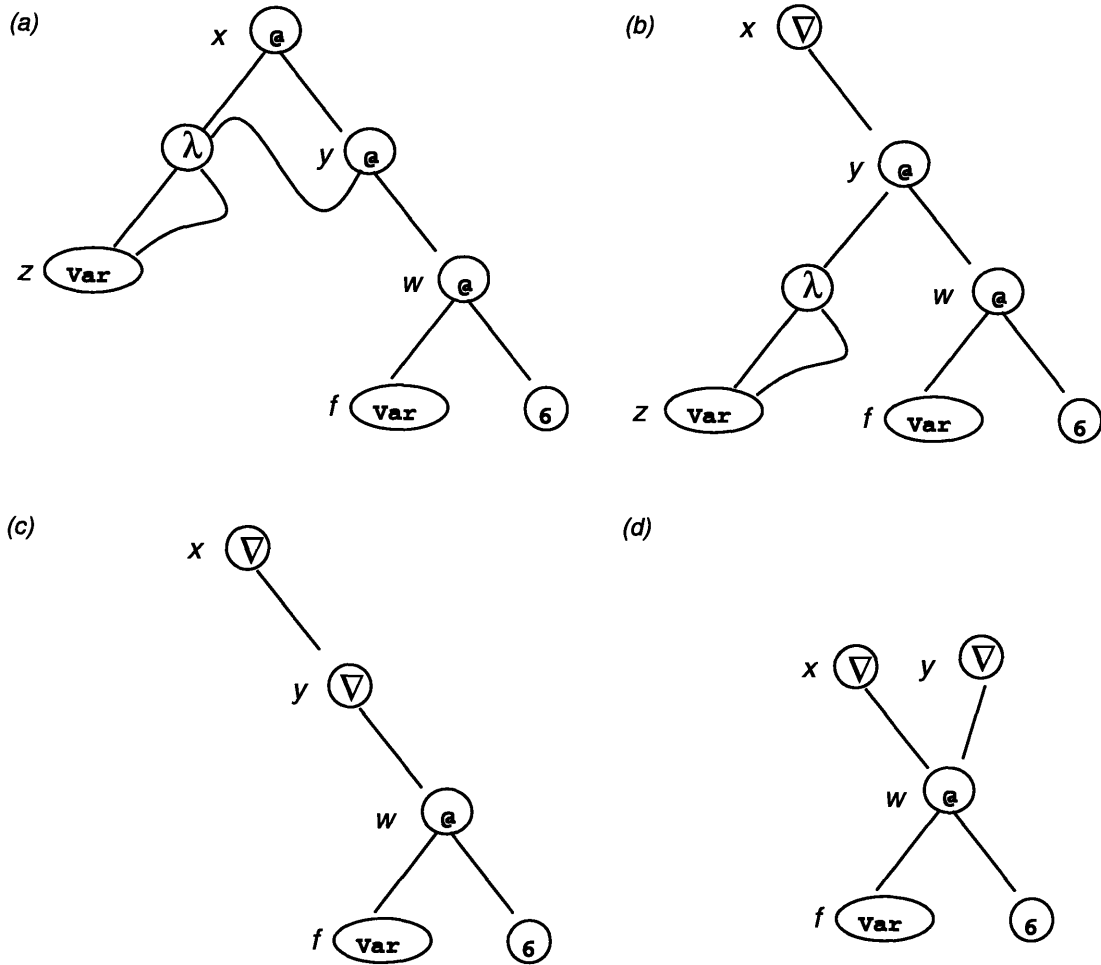


Figure 4-4: Multiple beta-reductions can lead to chains of indirection-nodes. The graph in (d) figure shows the use of short-circuiting to shorten the length of indirection-node chains in (c).

given in [6].

One approach to representing disjoint-sets is to use disjoint-forests. In a disjoint-forest representation each member of a set contains one pointer. The pointer points to another member of the set, called its parent. The member of a set that is also the representative of the set is its own parent. Figure 4-5 (a) shows a picture of two disjoint sets, where c and g are the representatives of the two sets. Each set is called a tree. The algorithm for performing a REPRESENTATIVE and UNION operation on disjoint-sets represented as disjoint-forests is given in 4-7. A UNION operation on two trees changes the representative of one set to the other by altering the parent node of the representative as in Figure 4-5 (b). The representative of a member of a set can be found by chasing parent pointers. A more efficient way of implementing the REPRESENTATIVE operation is to simultaneously chase parent pointers and rewire the parent pointers directly to the representative. This method is called *path-compression* and the revised algorithm for REPRESENTATIVE using path-compression is given in Figure 4-8. An example illustrating path-compression is given in Figure 4-6.

It is possible to use disjoint-forests to implement graph-reduction. In this approach, each node

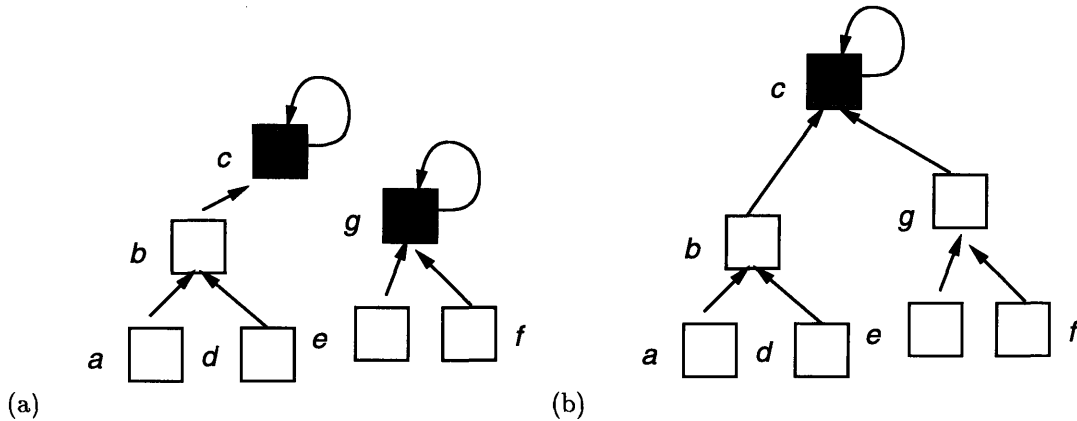


Figure 4-5: Two disjoint-set forests (a) and their union (b).

in the graph is also a member of a set. Before reductions begin, no unifications have been performed so every node is the only element in its set. To reduce node a to node b , perform a UNION on a and b . Any further queries about “What have a and b been reduced to?” can be answered by finding out to which set a and b belong using the REPRESENTATIVE operation. In order for this scheme to work correctly, the UNION operation can not choose the representative of the unified set arbitrarily, but in a left-to-right order. For instance, if we reduce variable a with the integer 1 by performing a UNION operation and then ask for the representative of 1, we expect the answer to be 1, not a .

The top left graph in Figure 4-9 is a program graph where every expression of the graph is also an element of its own disjoint-set. There are two different kinds of pointers depicted in this graph. The pointers without arrow-heads represent dependencies within the program graph. The pointers with arrow-heads represent dependencies within each set. The following two graphs illustrate beta-reductions of this graph using the disjoint-set UNION operation. The REPRESENTATIVE operation can be used to find out the tag of node a after the reduction/unification. The final figure in Figure 4-9 shows the effect of path-compression on the reduced graph.

4.3 Comparing Indirection-nodes and Disjoint-Set Unions

The similarity between Figure 4-4 and Figure 4-9 is undeniable. If we replaced indirection nodes with disjoint-set forest members in Figure 4-4, or replaced disjoint-set forest members with indirection nodes in Figure 4-9, the figures would be identical. It is, in fact, possible to perform disjoint-set union reductions using an indirection-node representation. By convention, a member of a set would be any node that has the tag ∇ . If the node had a tag other than ∇ , then the node would be the representative of its set. The representative of any member of a set could be found by chasing the

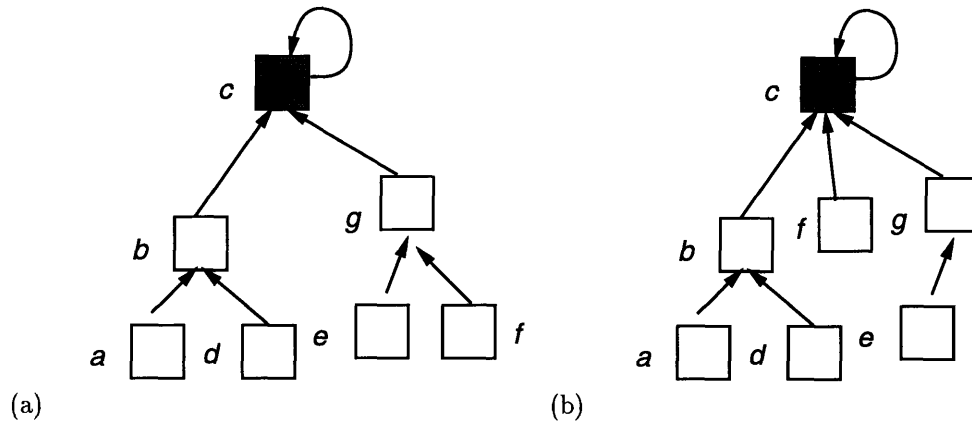


Figure 4-6: The path from member f to its representative c in a set (a) can be compressed (b).

UNION $x y$:

1. Find the representative of x and y by chasing parent pointers.
2. Set the parent of x to be y 's parent. Some versions of UNION use a test to decide whether x 's parent or y 's parent should become the representative of the unified set.

REPRESENTATIVE x : Chase parent pointers to find x 's representative and return this representative.

Figure 4-7: The operations UNION and REPRESENTATIVE for disjoint-set forests.

children of ∇ nodes. Figure 4-10 depicts the UNION and REPRESENTATIVE algorithms modified to use these conventions.

Why would anyone choose to use the disjoint-union approach over the original indirection-node approach? At first glance, the differences seem mostly stylistic. Indirection-nodes blur the line between program graphs and the data-structures that are used to represent program graphs. As a result, it is common a common mistake to assume that indirection nodes have a place in language grammars and reduction rules, even though they are semantically meaningless. The disjoint-set forest approach hides indirection-nodes under a layer of data-abstraction.

It is tempting to say that, apart from the stylistic difference, the approaches are otherwise the same. There is an important algorithmic difference between the two, however, that surfaces when we ask the question, What happens in each approach when we reduce a to b , when the representative of b is not known? An example of where this situation may arise is in the reduction of an expression $(+ E 0)$ to the expression E , using the identity rule for addition. At the time of the reduction the

REPRESENTATIVE x :

1. If x is a representative, then return x .
2. If x is a member with parent-pointer pointing to y , set x 's parent to be REPRESENTATIVE y , rather than y .

Figure 4-8: The modified REPRESENTATIVE operation, using path-compression, for disjoint-set forests.

representative of a , the node with the addition tag, must be known otherwise it would not have been possible to determine that a could be reduced. The representative of b , or E need not be known at the time of the reduction. Figure 4-11 (a) provides a picture of such an expression, represented as a program graph using indirection-nodes. Figure 4-11 (b) shows the result of applying the pure indirection-node reduction algorithm to this graph. Figure 4-11 (c) shows the result of applying the disjoint-forest UNION reduction algorithm to this graph. The difference between the approaches becomes clear. The indirection-node approach alters the representative a so that it points to b . The disjoint-set union approach alters the representative a so that it points to b 's representative.

The disjoint-set union algorithm may seem inferior to the indirection-node algorithm because it requires chasing b 's indirection-node chain unnecessarily. Consider the case where the operation directly succeeding the UNION $a b$ is REPRESENTATIVE a . The length of a 's chain in Figure 4-11 (c) is shorter than the length of a 's chain in (b). Performing REPRESENTATIVE on a in (c) requires fewer steps than performing REPRESENTATIVE on a in (b). The total number of steps required by a UNION operation followed by a REPRESENTATIVE operation is the same for both algorithms.

The algorithm that a specific application should use therefore depends on the mix of UNION and REPRESENTATIVE operations expected for that application. The disjoint-set union reduction algorithm is optimized for the case where an even mixture of UNION and REPRESENTATIVE operations on sets is expected to occur. In this case, the pure indirection-node approach to reductions would not fare as well as the disjoint-set union approach. In the case where many more UNION than REPRESENTATIVE operations are expected to occur, than the indirection-node approach is superior to the disjoint-set union approach. Program optimization involves many more REPRESENTATIVE than UNION operations. In this case, either either algorithm would be sufficient.

This chapter examined several data-structures for program transformations. It briefly examined functional data-structures, which were deemed too inefficient for use in a compiler. It also examined the problem of using mutable data-structures to represent programs and outlined two solutions, indirection-nodes and disjoint-set unions. Though these solutions were developed entirely independently, in the end, their differences are small. For the Id-in-Id compiler project, the most important difference between approaching reductions using indirection-nodes and using disjoint-set forests has

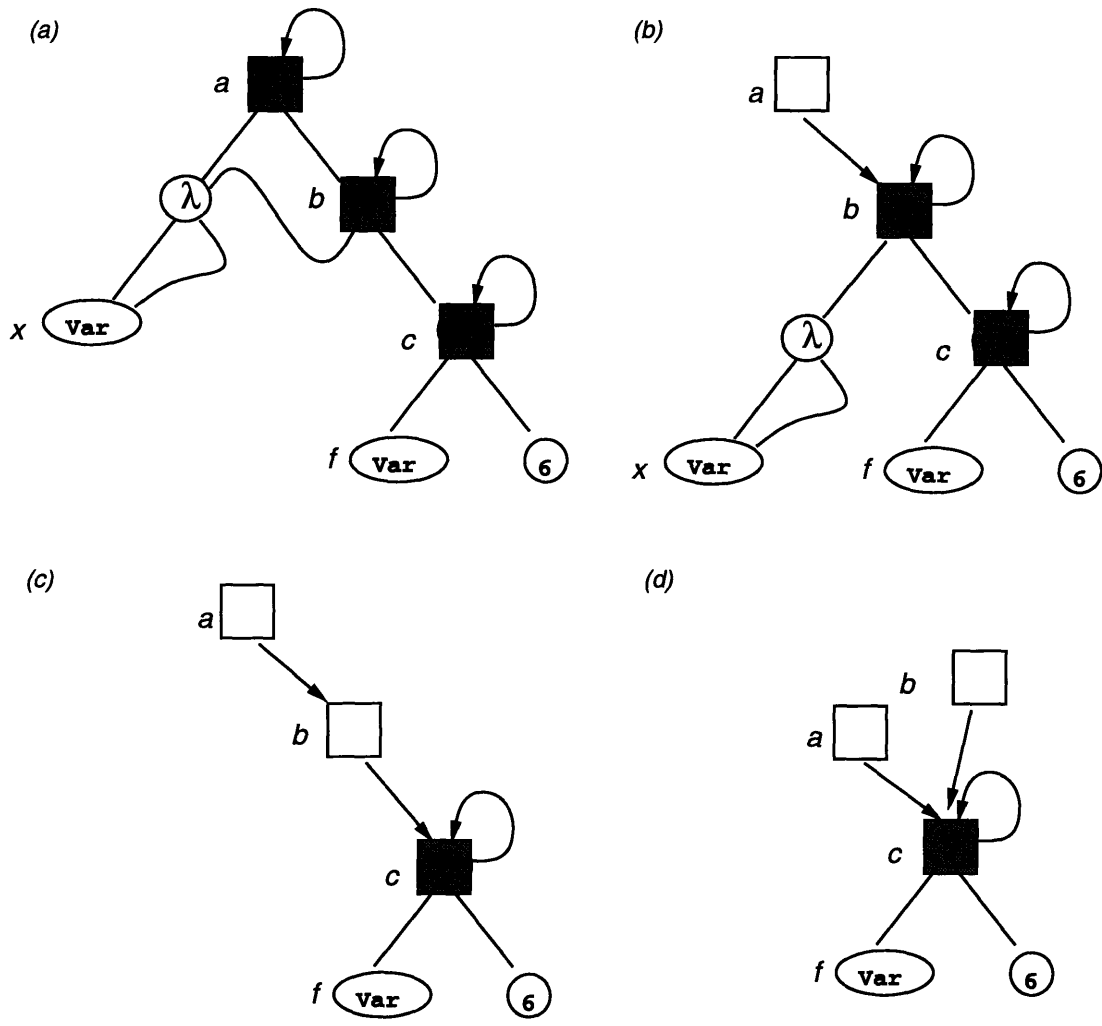


Figure 4-9: Using disjoint-sets to implement beta-reduction.

to do with parallelism. The task of implementing a parallel, disjoint-set forest data-structure is less intimidating than the task of implementing parallel, indirection-node reduction. In fact, before the Id-in-Id compiler project began, Barth had already accomplished a parallel implementation of disjoint-set forests in Id using m-structures in [5]. The Id-in-Id compiler therefore uses Barth's disjoint-set forests to perform reductions on KId graphs.

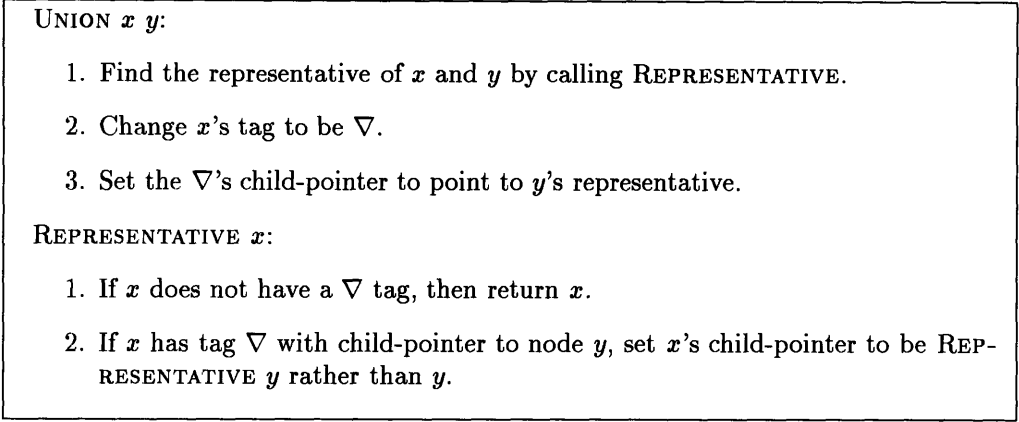


Figure 4-10: Revised UNION and REPRESENTATIVE operations, using ∇ nodes to represent members of sets.

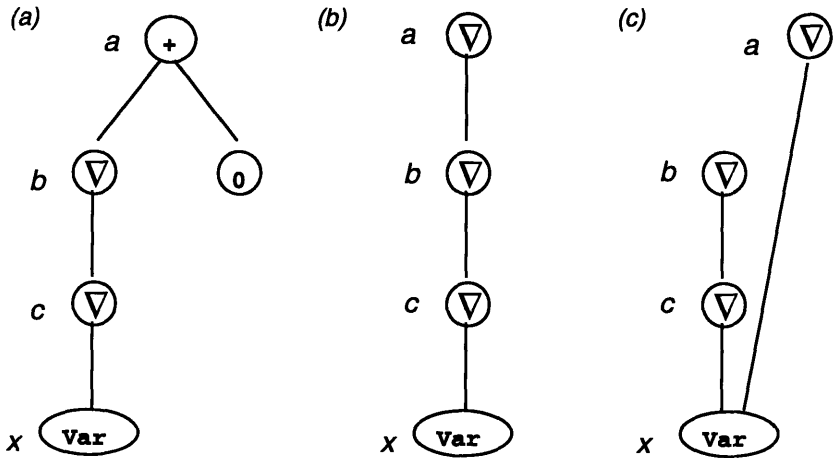


Figure 4-11: A program graph (a) representing the expression $(+ x 0)$ that can be reduced using the algebraic-identity optimization rule. The figures to the right depict the reduction of this program using the pure indirection-node algorithm for reduction (b) and the disjoint-set union approach to reduction (c).

Chapter 5

Keeping Track of Reductions

Parallel graph reduction is team work. A parallel machine carrying out graph reductions is like a human team whose members are working separately on interrelated parts of a project. The biggest challenge for the team manager would be to coordinate the work of the members. The biggest challenge in parallel graph reduction is coordinating the separate reductions.

The challenge of coordinating tasks in parallel graph-reduction is the subject of Section 5.1. Section 5.2 describes the Id-in-Id optimizer's solution to this problem, called *memobooks*. Memobooks are a general purpose utility for recording information about and coordinating operations on graphs. Examples illustrating the flexibility of memobooks are given in Section 5.3.

5.1 Simple Graphs Tied Into Complicated Knots

Certain operations on graphs require careful coordination. Consider, for example, the following problem. We are given a directed, acyclic program graph. Each node in the graph contains pointers to its children, and the nodes in the graph are not stored in a central-location. The graph data-structure is functional and must be copied in order to reflect changes. We are to perform two complete developments on the program graph. Recall that during a single development, we can only reduce the nodes that were reduceable before the development began.

What would happen if we did not use auxiliary mechanisms to keep track of the development? When we tried to copy a node a to a new node a' , we would need to create copies of a 's children, $c_1 \dots c_j$. We could create these copies, of course, but we would have no way to record which nodes we copied. When we tried to copy another node b that shared a child c_i with a , we would have no way of retrieving the copy of c_i that we made for a' . We would have no choice but to make another copy of c_i . Though b and a shared a common child, b' and a' would not. Without any auxiliary mechanisms, even single developments could not be performed correctly.

Now compare the problem of performing developments on functional graphs to the problem of

performing them on mutable graphs. With functional graphs, the fact that the entire graph must be copied enforces some order on a development. Every node in the functional graph must be copied, and no node can be copied before its children are copied. This dependence that orders operations on a functional graph does not order operations on mutable graphs. The mutation of a parent node does not require the mutation of its child first. The development of mutable graphs contains all of the pitfalls of functional graph development, plus others:

1. It may be that reduction of one node a depends on the reduction of another, b . Without proper coordination, there is no way to detect when the reduction of b is complete. a may therefore examine b before, during, or after its mutation.
2. The reduction of two nodes a and b may be inter-dependent. If we use locks to ensure sequencing of reductions, a and b may end up waiting for each other indefinitely.
3. Two reductions may simultaneously attempt to mutate a memory location. This could result in garbage being written to the memory location or a double-write error. Even if only one of the mutations succeeds, the process that attempted the unsuccessful mutation must be notified that its mutation was unsuccessful. Otherwise this process will believe that the mutated memory location contains one thing, when in fact it contains another.

With multiple, parallel developments, matters are even worse. With functional graphs, a development cannot proceed until a prior development has completed its copy of a graph, but with mutable graphs, all developments may proceed in parallel on the same graph. There is no way to prevent one development from overtaking another. A development has no way of knowing whether the information that is examining is its own result or the results of another development. The development also has no way of knowing whether, as it does make changes to the graph, it is destroying information created by another development. Performing parallel, developments on a mutable graph without careful coordination is a recipe for disaster. All of these problems are summarized pictorially in Figure 5-1.

5.2 Memobooks

If you were to send a team of workers out to simultaneously solve a set of interrelated sub-problems, you might coordinate their work by setting up a memo board on which the workers could post and read messages about the progress of their work. This same idea can be extended to parallel graph reductions. If you placed a memo board on every node on a graph, then parallel reductions could use the memo board to resolve issues such as:

- Has anyone tried to reduce this node yet?

- If someone has tried to reduce this node, is the work finished?
- What has this node been reduced to?

This model works very well for single developments of a graph, but as Barth, Nikhil, and Arvind have shown, it breaks down with multiple developments [20]. When several developments try to use the memo board for different purposes, the developments might get their memos mixed up, and the memo board might soon become cluttered with garbage memos. The solution is to separate the memos from the nodes. By placing memos in the hands of the development, the optimizer can keep a single development from confusing its own memos with those of other developments. A development can also throw away its memos once it has finished reducing a graph. Memobooks are the complete realization of this solution.

Figure 5-2 displays an abstract picture of a memobook. A memobook is a table, and each entry in the table is a memo. Memos contain three fields for storing information: a filled? flag, an object, and a result. The filled? flag is a boolean value that is changed when a memo has been filled. When a memo is created, the filled? flag is always false. Because the filled? flag has M-structure semantics, it is possible to use this flag as an exclusive lock on the memo. The object field of a memo is functional and indicates the object to which a given memo corresponds. Finally, the result field is available to users for storing values. The result field has I-structure semantics. When the memo is created, the result field is empty, and it can be filled only once. Attempts to read the value stored in the result field return a value only when the field has been filled. Complete descriptions of I-structures and M-structures are given in [3] and [20].

When a memobook is first created using the function `MK_MEMOBOOK`, it does not contain any memos. Two functions add information to and request information from memobooks: `PRODUCE_MEMO` and `READ_MEMO`. `PRODUCE_MEMO` takes a memobook, a function, and an object as its arguments. The job of `PRODUCE_MEMO` is to use the memobook to ensure that the function, called a *producing-function*, is only called with the given object as its argument. `PRODUCE_MEMO` does this by looking up the object in the memobook and finding the memo corresponding to that object. If a memo does not yet exist for that object, `PRODUCE_MEMO` creates one and inserts it into the table. `PRODUCE_MEMO` then checks the memo to see whether it has been filled. If not, `PRODUCE_MEMO` marks the memo as filled. It fills the result field of the memo by calling the producing function with the object as its argument. In any case, `PRODUCE_MEMO` reads the value of the result field and returns this value.

`READ_MEMO` takes a memobook and an object as its arguments. The job of `READ_MEMO` is to find the memo for the object and read the value from the result field of the memo. If a memo does not yet exist for that object, `READ_MEMO` creates one and inserts it into the table. It then reads the value in the result field and returns that value, regardless of whether the filled? field of the memo is true or false. `READ_MEMO` can read a memo that has not yet been filled because of the

I-structure semantics of the memo's result field. Figure 5-3 gives the algorithms for these operations and Figure 5-2 depicts some typical operations on a memobook.

The key to making these algorithms work in a parallel environment is to ensure atomicity of two actions. The first action is the insertion of memos into a table. Only one memo should appear in the table for any given object. If the insertion of memos in the table is not atomic, then independent processes may simultaneously discover that no memo exists for an object, and they may insert multiple memos for that object. Atomicity can be ensured by locking the elements of the table during lookups. The second place where atomicity is important is in the production of results for memos. Memobooks must also ensure that only one producing-function is ever called for any given object. Atomic production of values is ensured by using the `filled?` field of a memo as a lock. In order to check whether a memo has been filled, `PRODUCE_MEMO` has to take hold of this lock. Once it has the lock, it is certain that the value of the `filled?` field is the current value, and cannot be changed. If the `filled?` field indicates that the memo is not filled, `PRODUCE_MEMO` changes the value of the `filled?` field and replaces the lock. An Id implementation of memobooks is given in Appendix A.

It is important to note that memobooks do not provide more atomicity than is necessary. Notice what would happen, for example, if we used a global lock to ensure that only one producer examined the memobook at a time? If we called `PRODUCE_MEMO` with function f , we would lock the entire table and prevent anyone else from accessing the memobook until f had completed. The use of a global lock is appealing because it is a simple way to guarantee that double-writes never occur. On the other hand, use of a global lock sequentializes all accesses to the table. Furthermore, if function f contained a call to `PRODUCE_MEMO`, f would end up waiting forever to gain access to the lock.

5.3 Memobooks in Parallel Graph Algorithms

Memobooks were specifically designed so that they do not interfere with the parallelism of the applications that use them. Specifically, they were designed so that producing functions could be composed with each other the way that regular functions can be composed. This is convenient for formulating graph programs recursively. Memobooks were also designed so that consumers could consume memos even before values had been produced for them. The following examples show how memobooks can be used in a variety of parallel applications.

5.3.1 Copying Functional Graphs

First consider the problem of copying a graph like the ones that have been presented before. A recursive algorithm for copying a node in the graph would be to first construct copies of the nodes children, and then use these copies to make a copy of the node itself. This algorithm works perfectly

well for tree data-structures, where two parents can not share a single child. In order to make this algorithm work for graphs, it is important to ensure that that all parent nodes who share the same child node in the original graph point to the same copy of the child node in the new graph.

Figure 5-4 illustrates two functions for copying trees and graphs, `COPY_TREE` and `COPY_GRAPH`. Both functions copy the elements of their data-structures recursively. The main difference between them is that `COPY_GRAPH` is mutually recursive with another function `CHECK_AND_COPY`, rather than with itself. The job of `CHECK_AND_COPY` is to memoize `COPY_GRAPH` using memobooks. Thus `COPY_GRAPH` will only be called for any given node once. Though this is a relatively simple example, it raises several important points. First, the use of memobooks does not interfere with the inherent parallelism of the applications. In both `COPY_GRAPH` and `COPY_TREE`, sibling nodes are copied in parallel and child-nodes are copied before their parents. Second, memobooks are easy to use. Compare the code for `COPY_TREE` with that for `COPY_GRAPH`. Without the use of the memobook, the graph-copier and the tree-copier are almost identical. The memobook in the graph-copy merely ensures that repetitive recursions do not occur. It is often possible to develop a program for graph manipulation from an analogous program for tree manipulation. To make the tree-manipulation program apply to a graph, use a memobook to prevent repetitive recursions.

5.3.2 Constant-folding Cyclic Graphs

An interesting question to ask is, what happens if we use `COPY_GRAPH` to copy cyclic graphs like those given in Figure 5-5 (a) and (b)? Assuming that lenient-order reduction is used to execute the `COPY_GRAPH` algorithm, `COPY_GRAPH` would be able to copy both graphs in Figure 5-5 without deadlocking. This may seem surprising, because `COPY_GRAPH` *memos* *r* would end up calling `COPY_GRAPH` *memos* *r*. Wouldn't `PRODUCE_MEMO` halt on the recursive invocation of the copy procedure and wait for itself indefinitely? Because of lenient-order reduction, `COPY_GRAPH` does not have to complete all of its computations before returning a value to its caller. This means that the first invocation of `COPY_GRAPH` *memos* *r* can return a value to `PRODUCE_MEMO` as soon as it can allocate a pointer for the copy of node *r*. The second invocation would retrieve this pointer from the memobook and the copy would proceed normally.

Now consider the problem of constant-folding a graph. To constant-fold a graph means to replace arithmetic expressions with their equivalent constants, such as reducing $(2 * 3)$ to 6. The algorithm for performing constant-folding using memobooks is given in Figure 5-6. Though the algorithm differs little from the algorithm for copying graphs, `CONSTANT_FOLD` would not be able to constant-fold the graphs given in Figure 5-5 (a) and (b) without deadlocking. The reason that `CONSTANT_FOLD` deadlocks is because `CONSTANT_FOLD` can not return its new node until it has checked the contents of its children's new nodes. Unlike `COPY_GRAPH`, `CONSTANT_FOLD` must first finish executing its recursive calls before it can return a value.

It would be nice if `CONSTANT_FOLD` could detect cycles in graphs and abort any attempt to fold them, rather than hanging forever. One approach to cycle detection used in sequential programming is to keep a list of all nodes traversed along a path, and abort the traversal if a node is ever encountered twice. A modified algorithm for constant-folding using paths to detect cycles is given in Figure 5-7. The modified version of `CONSTANT_FOLD` does not deadlock on the graph in Figure 5-5 (a), but it still deadlocks on the graph given in (b). To see why, imagine that `CONSTANT_FOLD` starts recursively folding the graph at the node marked v . It spawns two separate recursions simultaneously, one that traverses w and one that traverses x . One of these recursions then begins to fold node y , the other node z . So far, none of the paths indicate that a cycle has been traversed. At this point, `CONSTANT_FOLD` deadlocks, as the two paths of recursion wait indefinitely for the other to finish folding w and x .

Unfortunately, many other useful optimizations other than constant-folding exhibit this same behavior when performed on cyclic graphs. The problem of designing a parallel reduction system for cyclic graphs is a difficult one and is beyond the scope of this thesis. One possible solution, to be explored in future work, is to perform cycle-detection as a separate pass before optimizations.

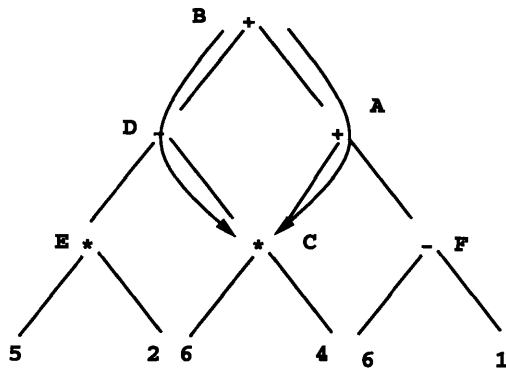
5.3.3 Summing the Values in a Mutable Graph

Now consider a problem that involves the summation of values on a graph. The problem is more complex than copying functional graphs. Suppose that we are given a graph that has been marked with integers. The integer value on any given node can be changed, or mutated. We wish to set the integer field of each node on the graph to be the sum of all the integers of its sub-graphs. Figure 5-8 gives the algorithm for performing the summation using memobooks. In this algorithm, the summing function does not return its result until the mutation is complete. This forces the consumers of a node's sum to wait until the mutation is complete before reading the sum.

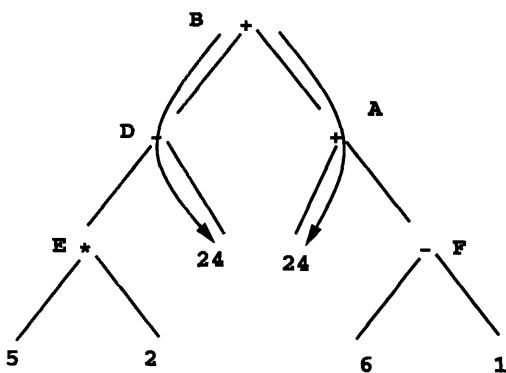
This algorithm works well with a single summation on the graph. Now consider the problem of performing multiple summations on a graph. If we use this algorithm to perform multiple summations in parallel, we will encounter the same problems that arose with multiple developments. There will be no way to keep multiple summations from overtaking each other. It is possible, however, to use memobooks to create a data-dependence between two summations. The second summation can then check on the progress of the first in the memobook, and it can use the information it finds to throttle its own progress and ensure that it never overtakes the first summation. With memobooks, it is often possible to pipeline two operations on a graph. The algorithm for performing a pipelined summation using memobooks is also given in Figure 5-8.

Memobooks are a very useful, flexible abstraction. They can be used in applications where the dependency of operations is very well structured, such as copying a functional graph, or where they are not, such as multiple summations of mutable graphs. The next chapter explores the difficulties

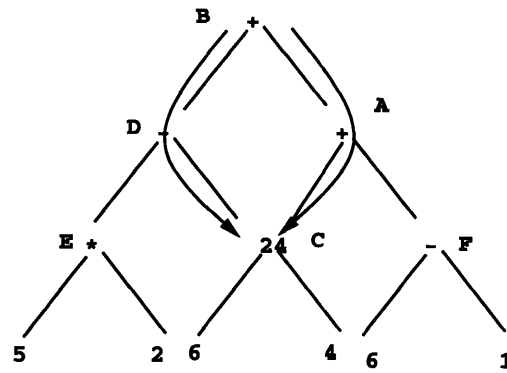
of designing parallel reduction strategies for optimization. Memobooks provide enough flexibility to fulfill the complicated synchronization requirements of these strategies.



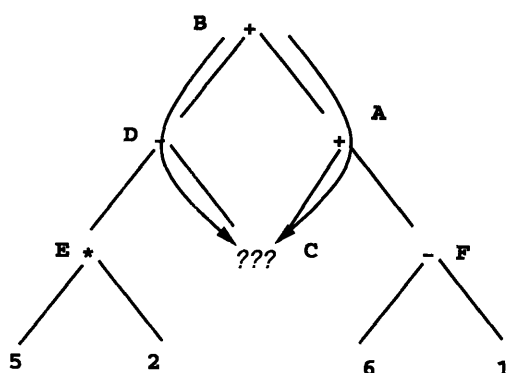
Reductions can proceed in parallel along the paths of the arrows.



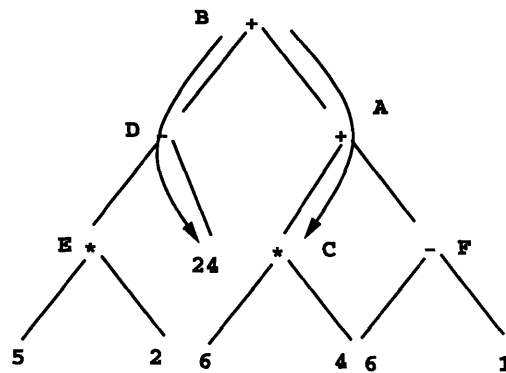
Both reduction paths succeed in reducing c. The expression corresponding to c has been duplicated.



Both paths try to mutate c at the same time. The resulting expression, (24 6 4), does not make sense.



Both paths wait for the other to complete the reduction. The result is a deadlock.



Only one path succeeds in reducing c. The result of this reduction is not communicated to the path through a.

Figure 5-1: Possible outcomes of parallel reduction.

<i>object</i>	<i>filled?</i>	<i>result</i>

When memobooks are created they do not contain any memos. Each row in the above memobook is a single memo.

<i>object</i>	<i>filled?</i>	<i>result</i>
a	Y	3

A producer can tell a memobook to fill in a memo using a producing-function. Only one producer may ever fill in a given memo.

<i>object</i>	<i>filled?</i>	<i>result</i>
a	Y	3
b	N	-

A consumer can ask a memobook for the result of a memo. If the result does not yet exist, as is the case for object b above, the consumer waits on the result.

<i>object</i>	<i>filled?</i>	<i>result</i>
a	Y	3
b	N	-
c	Y	-

In the memobook above, the result for c is marked filled but does not yet have a result. This is because the producing function has not yet returned a result for c.

Figure 5-2: Producer-consumer interaction with memobooks.

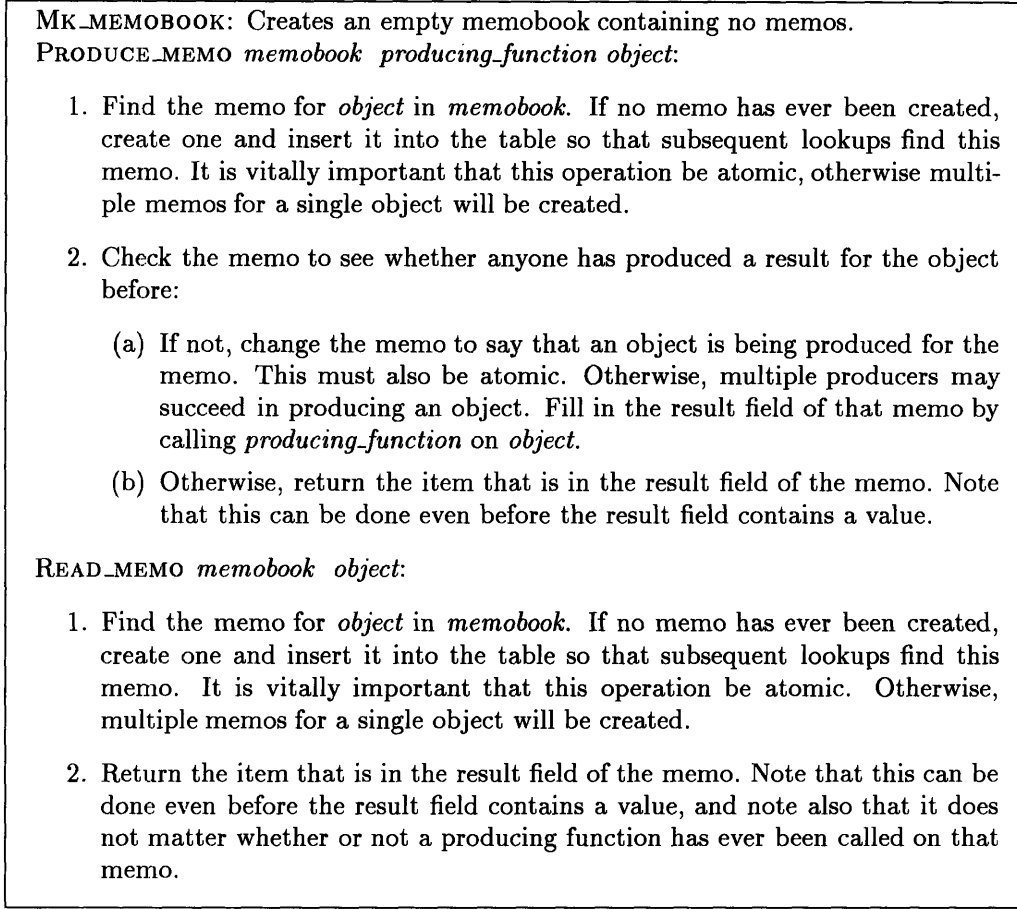


Figure 5-3: The operations PRODUCE_MEMO and READ_MEMO.

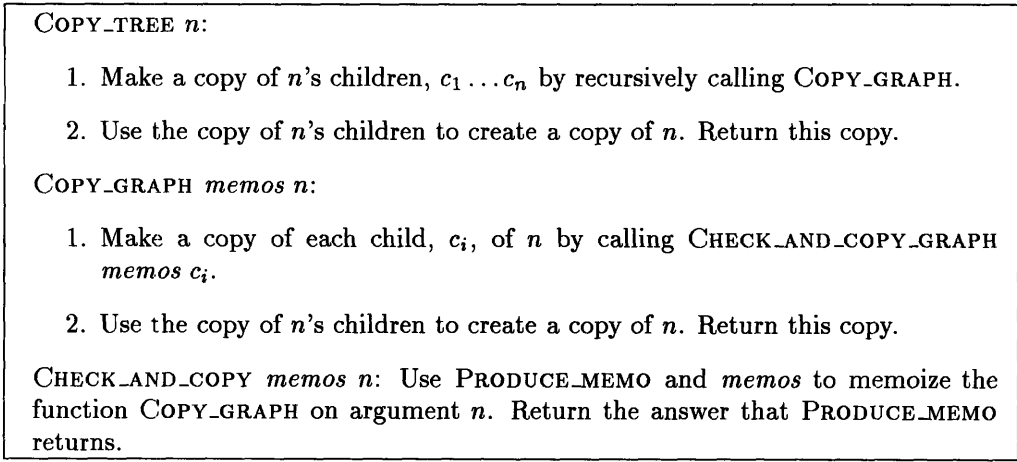


Figure 5-4: The COPY_TREE and COPY_GRAPH operations for copying trees and graphs. COPY_GRAPH uses memobooks to avoid copying a node more than once.

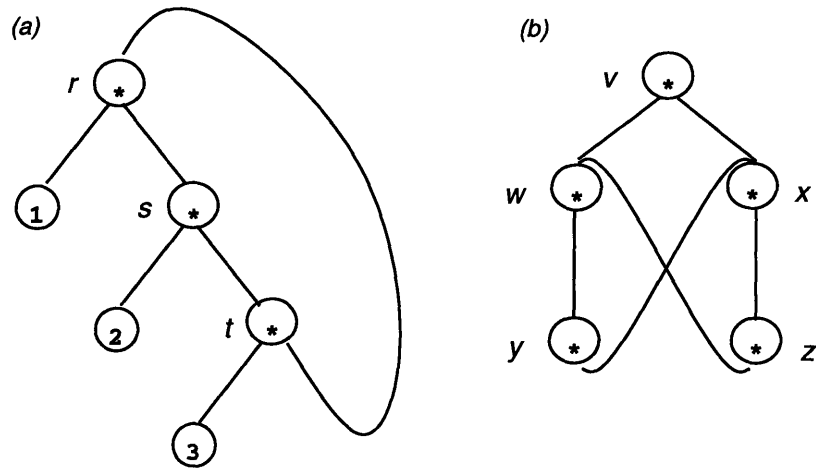


Figure 5-5: Two cyclic graphs.

CONSTANT_FOLD *memos n*:

1. Constant fold each child, c_i , of n by calling **CHECK_AND_CONSTANT_FOLD** *memos* c_i .
2. If n 's has tag p , where p is a primitive binary operator, and c'_1 and c'_2 are constants, apply p to the constants. Return the constant that results.
3. Otherwise, use the copy of n 's children to create a copy of n . Return this copy.

CHECK_AND_CONSTANT_FOLD *memos n*: Use **PRODUCE_MEMO** and *memos* to memoize the function **CONSTANT_FOLD** on argument n . Return the answer that **PRODUCE_MEMO** returns.

Figure 5-6: The **CONSTANT_FOLD** operation for constant-folding graphs.

CONSTANT_FOLD *memos l n*:

1. Create a new path list, l' , by appending n to l .
2. Constant fold each child, c_i , of n by calling **CHECK_AND_CONSTANT_FOLD** *memos l' c_i*.
3. If n 's has tag p , where p is a primitive binary operator, and c'_1 and c'_2 are constants, apply p to the constants. Return the constant that results.
4. Otherwise, use the copy of n 's children to create a copy of n . Return this copy.

CHECK_AND_CONSTANT_FOLD *memos l n*:

1. Check to see whether n is a member of list l . If n is a member, we have hit a cycle. Abort by returning n immediately.
2. Otherwise, use **PRODUCE_MEMO** and *memos* to memoize the function **CONSTANT_FOLD** on arguments l and n . Return the answer that **PRODUCE_MEMO** returns.

Figure 5-7: A modified **CONSTANT_FOLD** operation that uses paths to detect cycles.

SUM_GRAPH *memos n*:

1. If n is a leaf node, return n 's value, v , without mutating n .
2. If n is not a leaf node, for each child c_i of n , compute the sum, s_i , of that child by calling **CHECK_AND_SUM** *memos c_i*.
3. Sum all s_i 's with v to get s , the sum for n .
4. Mutate n 's value to be s . Only after this mutation has completed, return s .

CHECK_AND_SUM *memos n*: Use **PRODUCE_MEMO** and *memos* to memoize the function **SUM_GRAPH** on argument n . Return the answer that **PRODUCE_MEMO** returns.

PIPELINE_SUM_GRAPH *memos1 memos2 n*:

1. If n is a leaf node, return n 's value without mutating n .
2. If n is not a leaf node, for each child c_i of n , compute the sum, s_i , of that child by calling **CHECK_AND_PIPELINE_SUM** *memos1 memos2 c_i*.
3. Find the current value of n , v , by calling **READ_MEMO** *memos1 n*.
4. Sum all s_i 's with v to get s , the sum for n .
5. Mutate n 's value to be s . Only after this mutation has completed, return s .

CHECK_AND_PIPELINE_SUM *memos1 memos2 n*: Use **PRODUCE_MEMO** and *memos2* to memoize the function **PIPELINE_SUM_GRAPH** on argument n . Return the answer that **PRODUCE_MEMO** returns.

Figure 5-8: Two functions, **SUM_GRAPH** and **PIPELINE_SUM_GRAPH**, for summing values stored on a graph.

Chapter 6

Optimization Strategies

The goal of all optimizers, sequential or parallel, is to optimize programs by eliminating all redundant operations. Optimizers never achieve this ideal, however. Instead, they use internal standards to determine when a program has been improved enough to be passed from one stage of compilation to the next. In a sense, then, compiler designers must build a definition of “optimal” into their optimizers. This chapter discusses the difficult issues involved in defining an optimal program state and deciding how much optimization is enough. The chapter also discusses strategies used to achieve optimization and describes the specific strategy used in the Id-in-Id optimizer

6.1 Strategies for Optimizing with a Single Reduction Rule

Although the Id-in-Id optimizer uses a set of rules in carrying out optimizations, it is instructive to begin our consideration of optimization strategies with a strategy that uses only one reduction rule. Because of its long history in functional programming, the beta-reduction rule is the logical rule with which to start. Modern functional compilers often use this rule in optimizing programs, and it is the basis of the compiler optimization technique called compile-time interpretation, or partial evaluation. In this technique, the compiler converts input programs to graphs and feeds them to an interpreter. This interpreter, in turn, interprets the program as much as possible using the beta-reduction rule. When no further interpretation of the program is possible, the next phase of compilation proceeds.

Is it possible to improve on this approach to optimization without adding more optimizing reduction rules to the system? Figure 6-1 (a) shows a Kid program, before and after interpretation by a beta-reduction interpreter. Note that the interpreter does not perform reductions within the top-level definition for f because f is not applied to any arguments. A quick look shows that interpretation within the body of f could yield a reduction of function calls within f . One way to improve this system therefore would be to allow interpretation within a function definition, even if

the function is not applied. Figure 6-1 (b) shows another optimization of the same Kid program, this one produced by an interpreter that interprets within unapplied function definitions.

This approach seems to be an improvement. Why not continue interpreting until all the reductions in the program have been performed? Figure 6-2 shows a Kid program, before and after interpretation by an interpreter that uses this approach. Note, however, that this after-picture could not be produced by an actual machine because a real interpreter would go on interpreting this program forever.

(a)	<pre> Def f b = b * (g 1 2) + (g 2 3) ; def g a b = (a + b) ; </pre>	→	<pre> Def f b = b * ((g 1 2) + (g 2 3)) ; def g a b = (a + b) ; </pre>
(b)	<pre> Def f b = b * (g (g 1 2) (g 2 3)) ; def g a b = (a + b) ; </pre>	→	<pre> Def f b = b * ((1 + 2) + (2 + 3)) ; def g a b = (a + b) ; </pre>

Figure 6-1: Optimization of Kid programs using an interpreter. A real interpreter would not optimize the function `f` in (a) because it has not been applied to any arguments. However, it would be useful to interpret expressions in `f`, as in (b).

<pre> Def f n = if (n == 1) then 1 else (n + (f (n-1))) ; </pre>	→	<pre> Def f n = if (n == 1) then 1 else (n + if ((n-1) == 1) then 1 else ((n-1) + if (((n-1)-1) == 1) then 1 else)) </pre>
--	---	--

Figure 6-2: An example of too much compile-time interpretation.

Conventional compilers that do not use interpretation employ a different strategy for optimizing programs. The approach used in these compilers is often more practical than reduction-based interpretation. These optimizers usually repeat the cycle of optimizations a fixed number of times, and programs therefore usually have no influence on the number of optimization cycles. In such compilers, the definition of an optimal program is simply “a program that has iterated over n times by the optimizer,” where n is a fixed number. Compiler designers usually select n through trial and error.

In addition to deciding on a termination strategy, a compiler designer must make other decisions in designing a reduction strategy. Efficiency is one major concern, and correctness is crucial. For instance, using a normal-order reduction strategy to reduce a program that executes using applicative-order at run-time would be only partially correct. That is because a normal-order reduc-

tion strategy might remove infinite loops from programs that would otherwise execute at run-time using an applicative-order reduction strategy. Even with a single reduction rule, the number of factors and choices to consider when choosing an optimizing reduction strategy is bewildering.

6.2 Adding Reduction Rules

When multiple reduction rules are used to reduce programs, choosing a reduction-strategy for optimizations becomes even more difficult. The first problem with reduction using multiple rules is normalization. A reduction strategy is said to be normalizing if it guarantees reduction to normal form of all programs that have a normal form. When multiple reduction rules are used to optimize programs, however, a normal form program is a program to which none of the entire set of reduction rules applies. Such programs may not exist because one reduction rule may always produce programs that are the redex of another reduction rule. The use of multiple reduction rules forces us to rethink conventional notions of terminating reductions.

The second problem with multiple reduction rules is that at a given step in reduction, a single expression may be the redex of several different rules. Is the order in which the rules are applied important? One needs only to look at conventional compiling models to see that the answer must be yes. Conventional compiling models usually optimize programs in phases. The entire program is first optimized using one particular optimization; then it is optimized using another and another and so on. Researchers have tried to determine which optimization strategies are most effective and how optimizations should be done within and between phases for each compiler. One conclusion that can be drawn from this research is that the ordering of different optimizations has a significant impact on the effectiveness of the optimizer. An optimizing reduction strategy should therefore specify an ordering on the rules themselves, not only on the redexes available in the program.

When program reduction is used in optimization, the resulting model is a cross between classical reduction and conventional compilation. Clearly, the two camps differ on the subject of reduction strategies, and there seems little hope for compromise. If a compiler limits reductions to a fixed number of iterations, it would be impossible for theorists to prove anything about the optimality of the compiler's output programs. Surely this approach is more practical than continuing reductions until some provably optimal program is produced, however. Taking other factors into account, such as correctness, parallelism, interference of multiple rules, and inter-optimization strategies, confuses matters even more. The problem of choosing a reduction strategy for the Id-in-Id compiler is more difficult than just choosing an "off-the-shelf" strategy.

6.3 Lenient Reduction in the Id-in-Id Optimizer

The strategy used by the Id-in-Id optimizer to reduce programs when multiple reductions are present is modeled on the strategy used in interpretation of Id programs. In almost all cases, the strategy is lenient. This strategy gives some assurance of normalization, although the guarantee is not absolute. A lenient strategy also allows reductions to be performed in parallel. Most important, a lenient strategy allows newly created redexes to be considered during a single reduction. This makes it a very efficient strategy. More reductions are performed in a single pass than are possible with strategies that put newly created redexes off-limit.

Let a single pass of optimization be defined to be a single lenient-order reduction of the entire program. A single lenient-order reduction involves the recursive, parallel reduction of all the available redexes in the program, plus new redexes that occur during reduction. Because of the interference of rules, however, it is also possible for the program to be in normal-form with respect to only certain rules at the end of a single pass. Still, almost all of the available optimizations can be achieved by a second single pass. Two single passes are enough to reduce the majority of optimizations available in the program. The number of reductions performed in a single pass of the Id-in-Id optimizer is then dependent upon the program itself, just as in classic program reduction. The Id-in-Id optimizer performs a fixed number of single passes, however, as in a conventional optimization.

When we started designing the Id-in-Id optimizer, all we had to work with was an abstract language for describing Id programs, Kid, and a set of reduction rules for optimizing them. We then developed a concrete representation for Kid programs as graphs, a method for performing single reductions on these graphs, a method for coordinating reductions, and a reduction strategy for optimizations. Now it is time to pull all of these elements together to present a general algorithm for optimizations.

Performing a Single Pass. The general algorithm for performing a single pass of optimizations in the Id-in-Id compiler is shown in Figure 6-3. The single-pass engine is made up of two functions, `REDUCE` and `CHECK_AND_REDUCE`. `REDUCE` is the function that actually performs optimizations on the graph. `CHECK_AND_REDUCE` uses a memobook to ensure that `REDUCE` is only called once for every node on the graph.

The function `REDUCE` takes four arguments, a continuation function, an optimization environment, a memobook that memoizes the reductions, and the Kid Graph expression to be reduced. If the input expression is an expression other than a free-node or a lambda-node, `REDUCE` reduces the sub-expression of the input expression using `CHECK_AND_REDUCE`. If the input expression is a free-node, then the free-node's child falls in a different scope from the scope of the free-node itself. Instead of reducing this node using `CHECK_AND_REDUCE`, `REDUCE` recursively reduces the node using the continuation function. Likewise, whenever reduction proceeds into a new lambda definition,

REDUCE creates a new continuation function and a new optimization environment for optimizing the nodes within that lambda. After the sub-expressions have been reduced, REDUCE checks the new sub-expressions and the optimization environment to see whether the input node can be optimized to another node. If so, it performs a mutating UNION on the input node and the optimized node. If the input node cannot be optimized, REDUCE returns the input node without making a copy of it or mutating it.

CHECK_AND_REDUCE *cont env memos v*:

1. Call the function REDUCE with the above arguments using the memobook function PRODUCE_MEMO to ensure that REDUCE is only called once per node.
2. Return the result given by PRODUCE_MEMO.

REDUCE *cont env memos v*:

1. Reduce the children, $c_1 \dots c_n$, of v to $c'_1 \dots c'_n$:
 - (a) If v is a free-node, use *cont* to reduce v 's child, c_1 .
 - (b) If v is a lambda-node, create a new continuation, *new_cont* using *env*. Create a new environment, *new_env* and reduce v 's children using CHECK_AND_REDUCE, *new_cont*, *new_env*, and *memos*.
 - (c) If v is neither a free-node nor a lambda-node, call CHECK_AND_REDUCE with the above arguments to obtain $c'_1 \dots c'_n$.
2. Use $c'_1 \dots c'_n$, optimizing reduction rules, and the current optimization environment *env* to determine whether v can be optimized.
 - (a) If v can be optimized:
 - i. Determine v' , the Kid Graph expression that v reduces to given that v 's children are now $c'_1 \dots c'_n$ and the current environment is *env*.
 - ii. Reduce v to v' by calling UNION, the parallel disjoint-set union operation.
 - iii. Return v' , the optimized version of v .
 - (b) If v is not reduceable, return v as the optimized version of v .

Figure 6-3: Algorithm for performing a single pass.

With mutable graphs, problems arise in using memobooks that do not occur when memobooks are used only with purely functional graphs. For example, unification of a node A with a node B changes the identity of node A . This is problematic because PRODUCE_MEMO uses the identity of nodes to insert and retrieve memos from the table. Imagine that the optimizer traverses node A , and A is entered into the memobook and marked as traversed. Subsequently, another reduction unifies A with B , a node that has never been traversed. An entry for A exists in the table, but no entry exists for B . Anyone attempting to traverse A would then look up B in the table and find

that it had never been traversed.

There are several ways to get around this problem. The Id-in-Id optimizer solves it by inserting a dummy memo in the table for B before unifying A with B . The optimizer thus ensures that unifications are only performed on nodes that are marked as traversed.

Performing Multiple Passes. Multiple single passes also require special attention. It might seem at first that multiple passes could be pipelined using memobooks, an approach that was illustrated in Chapter 5. Although pipelining is an ideal approach in many respects, there is one insuperable obstacle to the approach: Bottom-up traversals cannot be pipelined when the traversals can change the shape of the graph. Here *bottom-up* refers to the fact that the children of a node are always traversed before the parent. The traversal performed by the summing procedure given in Figure 5-8 sums the values on a graph bottom-up, but the summation does not alter the structure of the graph, only the values that are stored on the graph. Reduction, on the other hand, does change the shape of the graph. This means that it is not possible to determine the structure of the top node until all of the nodes below it have been reduced. Try as you may to pipeline the traversals, the result would be that one traversal would run to completion before the next traversal began.

<p>OPTIMIZE g:</p> <ol style="list-style-type: none">1. Create a new memobook, $memos1$, optimization environment, $env1$, and dummy continuation, $cont1$, for the first reduction.2. Reduce g by calling CHECK_AND_REDUCE $cont1 env1 memos1 g$.3. Using a barrier, wait for CHECK_AND_REDUCE to completely terminate.4. Create a new memobook, $memos2$, optimization environment, $env2$, and dummy continuation, $cont2$, for the first reduction.5. Reduce g by calling CHECK_AND_REDUCE $cont2 env2 memos2 g$.6. Using a barrier, wait for CHECK_AND_REDUCE to completely terminate.7. Return g.

Figure 6-4: Algorithm for performing two single-passes.

It would be wasteful to use elaborate pipelining mechanisms to ensure that one traversal finishes completely before another begins. It is simpler to use barriers to ensure that one set of computations terminates completely before another set of computations stops. In addition, the Id-in-Id optimizer does not make any attempt to run multiple single-passes in parallel. Parallelism is allowed within single passes. Using barriers, each single pass is performed sequentially.

The Id-in-Id optimization strategy may seem like it was arbitrarily chosen. This is almost true. Even when past research into optimization and reduction strategies are taken into account,

a confusing array of possible strategies remain. The Id-in-Id optimizer design resembles that of the conventional compiler—the choice of optimization strategy was made mostly by trial and error. Fortunately, as the next chapter shows, it turns out that the Id-in-Id optimization strategy is able to optimize programs quite well.

Chapter 7

Results

There are several different platforms which can be used to test and run Id programs. GITA was the first platform ever available for running Id programs. GITA is a simulator for the Tagged-Token Dataflow (TTDA) Architecture. In the TTDA architecture, instructions are represented by tokens with inputs and an output. The machine contains waiting-matching units to route tokens for execution when their inputs arrive. The term Tagged-Token comes from the fact that each token carries a tag that indicates to the waiting-matching units to which instruction the token is headed. GITA simulates a TTDA machine with an infinite number of processors and zero communication overhead between processors. GITA provides support for profiling the performance of Id programs and is outlined in [2].

Monsoon, the successor of GITA, is a real dataflow machine for executing Id programs. Monsoon's waiting-matching units differ from GITA in that they allocate "frames" of storage for entire blocks of instructions, rather than on a per instruction basis. This aspect of Monsoon is considered a significant improvement, in terms of resource management, over GITA. MINT is a simulator for Monsoon that, like GITA, simulates a Monsoon dataflow machine with infinite processor resources and zero communication overhead between processors. MINT and Monsoon support statistics collection and are outlined in [14] and [7].

Although it is possible to run Id programs on single, commercial processors using GITA and MINT, their speed is limited by the layers of simulation that they provide. A recent approach to compiling Id code has been to use a compiler to partition Id programs into sequences of instructions, called threads. These threads can, in turn, run on stock hardware. The Id2TLO compiler is one such compiler and it can compile Id code into threaded code for TAM. The Threaded Abstract Machine (TAM), developed at the University of California at Berkeley, is an abstract machine that dynamically schedules programs that have been partitioned into threads. Current implementations of TAM, and therefore Id, run on sequential SPARC processors and also the CM5. The most important

result of this work is that it is possible, using TAM, to get acceptable sequential performance out of Id code.

Perhaps the anticlimax to the Id-in-Id compiling project is that, of these various platforms, only one platform is suitable for running the Id-in-Id compiler. The Id-in-Id compiler comprises approximately 32 KB of source code, making it by far the most significant body of Id code ever written. Compiled with the Id2TL0 compiler, which uses C as an intermediate language, the Id-in-Id compiler comprises 32 MB of object code. Both GITA and MINT are simulators and the Id-in-Id compiler would take a day to compile even the most trivial program running on these simulators. The CM5 and Monsoon are both real parallel machines. They provide the fastest execution vehicles for Id programs. However, these machines do not provide any support for virtual memory. Since code-size makes it impossible to load even the parser tables of the Id-in-Id compiler into the program memory of one of these machines, it is not possible to run the entire compiler on these machines. The only platform that fits all of the requirements for running the Id-in-Id compiler is a commercial, sequential processor. The compiler that compiles parallel programs in parallel currently runs on a Sun workstation.

This chapter attempts to provide some assessment of the Id-in-Id compiler optimizer. Section 7.1 discusses some tests of the Id-in-Id compiler running on a Sun workstation. Though the tests cannot reveal anything about the parallelism of the Id-in-Id optimizer, they do say something about the effectiveness of its optimizations. Section 7.2 discusses the cost of using memobooks to perform memoization, as measured on a GITA simulator. Finally, Section 7.3 discusses the advantages of using parallel disjoint-unions in graph-reduction.

7.1 The Id-in-Id Compiler Running on a Sun-Workstation

The Id-in-Id compiler, running on a Sun workstation, compiles Id source code to Tagged-Token Dataflow Architecture (TTDA) graphs. Another compiler, called the Id-in-Lisp compiler, was developed several years before the Id-in-Id compiler and is written in Lisp. This compiler can also compile Id code to TTDA graphs. By compiling benchmark programs with each compiler and running their TTDA output on GITA, it is possible to compare the quality of the code that each compiler produces.

Table 7.1 shows statistics that were collected by compiling several benchmark Id programs to TTDA and running them on the GITA simulator. The table shows how each program performed when it was compiled with the Id-in-Id compiler and the Id-in-Lisp compiler, with compiler optimizations off and compiler optimizations on. The table shows that the Id-in-Id compiler is almost as effective as the Id-in-Lisp compiler at performing optimizations. One of the chief differences between the Id-in-Id compiler and the Id-in-Lisp compiler is that one uses DeBruijn chains to represent programs, the other uses super-combinators. These figures seem to indicate that it is possible to

perform just as many optimizations using DeBruijn chains as super-combinators. This is an encouraging result, given that these figures were taken when the Id-in-Id compiler was in its infancy. The Id-in-Lisp compiler, the group's work-horse, had been fine-tuned to produce optimal code over the course of many years.

<i>Benchmark program</i>	Floating-Point Ops		I-Fetches		<i>Critical Path</i>	
	Id-in-Lisp	Id-in-Id	Id-in-Lisp	Id-in-Id	Id-in-Lisp	Id-in-Id
Matrix Multiply (unoptimized)	4,800	4,800	3,206	3,206	754	527
Matrix Multiply (optimized)	3,240	3,240	0	8	435	459
Paraffins (unoptimized)	510	510	5,984	5,320	3,986	1,511
Paraffins (optimized)	252	396	4,878	5,228	1,516	1,512
SIMPLE (unoptimized)	40,390	38,675	79,353	80,123	807	622
SIMPLE (optimized)	38,613	38,576	31,294	32,909	583	625

Table 7.1: Performance of programs compiled by Id-in-Id and Id-in-Lisp compilers.

7.2 Memoized Fibonacci and Factorial on GITA

If you observed the Id-in-Id compiler compiling a program on a Sun workstation and the Id-in-Lisp compiler compiling the same program on a similar workstation, you would observe that the Id-in-Id compiler takes 3 to 5 times longer to compile the program than the Id-in-Lisp Compiler. Though this is an interesting observation to make, it isn't possible to draw any significant conclusions from it. The comparative slowness of the Id-in-Id compiler might be due to many factors. The two compilers have very different designs and therefore very different implementations. The compilers are written in different languages. Id is an implicitly parallel language. Id programs are not intended to run on sequential machines, Lisp programs are.

More interesting observations about the performance of the Id-in-Id compiler can be made by testing individual pieces of the Id-in-Id compiler on parallel simulators or parallel machines. For example, memobooks seem to provide an elegant abstraction for performing memoization in a language like Id, but how practical are they?

The code in Figure 7-1 shows two versions of factorial, the program for computing the factorial of a number n , written in Id. The first version, `fact1`, is recursive with only itself. The second version is mutually recursive with a memoized factorial function `check_and_fact`. `check_and_fact` checks its input argument `n` to see whether `fact2 n` has been calculated before. The implementation of memobooks used for these examples is given in Appendix A. If so, `check_and_fact` returns the result of that calculation, rather than re-calculating `fact n`. Examination of factorial shows that calling factorial of n results in the calculation of factorial of $n - 1$, then $n - 2$, and so forth. Calling factorial of n will never result in the calculation of any $n - i$ twice. In this example, the memoization

of factorial in `fact2` is superfluous.

Table 7.2 shows the results obtained by simulating both of these factorial programs on MINT. As expected, the relationship between the number of instructions executed and n , the input to factorial, is linear. It takes approximately 40 instructions to perform an unmemoized call to `fibonacci`. It takes approximately 200 instructions to perform a memoized call to `fibonacci`. We can deduce from these results that performing the memoization adds 160 instructions on to the time required to execute a normal function call. Performing a memoized call requires 5 times as many instructions as an unmemoized call.

```
def fact1 n =
  if (n == 0) then 1
  else (times~int n
        (fact1 (n-1))) ;

def fact2 memos n =
  if (n == 0) then 1
  else (times~int n
        (check_and_fact memos (n-1))) ;

def check_and_fact memos n =
  (produce_memo (fact2 memos) memos n) ;
```

Figure 7-1: Un-memoized factorial, `fact1`, and memoized factorial, `fact2`.

<i>Input N/ Number of Calls</i>	Critical Path		Total # Instructions	
	<code>fact1</code>	<code>fact2</code>	<code>fact1</code>	<code>fact2</code>
1	40	670	105	1,394
25	420	2,100	1,090	6,194
50	820	4,080	2,114	11,194
75	1,220	6,050	3,139	16,194
100	1,620	8,040	4,164	12,194

Table 7.2: Performance results of `fact1` and `fact2`.

If the cost of performing a memoized function call is so high, why use them at all? For some examples memoization using memobooks may be worth the cost. Figure 7-2 shows two functions for calculating fibonacci numbers, one that is memoized and one that is not. Unlike factorial, recursive calls to `fibonacci` do require computing a single fibonacci number more than once. In fact, the number of redundant calls to `fibonacci` performed by the `fib1` is exactly $fib(n) - n$. Adding memoization to `fibonacci` is useful, because recursive calls can re-use results calculated by other calls. Given an input of n , the second version of `fibonacci`, `fib2` does not perform any redundant calls. The total number of calls performed by `fib2` is n . Table 7.3 shows the results of executing these programs on various input values for n . As expected, the use of memoization pays off for sufficiently large n .

In conclusion, if there is a lot of redundancy in the input program, memoization using memobooks can be worthwhile. On the other hand, if there isn't very much redundancy, then there is a big penalty for memoizing things that don't need to be memoized. When memobooks are used to

perform graph-reduction, the amount of redundancy depends entirely on the input program graphs. Sometimes there will be redundancy, other times not. Even though the Id-in-Id compiler does not run on a parallel machine, it is probably safe to assume that any improvement in the efficiency of memobooks would greatly improve the general efficiency of the compiler.

```

def fib1 n =
  if (n == 0) then 0
  else if (n == 1) then 1
  else (plus~int (fib1 (n-1))
          (fib1 (n-2))) ;

def fib2 memos n =
  if (n == 0) then 0
  else if (n == 1) then 1
  else (plus~int (check_and_fib memos (n-1))
          (check_and_fib memos (n-2))) ;

def check_and_fib memos n =
  (produce_memo (fib2 memos) memos n) ;

```

Figure 7-2: Un-memoized fibonacci, `fib1`, and memoized fibonacci, `fib2`.

<i>Input N</i>	Number of Calls		Critical Path		Total # Instructions	
	<code>fib1</code>	<code>fib2</code>	<code>fib1</code>	<code>fib2</code>	<code>fib1</code>	<code>fib2</code>
1	1	1	30	670	74	1,365
3	3	4	100	670	466	2,563
8	21	8	200	670	3,343	4,127
12	12	12	290	740	23,080	5,691
16	987	16	390	940	158,362	7,255

Table 7.3: Performance results of `fib1` and `fib2`.

7.3 Comparing Functional Reductions to Disjoint-Set Unions on GITA

The previous experiments showed that, depending on the application, the cost of memobooks may outweigh their usefulness. It is interesting to examine whether the same might be said for using mutable disjoint-set unions to perform reductions on graphs. One way to test disjoint-set data-structures is to construct several interpreters for simple lambda-calculus graphs. All the interpreters would use the beta reduction-rule to decide which nodes to reduce and the same reduction strategy for applying the rule. The interpreters would differ in that each would use a different scheme, chosen from the following three schemes, to make changes in the graphs:

Scheme 1 Make a clean copy of each node in the graph to form the new graph, regardless of whether the original node was reduced or not.

Scheme 2 Copy nodes conservatively. Only copy nodes that are reduced or ancestors of nodes that are reduced.

Scheme 3 Copy only nodes that are reduced. Use disjoint-set union to mutate the unreduced node to its reduced copy.

Each interpreter could be run on a variety of graphs and compared. Ideally, we would expect the first scheme to perform the best on an input graph where all the nodes in the graph must be reduced. We would expect the third scheme to perform the best on an input graph requiring few or no reductions. We would expect the second scheme to fall somewhere between the two.

There are several problems with the above experiment, however. First, the experiment tests more aspects of reduction than the cost of performing transformations. It also tests the time each interpreter takes to make decisions about reductions. This includes the time that it takes to obtain information, to test the precondition of the reduction rule, and to construct the results of a reduction. Any data that the experiment produces may be influenced more by these factors than by the different schemes used to perform transformations. More importantly, it would be difficult to characterize the input graphs for such an experiment in a useful way. Several graphs might differ not only in their size and number of possible reductions, but also the number of reductions in the graph that can from other reductions, the size of individual reductions, the number of irreducible nodes that are ancestors of reduceable nodes, etc. With so many ways to vary the input graphs, it would be difficult to set up systematic tests to perform.

Consider instead the following experiment. We are given an ordinary binary graph. An oracle has thoroughly analyzed this graph and has figured out exactly which nodes in this graph need to be reduced. It has marked the nodes to be reduced with “Reduce!” It is the job of the reducer to actually carry out these transformations. In this experiment, the cost of making decisions about reductions has been minimized. In order to further reduce extraneous costs, we could specify that to perform a reduction, the reducer must simply change a node marked “Reduce!” to a node marked “Reduced”. A figure illustrating such a reduction is shown in Figure 7-3.

Simplifying the experiment this way ensures the time each reducer takes to reduce a graph is made up primarily by the time it takes to perform individual transformations. A desirable result of this is that each input graph for this experiment can be quantified using three characteristics. Each graph can be quantified by the total number of nodes, the number of “Reduce!” nodes, and the number of non-reduceable ancestors of “Reduce!” nodes it contains. The graph given in Figure 7-3 (a) contains a total of 8 nodes, 2 of which are “Reduce!” nodes and 2 of which are non-reduceable ancestors of “Reduce!” nodes. Table 7.4 summarizes ten input graphs in terms of these characteristics. Table 7.5 shows the result of reducing these graphs, using each of three different transformation schemes. The Id code for implementing these reductions is given in Appendix B.

The previous examination of memobooks showed that the extra work required to use memobooks may outweigh their usefulness. Fortunately, the results given in Table 7.5 show that using disjoint-set unions to avoid copying does pay off. The results show that in those examples where there are

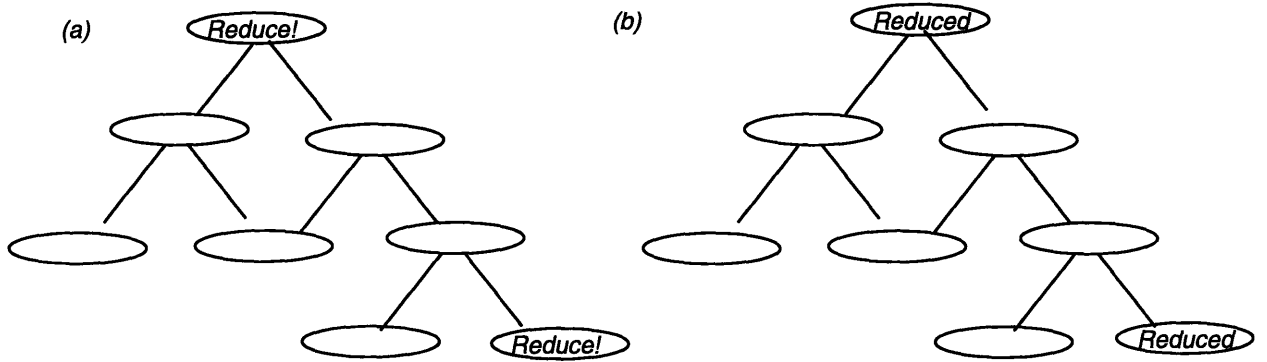


Figure 7-3: A graph with nodes marked to be reduced (a) and the same graph after it has been reduced (b).

<i>Graph Name</i>	<i>Size</i>	<i>Reduce! nodes</i>	<i>ancestors</i>
Graph1	20	0	0
Graph2	20	5	5
Graph3	20	10	10
Graph4	20	20	0
Graph5	41	0	0
Graph6	41	2	39
Graph7	41	5	5
Graph8	41	10	5
Graph9	41	18	20
Graph10	41	41	0

Table 7.4: Ten input graphs and their characteristics.

few changes, mutation requires the fewest number of instructions. For those examples where the entire graph changes, mutation takes the most number of instruction. Mutation seems to pay off, except in the examples where the graph is dominated by changes. In terms of optimization, these results are encouraging. Though optimization often requires making many, many changes to graphs, the number of changes is usually small compared to the size of the input graph.

<i>Graph Name</i>	Critical Path			Total # Instructions		
	Scheme 1	Scheme 2	Scheme 3	Scheme 1	Scheme 2	Scheme 3
Graph1	700	780	1,090	6,060	10,771	4,820
Graph2	700	790	1,090	9,214	10,829	4,820
Graph3	700	870	1,090	9,214	12,038	4,820
Graph4	700	700	2,770	9,214	9,214	16,948
Graph5	3,350	3,760	1,540	18,440	21,622	5,259
Graph6	3,350	3,910	1,540	18,440	23,020	5,384
Graph7	3,350	3,770	1,590	18,487	22,284	5,428
Graph8	3,350	3,410	1,800	18,492	21,189	5,928
Graph9	3,250	3,290	1,600	18,457	20,728	5,398
Graph10	3,350	3,350	5,380	18,440	18,440	25,614

Table 7.5: Performance results of different transformation schemes.

Chapter 8

Conclusions and Future Work

If you were to ask a group of compiler writers to list the sequential languages in which they would be willing to implement a compiler, you would probably get a very short list. Ask the same group for a list of parallel languages in which they would be willing to implement a compiler, and you would probably get an empty list. Compiler designers are smart. Compiler design is daunting enough in a sequential world.

Yet a compiler that can execute on parallel machines has been successfully implemented. The compiler is not just a “toy-compiler.” It works, and it works well. It uses sophisticated compiling techniques such as parallel graph-reduction, and it produces high-quality code. If for no other reason, the Id-in-Id compiler is remarkable because parallel compilation seems so unlikely a proposition.

Although Id-in-Id is the first compiler to compile programs in parallel, I do not believe that its development is the main contribution of this research project. I believe instead that the techniques developed for graph reduction in this project have broader significance. For example, DeBruijn chains can be useful in any implementation of a functional language. It does not matter whether these chains are used in implementations for parallel or sequential machines or in parallel or sequential compilers. These techniques are the principal findings of this thesis. This chapter summarizes each of these techniques and suggests how each may contribute to future in various areas of software development.

DeBruijn Chains

The representation of free variables in lambda-calculus programs is a problem that all functional language compilers and run-time systems must face. DeBruijn chains are the Id-in-Id solution to this problem. They work well within the framework of Kid Graphs. DeBruijn chains, like other elements of Kid Graphs, do not need to be kept in centralized locations on a program graph. They also facilitate certain optimizations, like constant-propagation and code-hoisting.

This thesis does not really test the limits of DeBruijn chains, however. One reason for this is

practical. The goal of the current Id-in-Id compiler was only to perform as many optimizations as the current Id-in-Lisp compiler. There are some optimizations involving free-variables that the Id-in-Lisp compiler does not attempt, because it uses a super-combinator representation. Future versions of the Id-in-Id compiler should be able to attempt these other optimizations using DeBruijn chains.

The other reason has to do with cyclic graphs. This thesis has only discussed aspects of reducing acyclic Kid Graphs, but eventually the optimizer should be able to optimize Kid Graphs that contain cycles. Chapter 5 provided some discussion of cyclic graphs and asserted that perhaps the only solution for performing parallel reductions on such graphs is to detect cycles in a separate phase before reduction. This approach uses detection to essentially break cycles before they can cause any trouble. In order for this scheme to hold up theoretically, one must be able to prove that optimizations would never introduce new cycles into a graph or remove old ones. What does cycle-breaking have to do with DeBruijn chains? The most obvious way to introduce a cycle into a program graph is through a recursive function call. A function that calls itself references itself as a free variable and would therefore be represented as a cycle through a DeBruijn chain. If the cyclic graph problem is ever to be solved for Kid Graphs, it will definitely involve re-examining the DeBruijn chain algorithm for the case of cyclic graphs.

Disjoint-Set Unions

Disjoint-set unions were introduced as an alternative approach to mutating graphs during program reduction from the indirection-node approach. In the end, it turned out that the differences between the two were minor. Where the approaches differ the most is mostly in abstraction barriers.

I do think that the exercise of coming up with the disjoint-set unions for reduction was an illuminating one, though in the end the differences between them may have been small. Even when reduction is approached from the most abstract level, in the end the solution involves indirection-nodes or something equivalent to it. So one conclusion to draw may be that the indirection-node approach to reduction is not only a good one, but it is an essential one for implementing reductions. More importantly, though, the exercise sheds light on other work that functional language implementors can draw upon to implement reductions. Algorithms researchers have many techniques for implementing disjoint-set unions efficiently, and it would be interesting to see whether functional-language implementations could make use of them by carrying them over to indirection nodes. Furthermore, researchers would be able to make use of any future work that was done in disjoint-set unions.

Memobooks

The development of memobooks has opened up the door for the development of a parallel graph-reduction optimizer in Id. Without them, it would be possible to perform individual reductions, but impossible to coordinate them. Two things make memobooks worth noting. They are flexible enough to be used in a variety of applications. They also have a simple synchronization protocol which does not limit the parallelism of the programs that use them. Computer scientists have long noted the usefulness of memoization in programming and algorithms such as theorem provers, game-tree algorithms, and dynamic programming applications. The development of memobooks has not only opened up the door for the development of a parallel optimizer, but it has opened the door for the development of these other parallel applications too.

Bibliography

- [1] Zena M. Ariola and Arvind. P-TAC: A parallel intermediate language. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. ACM PRESS, September 1989.
- [2] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Computation structures group memo, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, October 1984.
- [3] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. In *Functional Programming Languages and Computer Architecture, Proceedings of the Conference held September 16-19, 1985 in Nancy, France*. Springer-Verlag Lecture Notes in Computer Science, Volume 201, 1985.
- [4] H. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, and M. Sleep. Term graph rewriting. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands*. Springer-Verlag Lecture Notes in Computer Science, Volume 259, June 1987.
- [5] Paul S. Barth. *Atomic Data-Structures for Parallel Computing*. PhD thesis, Massachusetts Institute of Technology, 1992.
- [6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.
- [7] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, December 1984.
- [8] N.G. De Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–92, 1972.
- [9] John Hughes. Why functional programming matters. *Computer Journal*, 32:98–107, 1989.
- [10] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.

- [11] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.
- [12] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report No. 117, INRIA, France, February 1990.
- [13] R. S. Nikhil. Id (version 90.0) reference manual. Computation Structures Group Memo 284, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, July 1990.
- [14] Rishiyur S. Nikhil and Arvind. An abstract description of a Monsoon-like ETS interpreter. Computation Structures Group Memo 308, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, June 1984.
- [15] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [16] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In *Logic and Computer Science*, pages 361–368. Academic Press, 1990.
- [17] S. L. Peyton Jones. FLIC—A Functional Language Intermediate Code. *ACM SIGPLAN Notices*, 23(8):30–48, August 1988.
- [18] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1993.
- [19] Jr. Philip John Koopman. *An Architecture for Combinator Graph Reduction*. Harcourt Brace Jovanovich, 1993.
- [20] P.S. Barth, R.S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict functional language with state. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag Lecture Notes in Computer Science, Volume 523, 1991.
- [21] M. Schonfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–16, 1924.
- [22] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in IF1. In *Functional Programming Languages and Computer Architecture, Proceedings of the Conference held September 16-19, 1985 in Nancy, France*. Springer-Verlag Lecture Notes in Computer Science, Volume 201, 1985.
- [23] Kenneth R. Traub. Compilation as partitioning: A new approach to compiling non-strict functional languages. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 75–88. ACM PRESS, September 1989.

- [24] David Turner. A new implementation technique for applicative languages. *Software-Practice and Experience*, 9:31-49, 1979.
- [25] C. Wadsworth. *Semantics and Pragmatics of The Lambda Calculus*. PhD thesis, University of Oxford, 1971.
- [26] Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):8-59, 1994.
- [27] Z. M. Ariola and Arvind. Compilation of Id. Computation Structures Group Memo 341, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, September 1991.

Appendix A

Memobook Implementation

```
%% Type definition for Mutable Lists

type M_List *0 *1 = M_Cons !(S_box *0 *1) !(M_List *0 *1) | M_Nil ;

%% Type definition for a Memo, or Sbox

type S_box *0 *1 =
    {record object = *0;
      full? = !B;
      value = .*1 }
    ;

%% A memobook is a hash table:
%% a hash table is an array of m-lists, each containing a memo

def create~memobook number_of_buckets =
    {1D_M_array (0,102) of
      [i] = M_nil || i <- 0 to 102 };
```



```

%% Memobook functions

defsubst read_memo table key =
  (lookup_insert_not_found
   table
   key).value ;

defsubst produce_memo fn table key =
  {
    box = lookup_insert_not_found table key;
    (full? = box!full?;
     ---
     box!full? = true)
  in
    if full? then box.value
    else
      { result = fn key ;
        box.value = result ;
        in result }
  } ;

%% Utility functions for manipulating the memobook table

defsubst lookup_insert_not_found ht_array key =
  { index = mod~int key 103;
    (bucket_head_ptr,member_result_list) =
      member?_and_insert~M_list key bucket ;
    (bucket = ht_array![index];
     ---
     ht_array![index] = bucket_head_ptr );
  in
    member_result_list!!m_cons_1
  };

defsubst member?_and_insert~M_List key mlist =
  if M_nil? mlist then
    { new_sbox =
      {record object = key;

```

```

        full? = false } ;
new_end_of_list =
    (M_cons new_sbox M_nil)
in
    (new_end_of_list,new_end_of_list) }
else
if (eq~int ((mlist!!M_cons_1).object) key) then
    (mlist,mlist)
else
    { (mlist',found_list) = member?_and_insert~M_list key mlist!M_cons_2;
      mlist!M_cons_2 = mlist'
    In
      (mlist,found_list) } ;

```

Appendix B

Code for Reduction Tests

```
%%% THE GRAPHS
%%% Datatype definition for the graphs.
```

```
type node =
  Reduce uid node node
| Reduced uid node node
| No_Reduce uid node node
| Leaf uid
| Reduce_leaf uid
| Reduced_leaf uid ;

type mnode =
  MReduce mnode_set mnode_set
| MReduced mnode_set mnode_set
| MNo_Reduce mnode_set mnode_set
| MLeaf
| MReduced_leaf
| MReduce_leaf ;

type mnode_set =
  {record
    set_id = I ;
    set_lock = ! I ;
    set_parent = !mnode_set ;
    set_type = !SET_DESCRIPTOR ;
```

```
        set_object = mnode ;  
    } ;  
  
set_IDs = (make_counter 0);  
  
type SET_DESCRIPTOR = Member |  
    Representative ;
```

```

%% Utilities for manipulating graphs

def Reduced? n1 n1' = (not (same_node n1 n1')) ;

def get_uid n1 =
  {case n1 of
    Change_uid _ _ = uid
  | No_change_uid _ _ = uid
  | Leaf uid = uid
  | Change_leaf uid = uid
  } ;

def same_node n1 n2 = (eq~int (get_uid n1) (get_uid n2)) ;

def get_muid b1 = (get_rep b1).set_id ;

def hash_mnode range node = (mod~int ((get_rep node).set_id) range) ;

def mk_node exp =
  {record
    set_id = (get_and_increment_counter set_IDs);
    set_lock = 0 ;
    set_object = exp ;
    set_type = Representative;
  } ;

```

```

%% DISJOINT-SET UNION code

type assignment_descriptor =
    EQUATE | ASSIGN | REVERSE_ASSIGN ;

defsubst arbitration_fn desc1 desc2 = ASSIGN ;

def get_rep cell =
  { lock = cell!set_lock ;
    ---
    ctype = cell!!set_type ;
  In
    (if (Representative? ctype) then
      { the_rep = cell ;
        cell!set_lock = lock ;
        in
          the_rep
        }
      else
        { parent = cell!set_parent ;
          new_parent = (get_rep parent) ;
          (cell!set_parent = new_parent ;
            ----
            cell!set_lock = lock ;) ;
          In
            new_parent
          })
        } ;

def unify cell1 cell2 =
  % Because make_assignment has the potential to mutate the
  % first cell, we immediately take a lock on the first cell
  { lock = cell1!set_lock ;
    ----
  In
    (if (same? cell1 cell2) then
      { cell1!set_lock = lock ;

```

```

In
    cell1
}
else
{ case cell1!!set_type,cell2!!set_type of
    % Both cells are representatives, make the assignment
    | Representative,Representative =

        % Use the arbitration function along with cell ids
        % to decide whether the order of the assignment
        % should be reversed
        (if (((Equate? (arbitration_fn cell1.set_object cell2.set_object))
            and
            (lt~int cell2.set_id cell1.set_id)) or
            (Reverse_assign? (arbitration_fn cell1.set_object
                cell2.set_object)))) then
            % Reverse the order of the assignment
            { cell1!set_lock = lock ;
              In
                (unify cell2 cell1 )
              }

            else
            % Make the assignment
            { cell1!set_parent = cell2 ;
              cell1!!set_type = Member ;
              ---
              cell1!set_lock = lock
              In
                cell2 })

            % In the following cases, we need to keep chasing
            % parent-pointers until we get to the representatives.
            | Representative,Member =
            { parent = cell2!!set_parent ;
              cell1!set_lock = lock ;
              in
                (unify cell1 parent)
            }

```

```

| Member, Representative =
  { parent = cell1!!set_parent ;
    cell1!set_lock = lock ;
  in
    (unify parent cell2)
  }
| Member, Member =
  { parent1 = cell1!!set_parent ;
    parent2 = cell2!!set_parent ;
    cell1!set_lock = lock ;
  in
    (unify parent1 parent2)
  }
}
)
};

```

```

def same_set cell1
  cell2 =
  { rep1 = get_rep cell1 ;
    rep2 = get_rep cell2;
  In (same? rep1 rep2)
  } ;

```



```
%% THE REDUCERS
```

```
%% SCHEME 1
```

```
def check_and_reduce memos n =  
  produce_memo (reduce1 memos) memos n ;
```

```
def reduce1 memos n =  
  {case n of  
    Reduce _ n1 n2 =  
      (Reduced (mk_uid ()) (check_and_reduce memos n1) (check_and_reduce memos n2))  
  | No_Reduce _ n1 n2 =  
      (No_Reduce (mk_uid ()) (check_and_reduce memos n1) (check_and_reduce memos n2))  
  | Leaf _ = Leaf (mk_uid ())  
  | Reduce_leaf _ = Reduced_leaf (mk_uid ())  
  } ;
```

```
def toplevel_reduce n =  
  { memos = create~memobook () ;  
    n' = check_and_reduce memos n ;  
  In  
    n' } ;
```

```
%% SCHEME 2
```

```
def check_and_spec memos n =  
  produce_memo (spec_reduce memos) memos n ;
```

```
def spec_reduce memos n =  
  {case n of  
    Reduce _ n1 n2 =  
      (Reduced (mk_uid ()) (check_and_spec memos n1) (check_and_spec memos n2))  
  | No_Reduce _ n1 n2 =  
      { n1' = (check_and_spec memos n1) ;  
        n2' = (check_and_spec memos n2) ;  
      In  
        if (Reduced? n1 n1') or (Reduced? n2 n2') then
```

```

        (No_Reduce (mk_uid ()) n1' n2')
      else n }
    | Leaf _ = n
    | Reduce_leaf _ = Reduced_leaf (mk_uid ())
  } ;

def toplevel_spec n =
  { memos = create~memobook () ;
    n' = check_and_spec memos n ;
    In
    n' } ;

%% SCHEME 3

def check_and_mreduce memos n =
  produce_mmemo (mreduce memos) memos n ;

def mreduce memos n =
  {case (get_rep n).set_object of
    MReduce n1 n2 =
      (mk_node (MReduced (check_and_mreduce memos n1) (check_and_mreduce memos n2)))
    | MNo_Reduce n1 n2 = n
    | MLeaf = n
    | MReduce_leaf = (mk_node MReduced_leaf)
  } ;

def toplevel_mreduce n =
  { memos = create~memobook () ;
    n' = check_and_mreduce memos n ;
    ---
    _ = {for i<- 0 to 30 do
      bucket = memos!![i] ;
      _ = {while (M_cons? bucket) do
        e = bucket!!M_cons_1 ;
        (next bucket) = bucket!!m_cons_2 ;
        o = e.object ;
        v = e.value ;
        _ = unify o v ;
      finally () } ;
  } ;

```

```
finally () } ;
```

```
-----
```

```
In n' } ;
```

```
%% TEST GRAPHS
```

```
def mreduce_leafnode = (mk_node Mreduce_leaf) ;
```

```
def mleafnode = (mk_node Mleaf) ;
```

```
def graph1 =
```

```
{ a1 = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (Leaf (mk_uid ()))) ;
```

```
  a2 = (no_reduce (mk_uid ()) (Leaf (mk_uid ()))
```

```
    (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (leaf (mk_uid ()))))) ;
```

```
  a3 = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) a5) ;
```

```
  a4 = (no_reduce (mk_uid ()) (Leaf (mk_uid ()))
```

```
    (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (leaf (mk_uid ()))))) ;
```

```
  a5 = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) a2) ;
```

```
In
```

```
(No_reduce (mk_uid ()))
```

```
(No_reduce (mk_uid ())) a1 a2)
```

```
(No_reduce (mk_uid ())) a3 a4))
```

```
} ;
```

```
def mgraph1 =
```

```
{ a1 = (mk_node (mno_reduce mleafnode mleafnode)) ;
```

```
  a2 = (mk_node (mno_reduce mleafnode
```

```
    (mk_node (mno_reduce mleafnode mleafnode)))) ;
```

```
  a3 = (mk_node (mno_reduce mleafnode a5)) ;
```

```
  a4 = (mk_node (mno_reduce mleafnode
```

```
    (mk_node (mno_reduce mleafnode mleafnode)))) ;
```

```
  a5 = (mk_node (mno_reduce mleafnode a2)) ;
```

```
In
```

```
(mk_node (mno_reduce
```

```
(mk_node (mno_reduce a1 a2))
```

```
(mk_node (mno_reduce a3 a4))))
```

```
} ;
```

```
def graph2 =
```

```
{ a1 = (Reduce (mk_uid ()) (Leaf (mk_uid ())) (Leaf (mk_uid ()))) ;
```

```

a2 = (No_reduce (mk_uid ()) (Leaf (mk_uid ()))
      (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (leaf (mk_uid ()))))) ;
a3 = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) a5) ;
a4 = (Reduce (mk_uid ()) (Leaf (mk_uid ()))
      (no_reduce (mk_uid ()) (Reduce_leaf (mk_uid ())) (Reduce_leaf (mk_uid ()))))) ;
a5 = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) a2) ;
In
  (No_reduce (mk_uid ())
    (No_reduce (mk_uid ()) a1 a2)
    (No_reduce (mk_uid ()) a3 a4))
} ;

def mgraph2 =
{ a1 = (mk_node (mReduce mleafnode mleafnode)) ;
  a2 = (mk_node (mno_reduce mleafnode
    (mk_node (mno_reduce mleafnode mleafnode)))) ;
  a3 = (mk_node (mno_reduce mreduce_leafnode a5)) ;
  a4 = (mk_node (mReduce mleafnode
    (mk_node (mno_reduce mreduce_leafnode mreduce_leafnode)))) ;
  a5 = (mk_node (mno_reduce mleafnode a2)) ;
In
  (mk_node (mno_reduce
    (mk_node (mno_reduce a1 a2))
    (mk_node (mno_reduce a3 a4))))
} ;

def graph3 =
{ a1 = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (Reduce_Leaf (mk_uid ()))) ;
  a2 = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ()))
    (no_reduce (mk_uid ()) (reduce_Leaf (mk_uid ())) (Reduce_leaf (mk_uid ()))))) ;
  a3 = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) a5) ;
  a4 = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ()))
    (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (Reduce_leaf (mk_uid ()))))) ;
  a5 = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) a2) ;
In
  (No_reduce (mk_uid ())
    (No_reduce (mk_uid ()) a1 a2)
    (No_reduce (mk_uid ()) a3 a4))
} ;

```

```

def mgraph3 =
  { a1 = (mk_node (mno_reduce mreduce_leafnode mreduce_leafnode)) ;
    a2 = (mk_node (mno_reduce mreduce_leafnode
      (mk_node (mno_reduce mreduce_leafnode mreduce_leafnode)))) ;
    a3 = (mk_node (mno_reduce mreduce_leafnode a5)) ;
    a4 = (mk_node (mno_reduce mreduce_leafnode
      (mk_node (mno_reduce mreduce_leafnode mreduce_leafnode)))) ;
    a5 = (mk_node (mno_reduce mreduce_leafnode a2)) ;
  } ;

In
  (mk_node (mno_reduce
    (mk_node (mno_reduce a1 a2))
    (mk_node (mno_reduce a3 a4))))
} ;

def graph4 =
  { a1 = (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (Reduce_Leaf (mk_uid ()))) ;
    a2 = (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ()))
      (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (reduce_leaf (mk_uid ()))))) ;
    a3 = (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) a5) ;
    a4 = (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ()))
      (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (reduce_leaf (mk_uid ()))))) ;
    a5 = (reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) a2) ;
  } ;

In
  (reduce (mk_uid ())
    (reduce (mk_uid ()) a1 a2)
    (reduce (mk_uid ()) a3 a4))
} ;

def mgraph4 =
  { a1 = (mk_node (mreduce mreduce_leafnode mreduce_leafnode)) ;
    a2 = (mk_node (mreduce mreduce_leafnode
      (mk_node (mreduce mreduce_leafnode mreduce_leafnode)))) ;
    a3 = (mk_node (mreduce mreduce_leafnode a5)) ;
    a4 = (mk_node (mreduce mreduce_leafnode
      (mk_node (mreduce mreduce_leafnode mreduce_leafnode)))) ;
    a5 = (mk_node (mreduce mreduce_leafnode a2)) ;
  } ;

In
  (mk_node (mreduce

```

```

    (mk_node (mreduce a1 a2))
    (mk_node (mreduce a3 a4))))
} ;

def graph5 =
{ root = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (Leaf (mk_uid ()))) ;
  in
  {for i <- 1 to 19 do
    next root = (No_reduce (mk_uid ()) (Leaf (mk_uid ())) root) ;
    finally root }
} ;

def mgraph5 =
{ root = (mk_node (Mno_reduce mleafnode mleafnode)) ;
  in
  {for i <- 1 to 19 do
    next root = (mk_node (MNo_reduce mleafnode root)) ;
    finally root }
} ;

def graph6 =
{ root = (no_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (Reduce_Leaf (mk_uid ()))) ;
  in
  {for i <- 1 to 19 do
    next root = (No_reduce (mk_uid ()) (Leaf (mk_uid ())) root) ;
    finally root }
} ;

def mgraph6 =
{ root = (mk_node (Mno_reduce mreduce_leafnode mreduce_leafnode)) ;
  in
  {for i <- 1 to 19 do
    next root = (mk_node (MNo_reduce MLeafnode root)) ;
    finally root }
} ;

def graph7 =
{ root = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (Leaf (mk_uid ()))) ;
  root2 =

```

```

    {for i <- 1 to 14 do
      next root = (No_reduce (mk_uid ()) (Leaf (mk_uid ())) root) ;
      finally root } ;
  In
    {for i <- 1 to 5 do
      next root2 = (No_reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) root2) ;
      finally root2 }
} ;

def mgraph7 =
  { root = (mk_node (Mno_reduce mleafnode mleafnode)) ;
    root2 = {for i <- 1 to 14 do
      next root = (mk_node (MNo_reduce mleafnode root)) ;
      finally root }
  in
    {for i <- 1 to 5 do
      next root2 = (mk_node (MNo_reduce mreduce_leafnode root2)) ;
      finally root2 }
} ;

def graph8 =
  { root = (no_reduce (mk_uid ()) (Leaf (mk_uid ())) (Leaf (mk_uid ()))) ;
    root2 =
      {for i <- 1 to 9 do
        next root = (No_reduce (mk_uid ()) (Leaf (mk_uid ())) root) ;
        finally root } ;
    root3 =
      {for i <- 1 to 5 do
        next root2 = (Reduce (mk_uid ()) (Reduce_leaf (mk_uid ())) root2) ;
        finally root2 } ;
  In
    {for i <- 1 to 5 do
      next root3 = (No_reduce (mk_uid ()) (Leaf (mk_uid ())) root3) ;
      finally root3 }
} ;

def mgraph8 =
  { root = (mk_node (Mno_reduce mleafnode mleafnode)) ;
    root2 = {for i <- 1 to 14 do

```



```

        next root = (mk_node (MNo_reduce mleafnode root)) ;
    finally root } ;
root3 = {for i <- 1 to 5 do
    next root2 = (mk_node (MReduce mreduce_leafnode root2)) ;
    finally root2 }
in
    {for i <- 1 to 5 do
        next root3 = (mk_node (MNo_reduce mleafnode root3)) ;
        finally root3 }
} ;

def graph9 =
{ root = (Reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (Reduce_Leaf (mk_uid ()))) ;
  root2 =
    {for i <- 1 to 9 do
        next root = (Reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) root) ;
        finally root } ;
  In
    {for i <- 1 to 10 do
        next root2 = (No_reduce (mk_uid ()) (Leaf (mk_uid ())) root2) ;
        finally root2 }
} ;

def mgraph9 =
{ root = (mk_node (MReduce mreduce_leafnode mreduce_leafnode)) ;
  root2 = {for i <- 1 to 9 do
    next root = (mk_node (MReduce mreduce_leafnode root)) ;
    finally root }
  In
    {for i <- 1 to 10 do
        next root2 = (mk_node (MNo_Reduce MLeafnode root2)) ;
        finally root2 }
} ;

def graph10 =
{ root = (Reduce (mk_uid ()) (Reduce_Leaf (mk_uid ())) (Reduce_Leaf (mk_uid ()))) ;
  in
    {for i <- 1 to 19 do
        next root = (Reduce (mk_uid ()) (Reduce_leaf (mk_uid ())) root) ;

```

```
        finally root }
    } ;

def mgraph10 =
{ root = (mk_node (MReduce mreduce_leafnode mreduce_leafnode)) ;
  in
  {for i <- 1 to 19 do
    next root = (mk_node (MReduce mreduce_leafnode root)) ;
    finally root }
} ;
```