LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-417

SEQUENTIAL IMPLEMENTATION OF LENIENT PROGRAMMING LANGUAGES

Kenneth R. Traub

October 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA200984

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TR-417

5. MONITORING ORGANIZATION REPORT NUMBER(S)
N00014-84-K-0099

6a. NAME OF PERFORMING ORGANIZATION
MIT Laboratory for Computer Science

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research/Department of Navy

6c. ADDRESS *(City, State, and ZIP Code)*
545 Technology Square
Cambridge, MA 02139

7b. ADDRESS *(City, State, and ZIP Code)*
Information Systems Program
Arlington, VA 22217

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA/DOD

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
1400 Wilson Blvd.
Arlington, VA 22217

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Sequential Implementation of Lenient Programming Languages

12. PERSONAL AUTHOR(S)
Traub, Kenneth R.

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
October 1988

15. PAGE COUNT
200

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Compilers, Dependence Graphs, Functional Languages, Functional Quads, Lazy Evaluation, Lenient Evaluation, Multi-Threaded, Non-Strictness, Partitioning of Programs

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

In non-strict functional languages, a data structure may be read before all its components are written, and a function may return a value before finishing all its computation or even before all its arguments have been evaluated. Such flexibility gives expressive power to the programmer, but makes life difficult for the compiler because it may not be possible to totally order instructions at compile time; the correct order can vary dramatically with the input data. Presently, compilers for non-strict languages rely on lazy evaluation, in which a subexpression is not evaluated until known (at run time) to contribute to the final answer. By scheduling each subexpression separately, lazy evaluation automatically deals with the varying orderings required by non-strictness, but at the same time incurs a great deal of overhead. Recent research has employed strictness analysis and/or annotations to make more scheduling decisions at compile time, and thereby reduce the overhead, but because these techniques seek to retain laziness, they are limited in effectiveness.

(continued on back)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Judy Little, Publications Coordinator

22b. TELEPHONE *(Include Area Code)*
(617) 253-5894

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆U.S. Government Printing Office: 1985—507-047

Unclassified

19. We present an alternative compilation strategy which deals with non-strictness independent of laziness, through the analysis of data dependence. Our analysis determines which instructions can be ordered at compile time and which must be scheduled at run time in order to implement non-strictness properly. We then have the option of imposing laziness or ignoring it--and we find that choosing the latter path can lead to significantly reduced overhead. Abandoning laziness means certain programs (those which use "infinite objects") may fail to terminate properly. We suspect that non-strictness and not the ability to handle infinite objects is the more important feature for the programmer; nevertheless, we can provide some annotations to the programmer to achieve termination in the presence of infinite objects. Even with annotations, our approach entails less overhead.

We discuss our strategy in the context of both sequential implementations and parallel implementations where the object code is partially sequentialized. We also show how lazy code can be generated from our framework.

Sequential Implementation of Lenient Programming Languages

Kenneth R. Traub

MIT / LCS / TR-417
September 1988

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099. The author has been supported in part by a National Science Foundation Fellowship and an A. T. & T. PhD Scholarship.

This report was originally published as the author's doctoral dissertation. In addition to typographical corrections throughout, Chapter 8 has been completely revised. In case of discrepancy between the thesis and this report, the text of this report should be preferred.

Sequential Implementation of Lenient Programming Languages

Kenneth R. Traub

Technical Report MIT / LCS / TR-417
September 1988

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

In *non-strict* functional languages, a data structure may be read before all its components are written, and a function may return a value before finishing all its computation or even before all its arguments have been evaluated. Such flexibility gives expressive power to the programmer, but makes life difficult for the compiler because it may not be possible to totally order instructions at compile time; the correct order can vary dramatically with the input data. Presently, compilers for non-strict languages rely on *lazy evaluation*, in which a subexpression is not evaluated until known (at run time) to contribute to the final answer. By scheduling each subexpression separately, lazy evaluation automatically deals with the varying orderings required by non-strictness, but at the same time incurs a great deal of overhead. Recent research has employed strictness analysis and/or annotations to make more scheduling decisions at compile time, and thereby reduce the overhead, but because these techniques seek to retain laziness, they are limited in effectiveness.

We present an alternative compilation strategy which deals with non-strictness independent of laziness, through the analysis of data dependence. Our analysis determines which instructions can be ordered at compile time and which must be scheduled at run time in order to implement non-strictness properly. We then have the option of imposing laziness or ignoring it—and we find that choosing the latter path can lead to significantly reduced overhead. Abandoning laziness means certain programs (those which use "infinite objects") may fail to terminate properly. We suspect that non-strictness and not the ability to handle infinite objects is the more important feature for the programmer; nevertheless, we can provide some annotations to the programmer to achieve termination in the presence of infinite objects. Even with annotations, our approach entails less overhead.

The author discusses his
~~We discuss our~~ strategy in the context of both sequential implementations and parallel implementations where the object code is partially sequentialized. We also show how lazy code can be generated from our framework. (KR)

Key Words and Phrases: Compilers, Dependence Graphs, Functional Languages, Functional Quads, Lazy Evaluation, Lenient Evaluation, Multi-Threaded, Non-Strictness, Partitioning of Programs.

Acknowledgments

This thesis is the culmination of four years spent at the MIT Laboratory for Computer Science. It is uniquely a product of that working environment, and I am grateful for the support and friendship of my colleagues there.

I wish especially to thank Arvind, my thesis advisor, who first suggested the topic to me as a potential master's thesis in 1985. His intuition and conviction about the importance of this work led to its "promotion" to the doctoral level, and he has continually cajoled me to view the work in an ever-broader context and approach it from more and more relevant points of view. Without his influence, this work would not have been nearly as significant a contribution to the field as I feel it is now. I also am grateful to him for helping me to finish on schedule.

I also wish to thank my readers, Rishiyur S. Nikhil and Stephen Ward, for their time and patience in reading drafts of my work. Nikhil provided many thoughtful technical comments with an occasional welcome interjection of his inimitable wit, while Steve's perspective was a great help in overcoming some of the inbred preconceptions of my research group.

Several other people have helped in the preparation of this thesis. David Culler, Steve Heller, and Natalie Tarbet were very kind in offering to proofread various drafts. I had many enjoyable discussions with Bob Iannucci about the material as it was being developed. It is a pleasure to acknowledge my officemate of three years, Vinod Kathail, who was always willing to educate me about lambda calculus and related mathematical methods; his influence is clearly visible in Chapters 4 and 5. Serge Plotkin provided the construction used in the proof of Section 5.7.

I have felt privileged in my four years here to work with the members of the Computation Structures Group, without a doubt the most talented group of individuals I have ever encountered. In addition to those already mentioned, I wish to thank Greg Papadopoulos, Richard Soley and Suresh Jagannathan for their interest in and enthusiasm for my work. Steve Heller, Andrew Chien, and Kalyn Culler provided much-needed moral support.

Finally, I wish to thank my parents, the value of whose love and support is beyond measure.

The author has been supported in part by a National Science Foundation Fellowship and an A. T. & T. PhD Scholarship.

To my parents,
Pat and Carol Traub

Contents

Chapter 1

Introduction

In *non-strict* functional languages, a data structure may be read before all its components are written, and a function may return a value before finishing all its computation or even before all its arguments have been evaluated. Such flexibility gives expressive power to the programmer, but makes life difficult for the compiler because it may not be possible to totally order instructions at compile time; the correct order can vary dramatically with the input data. Presently, compilers for non-strict languages rely on *lazy evaluation*, in which a subexpression is not evaluated until known (at run time) to contribute to the final answer. By scheduling each subexpression separately, lazy evaluation automatically deals with the varying orderings required by non-strictness, but at the same time incurs a great deal of overhead. Recent research has employed strictness analysis and/or annotations to make more scheduling decisions at compile time, and thereby reduce the overhead, but because these techniques seek to retain laziness, they are limited in effectiveness.

We present an alternative compilation strategy which deals with non-strictness independent of laziness, through the analysis of data dependence. Our analysis determines which instructions can be ordered at compile time and which must be scheduled at run time in order to implement non-strictness properly. We then have the option of imposing laziness or ignoring it—and we find that choosing the latter path can lead to significantly reduced overhead. Abandoning laziness means certain programs (those which use "infinite objects") may fail to terminate properly. We suspect that non-strictness and not the ability to handle infinite objects is the more important feature for the programmer; nevertheless, we can provide some annotations to the programmer to achieve termination in the presence of infinite objects. Even with annotations, our approach entails less overhead.

We discuss our strategy in the context of both sequential implementations and parallel implementations where the object code is partially sequentialized. We also show how lazy code can be generated from our framework.

The guiding principle behind the compilation method presented in this thesis is simply that compilation is a process of choosing the order in which subexpressions will be evaluated. Unlike imperative programs, functional programs do not give any explicit indication of subexpression ordering; this is why they are termed "declarative". But sequential code is by definition ordered, so if sequential code is to be produced from functional programs, the compiler must take ordering decisions. A compiler must perform considerable analysis to make these decisions correctly, because this ordering information is not explicit in the source code—this sort of analysis is the heart of any compiler for a non-strict functional language. Existing functional language compilers perform this analysis indirectly, by attempting to emulate an evaluator for the language which executes subexpressions in proper sequence. In contrast we attack the ordering issue head on, and consequently we are able to achieve a much cleaner separation between describing the behavior object code must exhibit to faithfully implement the semantics and developing techniques to achieve that behavior on the target architecture. We find that non-strict object code can be described abstractly as a set of sequential threads, each internally ordered but whose relative order with respect to other threads is determined at run time. We then formulate the conversion of a source program into these sequential threads as first determining the constraints on thread construction imposed by the language semantics, and then partitioning the original program into threads based on these constraints. From there, the abstract threads may be converted into concrete object code for a particular target architecture, given the execution mechanisms it provides.

In presenting the conversion from source program to sequential threads we have tried to balance the desire for mathematical rigor against the need for practical techniques. We actually present two parallel developments of the method. The first consists of a formal operational semantics of functional languages called *functional quads* and a formal theory of *requirement* which characterizes the ordering relationships that must hold for a program to satisfy the semantics. In this mathematical world we are able to capture the essential constraints that object code must satisfy, independent of any particular analysis technique a compiler might employ to infer them. The second development presents a complete example of techniques that could form the basis of a real compiler, in which ordering relationships are inferred through the analysis of *data dependence*. Each stage of the practical method is proved safe relative to the standard provided by the theoretical model, through a well-defined notion of approximation. Requirement theory therefore provides strong assurances about the correctness of actual compilation methods, and is also a way of formally characterizing the concept of data dependence.

The remainder of this chapter describes some notational conventions used in this thesis. Chapter 2 sets the stage by reviewing the current state of the art in functional language compiling, including strict and lazy evaluation. Chapter 3 illustrates the kind of code generated by conventional techniques and by our techniques, comparing the overhead. In the next two chapters we lay the theoretical foundations of our method: we present the functional quads operational semantics of non-strict programs in Chapter 4 and develop the theory of requirement and compiling into sequential threads in Chapter 5. The parallel to the theoretical development in Chapter 5 is found in the next two chapters; in Chapter 6 we discuss the analysis of data dependence and in Chapter 7 we describe algorithms for using data dependence information to partition a program into sequential threads. Chapter 8 completes the description of a practical compiler by considering the conversion of sequential threads into actual object code for a variety of implementations, both sequential and parallel, along with optimization. Chapter 9 concludes.

1.1 Notation: Functional Programs

Modern functional programming languages have a number of advanced features, including pattern matching, powerful iteration facilities, algebraic and abstract data types, and type inference systems. Functional programming examples in this thesis will be presented in a minimal kernel language, which we describe here. This language is sort of the "least common denominator" of such common non-strict functional languages as Miranda [74], LML [10], and the forthcoming Haskell [76]; it includes numbers, simple data structures, arithmetic expressions, conditional expressions, curried function application, and lexical scoping through "letrec" blocks. All of the advanced features mentioned above can be expressed in the kernel language through source-to-source transformations. The issues related to type systems, too, are absent in the kernel language, for we can assume that type-checking is done before conversion to the kernel language. The concrete syntax of the kernel language is patterned after Id Nouveau [53], but this is not meant to imply that Id syntax has any particular advantages over the syntax of other languages. The kernel used here simply has an uncluttered, easy-to-read appearance in Id syntax.

The grammar of our language is given below.

```
Expression   ::=  Number | Boolean | Identifier | Expression Op Expression |
                  if Expression then Expression else Expression |
                  Expression Expression | Block | (Expression)
Op           ::=  + | - | * | / | == | < | ...
Block        ::=  { Binding ; Binding ; ... in Expression }
Binding      ::=  Identifier = Expression | Identifier Identifier ... = Expression
```

A brief description of these features:

Application Function application is written by juxtaposition, so if `f` is a function of three arguments, and `g` a function of two, a legal expression is:

```
f 5 (g x 3) y
```

where the second argument of `f` is the result of applying `g` to `x` and 3. Functions in Id are *curried*, so that a partial application like (g 5) is simply a function of one argument which calls g on 5 and that argument. Application, therefore, associates to the left.

Infix Expressions To improve the readability of arithmetic expressions, infix notation is provided as a syntactic sugar for the application of arithmetic, relational, and logical primitives. Thus, we have

```
x + y   ≡   (+) x y
```

where (+) represents the primitive addition function. The usual precedence and associativity rules apply to the infix operators, with application having highest precedence.

Conditionals An expression of the form `if` E_1 `then` E_2 `else` E_3 evaluates to the value of E_2 or E_3, depending on whether E_1 evaluates to `true` or `false`, respectively.

Blocks Blocks allow names to be given to expressions; each binding introduces a new name whose scope is the entire block. This "letrec" scoping rule allows bindings to be recursive or mutually recursive. A binding either gives a name to the value of an expression, as in:

```
{...
 x = y * 5;
 ...}
```

or defines a new function, as in:

```
{...
 dist x y = sqrt(x * x + y * y);
 ...}
```

In a function definition binding, the scope of the function name (the leftmost identifier) is the whole block, while the scope of the formals (the other identifiers on the left hand side) is only the body of the function (the right hand side). The number of formals is the *arity* of the function. Functions of zero arity are not supported.

The value of a block expression is the value of the expression following the `in` keyword.

Primitives We assume a basic repertoire of primitive functions for manipulating the primitive data types. We have already mentioned the arithmetic, relational, and logical operators, for which infix syntax is provided. We also assume primitives for data structures. In general we provide tagged n-tuples, with associated constructor functions, selector functions, and tag predicates. Specifically, for each n-ary type t we provide:

$$\begin{array}{rcl}
\texttt{make\_t}\; v_1 \ldots v_n & \longrightarrow & \langle t, v_1, \ldots, v_n \rangle \\
\texttt{sel\_t\_i}\; \langle t, v_1, \ldots, v_i, \ldots, v_n \rangle & \longrightarrow & v_i \\
\texttt{is\_t?}\; v & \longrightarrow & \begin{cases} \texttt{true} & \text{if } v = \langle t, v_1, \ldots, v_n \rangle \\ \texttt{false} & \text{otherwise} \end{cases}
\end{array}$$

0-ary types are supported; they have no selectors, and instead of a constructor function there is just a constant.

For convenience, we will often make use of the 2-ary "cons" type, with constructor `cons`, selectors `hd` and `tl`, and predicate `cons?`, along with the 0-ary `nil` and its predicate `nil?`.

1.2 Notation: Sets, Relations, and Graphs

Given a set A, a binary relation $\vdash$ on A is a subset of $A \times A$; we write $a_0 \vdash a_1$ if $(a_0, a_1) \in \vdash$. We also write $a_0 \vdash^0 a_0$, and $a_0 \vdash^{i+1} a_{i+1}$ if $a_0 \vdash^i a_i$ and $a_i \vdash a_{i+1}$. We write $a_0 \vdash^\epsilon a_1$ if $a_0 \vdash^0 a_1$

or $a_0 \vdash a_1$, $a_0 \vdash^+ a_1$ if $a_0 \vdash^i a_1$ for some $i > 0$, and $a_0 \vdash^* a_1$ if $a_0 \vdash^i a_1$ for some $i \geq 0$; $\vdash^\epsilon$, $\vdash^+$, and $\vdash^*$ are called the *reflexive closure*, the *transitive closure*, and the *reflexive transitive closure* of $\vdash$, respectively. A relation is reflexive (transitive) if it is equal to its own reflexive (transitive) closure. Given a subset $A' \subseteq A$, the *restriction* of a relation to this subset, notation $(\vdash \mid A')$, is the set $\vdash \cap (A' \times A')$.

A directed graph $\vec{E}$ is an ordered pair (V, E) where $E \subseteq V \times V$; V is the set of *vertices*, and E the set of *edges*. Because we will have many graphs with the same vertex set, we use a non-standard convention whereby the graph is represented by the edge set symbol with an arrow on top. Since the edge set is a binary relation on the vertices, all of the terminology in the preceding paragraph applies. We use the notation $u \xrightarrow{E} v$ to indicate that $(u, v) \in E$, and so $u \xrightarrow{E}{}^i v$, $u \xrightarrow{E}{}^\epsilon v$, $u \xrightarrow{E}{}^+ v$, and $u \xrightarrow{E}{}^* v$ indicate that between u and v there is a path of length i, a path of length zero or one, a path of length at least one, and a path of length at least zero, respectively. A graph is *cyclic* if there exists $u \in V$ such that $u \xrightarrow{E}{}^+ u$, and *acyclic* otherwise. If $V' \subseteq V$, then $(V', (E \mid V'))$ is the *induced subgraph* of $\vec{E}$ on V'.

An undirected graph is a graph such that for any $u, v \in V$, if $(u, v) \in E$, then $(v, u) \in E$ also. We use the slightly informal notation $\{u, v\} \in E$ to indicate that $(u, v), (v, u) \in E$ for an undirected graph.

Chapter 2

Background—Functional Language Compilers

To produce sequential object code, whether for a von Neumann architecture or a parallel architecture which executes sequential code segments in parallel, many decisions about the relative order of subexpression evaluation must be taken at compile time. In an imperative programming language such as Fortran, Pascal, C, *etc.*, all of these decisions are made by the programmer and communicated to the compiler through the textual ordering of the program. Imperative compilers, therefore, need only worry about ordering if they wish to change the programmer's ordering, say to perform code *motion* optimizations. In declarative programming languages, however, the programmer makes no assertions about the ordering of subexpressions; instead, the programmer describes how the results of one computation are used by another. It is up to the compiler to choose an order which satisfies the data dependences specified by the programmer.

For functional languages, the standard practice is to define an interpreter, and generate compiled code for a program which mimics the order of evaluation steps performed by the interpreter on that program. There are two standard functional language interpreters, each having a simple rule for deciding what expression to evaluate next:

The Strict Interpreter To evaluate (f e_1 ... e_n), where f has arity n, first evaluate each of the argument expressions e_1 through e_n, and then evaluate the body of f, passing the evaluated arguments.

The Lazy Interpreter To evaluate (f e_1 ... e_n), where f has aiity n, evaluate the body of f, passing the arguments unevaluated.

Data constructors are treated like procedure calls, with the strict interpreter evaluating the component expressions before building the structure, and the lazy interpreter leaving the components unevaluated. The rules for letrec blocks are analogous: the strict interpreter evaluates the right-hand sides of the bindings before binding the variables and evaluating the final expression, while the lazy interpreter evaluates the final expression with the right-hand sides of the bindings unevaluated (in both cases this is a little trickier than it sounds because the bindings can refer to one another; we defer the details until the next chapter). Strict and lazy evaluation are identical for the remaining language constructs: constants, identifiers, conditionals, and arithmetic. It should be noted, however, that in lazy evaluation an identifier might be bound to an unevaluated expression. Simply referring to such an identifier does not evaluate the expression, but if the argument to an arithmetic operation or the predicate of a conditional is unevaluated, it must be evaluated before further progress can be made.[1] The net effect in lazy evaluation is that an expression is not evaluated before it is known to contribute to the final answer. In place of *strict* and *lazy* are sometimes found the terms *applicative-order* and *normal-order*, or *call-by-value* and *call-by-need*.[2]

We illustrate the difference between strict and lazy evaluation with a small example:

```
examp a b =
  {f x y z = if x < 0 then y * y else z * z;
   in
     f a (b+4) (b-17)};
```

Suppose we call examp with arguments of 10 and 20. Under strict evaluation, we have a call to internal procedure f, so we first have to evaluate its arguments, yielding 10, 24, and 3. We then call f, which compares 10 to 0, selects the "else" clause, multiplies 3 times 3, and returns 9 as the answer. Notice that we performed the addition (b+4) even though its value was ultimately ignored. Under lazy evaluation, neither (b+4) nor (b-17) would be evaluated before calling f. But once inside f, when the "else" clause has been selected, we need the value of z, so we then perform the subtraction (b-17). The answer is the same, but we have avoided the addition (b+4).

In some sense, the difference between strict and lazy evaluation is one of efficiency, as lazy evaluation seeks to avoid the computation of useless intermediate results. But in fact, lazy

[1] In some sense, it is an implementation choice *how much* an argument to a primitive need be evaluated. For example, to evaluate (x > 0) we need only evaluate x enough to determine its sign. For pragmatic reasons, of course, expressions denoting scalar types are generally evaluated completely when any information about their values is needed. On the other hand, a data structure selector function will evaluate its argument only enough to get the desired component.

[2] A note on usage: *strict* and *lazy* are most often used when distinguishing between evaluation rules applied consistently in an implementation. On the other hand, *call-by-value* and *call-by-need* tend to be used when both mechanisms are available in the same implementation, at the discretion of the programmer [17]. *Applicative-order* and *normal-order* are most correct when applied to lambda calculus, where they are also called *leftmost-innermost* and *leftmost* (or *leftmost-outermost*).

evaluation gives more expressive power to the programmer than does strict evaluation, because there are programs which produce answers under lazy evaluation but not under strict evaluation. The most obvious subclass of these programs are ones in which the useless intermediate results avoided by lazy evaluation require an infinite amount of computation under strict evaluation. An example:

```
{ints_from n = cons n (ints_from (n+1));
 in
   nth 10 (ints_from 1)}
```

((nth i l) returns the *i*th element of list *l*.) Under strict evaluation, the call to ints_from results in an infinite recursion, but under lazy evaluation the calls to ints_from for *n* larger than 10 are not made, since they do not contribute to the final result. The utility of programs like this which manipulate "infinite lists" was one of the earliest reasons for interest in lazy evaluation [25]. We will have a lot more to say about the relative expressive power of various evaluation methods in Chapter 3.

The above exposition of strict and lazy evaluation was intentionally informal, to avoid burdening the reader with a lot of detail and notation. The reader can rest assured that we will define them very precisely in Chapter 4; it may be worth rereading this chapter and the next after the formal definition has been presented.

In the remainder of this chapter we focus only on how today's compilers produce object code which mimics strict or lazy evaluation. Strict compilers draw mainly on Lisp compiler technology, the most important aspects coming from Steele's pioneering Rabbit compiler [66] and the later Orbit compiler [50] from Yale. The lazy approach has three substrands: force-and-delay style compilers [29, 34], combinator and supercombinator graph reduction [72, 38], and abstract machine based compilers [41, 43]. We discuss all of these techniques below. We are mainly concerned with strategy in this chapter; in the next we examine in detail the object code produced from these compilers.

2.1 Strict Compilers

Most imperative programming languages have strict procedure calls, so strict functional compilers have a lot in common with imperative compilers. But in functional languages, procedures are first-class objects and may be created and manipulated more freely than in most imperative

```
example_a x y =
  {f a = a * x;
   in
     f (y-5)+
     f (y+8) };
```

Arguments, results, and environment in registers.

(a)

```
example_b x l =
  {f a = a * x;
   in
     map f l};
```

Downward-only procedure: environment on stack.

(b)

```
example_c x y =
  {f a = (a+x)/y;
   g b = x/(b-y);
   in
     cons f g};
```

Procedures share environment for x and y on heap.

(c)

Figure 2.1: Procedure Linkage Optimization by Orbit

languages. Not surprisingly, then, strict functional compilers are based on compilers developed for Lisp and its variants, as procedures are also first-class in Lisp. Lisp-like languages have a well-defined functional subset, and so Lisp compiler technology carries over directly into the functional regime. The first compiler to concentrate on efficient implementation of first-class procedures was the Rabbit compiler [66], developed for the Scheme dialect of Lisp [59]. Rabbit technology was subsequently refined and put into operation by the Orbit compiler [50].

The two main innovations in these compilers were the use of the continuation passing style (CPS) transformation [67, 65] and efficient procedure construction and invocation. The CPS transformation is used to express a variety of complex imperative control constructs in terms of procedure calling; since these imperative constructs are absent in functional languages, CPS is not of great interest here. In a functional context, all CPS accomplishes is fixing the relative ordering in which arguments to the same procedure are evaluated.

Efficient handling of procedures, on the other hand, is critically important to functional compilation, because first-class procedures are heavily used in the functional programming style. The overhead associated with procedures and procedure calls stems from parameter (and result) passing and from the use of free variables bound in enclosing lexical scopes (the "environment"). The environment problem has two sides: the values of free variables must be recorded at the time the procedure is created, and they must be accessed when the procedure is called. The main insight in Rabbit/Orbit is that there need not be a uniform convention for parameter passing, nor for environment representation; instead, these can be customized for efficiency on a procedure by procedure basis. Among the customizations:

- If all callers of a procedure can be identified and are in the same activation as where the procedure is created, then parameters, results, and the environment can be passed in registers rather than on the stack. (Figure 2.1a)

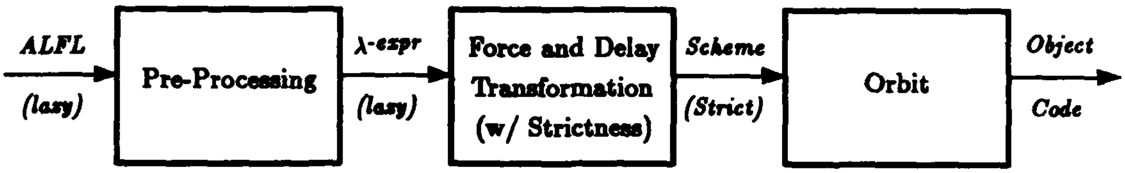


Figure 2.2: Force-and-Delay Lazy Compilation: the ALFL Compiler

- If a procedure can only be accessed by its creating activation and that activation's descendants (*i.e.*, if the procedure does not "escape upward"), then the procedure can be compiled to fetch its environment variables from the stack, as in Algol. This saves the overhead of constructing the environment, since the stack is already there. (Figure 2.1b)
- If a procedure escapes upward, then the environment must be allocated on the heap. If several procedures are created in the same activation, however, it may be possible to share their environments or a portion thereof, or share them with their parent. (Figure 2.1c)

With these and other procedure-calling optimizations, Orbit is able to achieve superior performance compared with other compilers, especially for programs which make heavy use of procedures [50].

Lisp and strict functional language compilers also face the problems of register allocation, optimization, *etc.*, faced by compilers for languages such as Fortran, C, and Pascal. We will not delve into these issues, as they are adequately described elsewhere [3], and are not significantly different in the functional context.

2.2 Lazy Compilers: Force-and-Delay

As we discussed earlier, lazy evaluation requires that argument expressions be passed to procedures unevaluated, and that primitives like arithmetic must cause such unevaluated expressions to be evaluated. By introducing unevaluated expressions as first-class objects into the language, Henderson [29] developed a source-to-source transformation which makes explicit where expressions are delayed and where the delayed expressions are to be evaluated. The transformed code can be executed using strict evaluation, but will behave as if executed under lazy evaluation. The ALFL compiler [34] uses this transformation to convert programs written in a lazy functional language, ALFL, into Scheme, which is then compiled into object code by Orbit (see Figure 2.2).[3]

[3] In [34], the ALFL compiler is described as a combinator-based compiler. In fact, it only uses combinators as an intermediate form for doing optimizations like common subexpression elimination, constant folding, and

Basic Transformation Rules

e_1 e_2	→	(force e_1) (delay e_2)	
if e_1 then e_2 else e_3	→	if (force e_1) then e_2 else e_3	
{x1 = e_1;		{x1 = (delay e_1);	
...		...	
xn = e_n;	→	xn = (delay e_n);	
in		in	
e_{in}}		e_{in}}	

Implementation of Primitive Functions

cons x y	=	cons^S x y	
hd x	=	hd^S (force x)	similarly for tl
x + y	=	(force x) +^{S} (force y)	similarly for -, *, >, *etc.*

Derived Transformation Rules

cons e_1 e_2	→	cons^S (delay e_1) (delay e_2)	
hd e_1	→	hd^S (force e_1)	similarly for tl
e_1 + e_2	→	(force e_1) +^{S} (force e_2)	similarly for -, *, >, *etc.*
f $e_1 \cdots e_n$	→	f (delay e_1) $\cdots$ (delay e_n)	if f's arity is n
f $e_1 \cdots e_{i-1} e_i e_{i+1} \cdots e_n$	→	f (delay e_1) $\cdots$	
		(delay e_{i-1}) e_i (delay e_{i+1})	
		$\cdots$ (delay e_n)	if f is strict in its ith argument

Note: The S superscript indicates a strict primitive.

Figure 2.3: Conversion of Lazy Programs to Strict Programs

Henderson's transformation is summarized in Figure 2.3. (delay e) is not a procedure application, but is instead a special form which evaluates to an unevaluated representation of e called a *thunk*.[4] As we mentioned before, thunks are first-class objects and may be passed freely between procedures, stored in data structures, *etc.* The force procedure takes an object, and if it is a thunk, evaluates the unevaluated expression recorded in the thunk, repeating the process as necessary to obtain a non-thunk value. To achieve anything approaching efficiency, force must also *memoize* the value of the thunk so that later forces of the same thunk will simply fetch the memoized value. Without being any more specific about the representation of a thunk, we note that a thunk must carry a proper environment for evaluating e, if e has any free variables.

The overhead of performing lazy evaluation compared to strict evaluation is directly reflected

inline substitution. After these optimizations the combinator code is translated back into lambda-expressions; this whole process is shown as "pre-processing" in Figure 2.2. In effect, the pre-processing phase is just source-to-source optimization, which happens to use combinators internally.

[4] Also called a *promise*, *recipe*, or a *closure*. We will use the term *closure* in a broader sense, referring to the code/environment pair compiled from any lexically nested procedure definition, whether part of a thunk or not. The term *promise* will be given a special meaning in Chapter 8 as a placeholder for an unevaluated value.

by the appearance of force and delay in the transformed code. Wherever a delay appears, the compiled program must include code to allocate storage for the thunk (enough to hold a pointer to the code for *e*, a pointer to its environment, and later, its evaluated value) and code to construct the thunk's environment. Wherever a force appears, there must be code to check whether the value being forced is a non-thunk value, previously evaluated thunk, or unevaluated thunk. In the latter case, there may also be overhead in setting up the environment for the thunk's expression, usually in the guise of saving registers or pushing stack.

Efficiency is improved, therefore, by eliminating as many forces and delays as possible. The derived rules in Figure 2.3 are examples of this. For example, since + forces both its arguments, there is no need to delay arguments when we can recognize a call to + is being made. A procedure that always forces a particular argument is said to be *strict* in that argument. *Strictness analysis* [20, 36, 16] is commonly employed to determine the strictness of user-defined procedures; strictness information allows delays to be eliminated as illustrated by the last derived rule in Figure 2.3. We will discuss strictness analysis and its applications in detail in Section 2.5.

The ALFL compiler from Yale [34] is one example of a lazy compiler based directly on the force-and-delay transformation. To implement force and delay, the ALFL compiler uses Henderson's observation that a delayed expression may be represented as a procedure of no arguments, so that forcing the expression is just calling that procedure. The resulting translation of delay and definition of force are shown in Figure 2.4. The output of the ALFL compiler is actually Scheme code, which is then fed to the Orbit compiler to produce object code for a sequential machine. Notice that all of the efficient procedure call machinery in Orbit carries over directly to the implementation of thunks in the ALFL compiler, since they are translated into procedures before processing by Orbit. Thus, the ALFL compiler achieves excellent performance relative to other lazy compilers, because Orbit is able to optimize the management of environments for thunks.

2.3 Lazy Compilers: Graph Reduction

Graph reduction was among the earliest lazy implementations of functional languages [72], and provided much of the impetus for research in the field. The idea in graph reduction is to translate the original program into a *combinator graph*, which is "executed" by successive application of

```
(delay e)  →
  {thunkfn () =
    %% Sequential code follows.
    {val = e;
     (code to update thunk with val);
     return val};
   thunk = mk_thunk thunkfn;
   in
     thunk}
```

```
force x =
  if isthunk(x) then
    if thunk_evaluated(x) then
      fetch_memoized_value(x)
    else
      (thunk_closure x) ()    else
    x
```

Figure 2.4: Implementation of force and delay

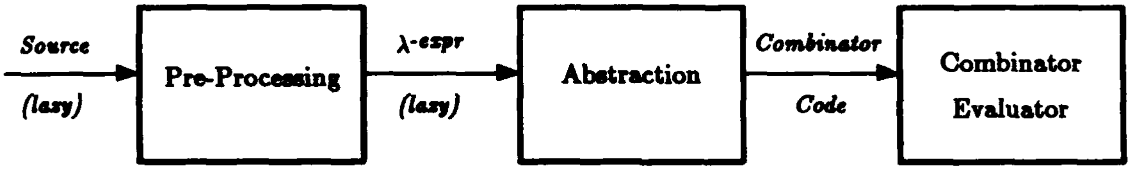
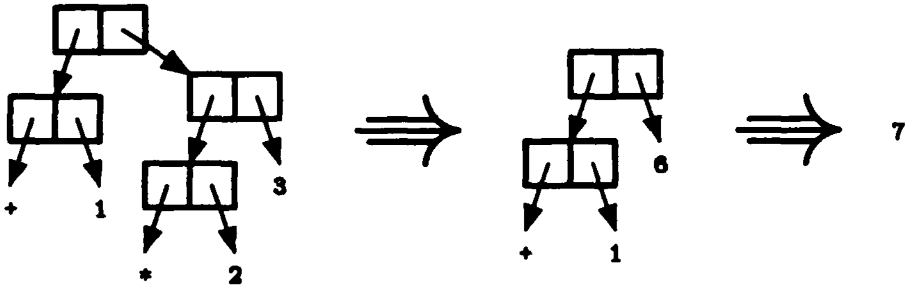


Figure 2.5: Graph Reduction Lazy Compilation

rewrite rules which transform the graph into the final answer. The object code from a graph reduction compiler, therefore, is not object code for conventional hardware, but code which must be executed by a software interpreter (see Figure 2.5). Graph reduction is of interest here only because it forms the basis for the abstract machine compilers, to be discussed in the next section. It should be pointed out, however, that much research effort has been devoted to hardware architectures which directly execute combinator graphs, both uniprocessor [62, 69] and multiprocessor [56].

Here is an example showing the reduction of the program (1 + (2 * 3)) in two steps:



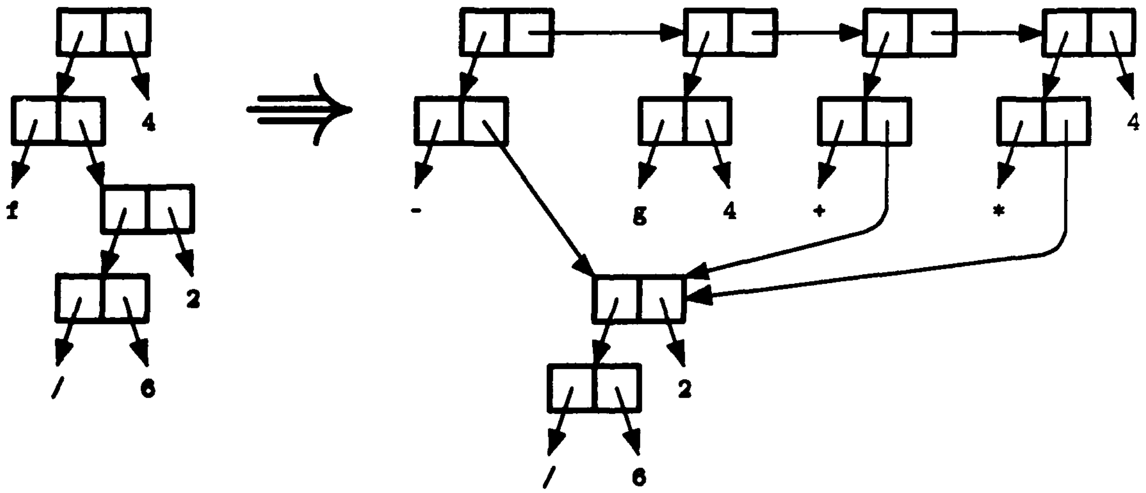
The nodes of a combinator graph represent application, and since all functions are curried in graph reduction, the first graph is just the graphical representation of ((+ 1) ((+ 2) 3)).

The leaves of the graph are constants, which include *combinators* such as +. Associated with each combinator is a reduction rule, and the reduction rule for + says that an instance of + applied to two integers is reduced to the integer which is their sum. When a subgraph is reduced, the reduced version replaces the original in the graph, as shown in the second graph above. Performing the first reduction exposes another opportunity for reduction, and this second reduction reduces the graph to the answer. In general, there many be many reducible subgraphs ("redexes"), and a proper choice of which to reduce at each step is necessary to achieve lazy evaluation. The rule which corresponds directly to the rule for lazy evaluation given on page 15 always selects the leftmost redex, which in terms of the graph is the subgraph located closest to the bottom of the graph's "spine", which extends to the left from the root node. More elaborate strategies are also possible [12].

In addition to primitive reduction rules such as those for +, user-defined functions can be treated as reduction rules in their own right. Suppose we had the following program:

```
{f x y = x - (g y (x + x*y));
 g a b = if a < 0 then b else 7;
 in
   f (/ 6 2) 4}
```

This program's graph before and after the first step of reduction is shown below:



Notice that the argument (/ 6 2) to f is effectively delayed, since it remains unevaluated after the call to f is reduced. Furthermore, when it is eventually evaluated, its value will replace the top node of the subgraph which represents it. Since the reduction of f caused the unevaluated subgraph to be shared, so will the result. Hence, the values of delayed expressions are memoized when evaluated, just as they were in the force-and-delay implementation discussed earlier.

In graph reduction, all of the subexpression delaying mechanism needed to support lazy evaluation is embedded in the graph interpreter. All the compiler must do is produce a set of combinator reduction rules and a graph representing the query expression. Producing combinator definitions is slightly more difficult than our example above implies, for a function definition can only be used as a reduction rule if it has no free variables. One technique for removing free variables from a function is simply to include them in the formal parameter list and in every reference to that function. This method is called *lambda lifting* [42], and is illustrated below:

```
f x y l =                         f x y l =
  {g a = a + y;                     {g yy a = a + yy;
   in                      ⟶         in
     (map g l), g x};                  (map (g y) l), g y x};
```

Another technique is Hughes' *mfe*-abstraction [38]. Any technique which adds new combinators to the set of primitives is called a ***supercombinator*** technique; it is also possible to compile a program into a fixed set of primitive combinators [72]. The fixed set approach is not of much practical use on conventional hardware, but has been used in specialized machines where the set of combinators can be viewed as an instruction set [46, 69].

The main difficulty with combinator approaches is that they are basically interpretive, since they do not result in code directly executable by conventional hardware. In fact, the only difference between graph reduction and what is normally called an interpreter is the sharing of computation made possible by the graph representation. The desire to implement graph reduction efficiently led to the development of abstract machine approaches, discussed next.

2.4 Lazy Compilers: Abstract Machines

Abstract machine compilation produces object code for conventional hardware which mimics the behavior of a graph reduction interpreter, and therefore of a lazy evaluator. In graph reduction, each combinator is a reduction rule which controls the behavior of a graph interpreter. In the abstract machine approach, a combinator is compiled into conventional object code which performs graph manipulations simulating an application of that combinator's reduction rule. In effect, the graph interpreter has been compiled into the combinator definitions. The term

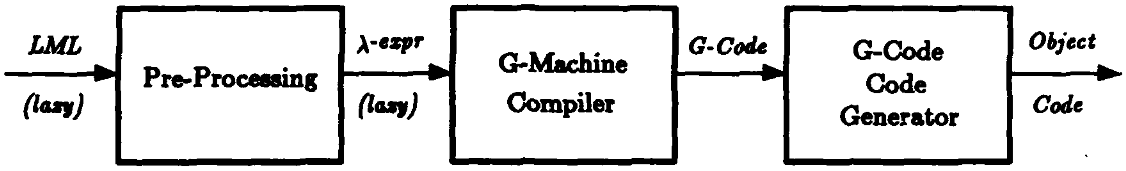


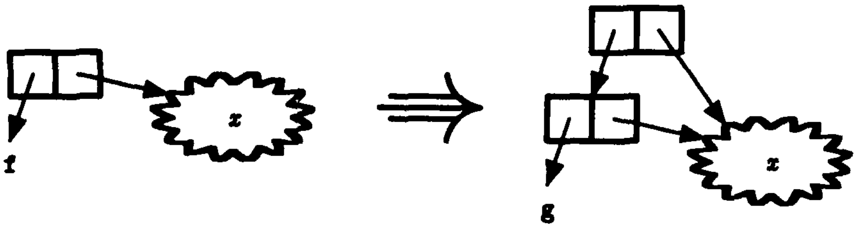
Figure 2.6: Abstract Machine Lazy Compilation: the LML/G-Machine Compiler

"abstract machine" refers to an intermediate step in the compilation process: the program is first compiled into code for an abstract machine whose instructions include primitives for manipulating combinator graphs, and then from abstract instructions into conventional object code. One well-known abstract machine compiler is the compiler for Lazy ML [10], outlined in Figure 2.6, based on the G-machine [41] abstract machine. Another recently proposed abstract machine is the Three Instruction Machine (TIM) [23]. In our discussion here we ignore the intermediate code and just describe in general terms the target code produced from it.

Suppose we had this combinator definition:

```
f x  = g x x
```

where g is the name of another combinator. In graph reduction, reducing a call to f would first rewrite the graph:



and then proceed to reduce the newly rewritten graph. The LML compiler imitates this with the following code for f (expressed in an informal "quads" notation):

```
function f (x, Top)
Temp1  := mkap(g, x)
Temp2  := mkap(Temp1, x)
Ans    := eval(Temp2)
Top[*] := Ans
return Ans
```

Eval evaluates a graph by traveling down the spine of its argument until a the name of a combinator is found. If its arity is satisfied, pointers to its argument graphs (found in cells along the spine) are passed to the code for that combinator as arguments. Also passed as argument is a pointer to the top node of the combinator call, so that it can be updated with the answer. So the x in the code above is really a pointer to the subgraph for x, and Top points to the original application node.

The interesting aspect of the G-machine approach is that the code for a combinator can perform computation instead of just building graphs. Suppose that instead of (g x x), the right hand side of f were (+ x x). With the translation given above, the code for f would build the graph for (+ x x) and then evaluate it, ultimately causing x to be evaluated and added to itself. The actual LML compiler would instead generate the following code:

```
function f (x, Top)
Temp1   := eval(x)
Ans     := Temp1 + Temp1
Top[*]  := Ans
return Ans
```

which just evaluates the graph for x, adds it to itself, and updates the graph.

The LML compiler was able to avoid building the graph for (+ x x) because the graph was to be evaluated immediately after construction. Of course, graphs still need to be built when arguments must be delayed according to the principles of lazy evaluation. Consider again the example from page 23:

```
{f x y = x - (g y (x + x*y));
 g a b = if a < 0 then b else 7;
 in
   f (/ 6 2) 4}
```

In compiling f, the LML compiler would not build a graph for the subtraction, nor for the application of g. On the other hand, a graph *would* be built for (x + x*y), since that expression might be ignored by g. Early G-machine compilation schemes would actually build a graph containing + and * combinators. On the other hand, it has been pointed out [43, 55] that more efficient code is possible by defining a combinator h for this subexpression:

```
{f x y = x - (g y (h x y));
 h x y = (x + x*y);
 g a b = if a < 0 then b else 7;
 in
   f (/ 6 2) 4}
```

Now, the delayed expression `(x + x*y)` appears in the graph as `(h x y)`, and `h` is compiled into efficient code which just performs an addition and multiplication of values on the stack. If this transformation is applied consistently, we get a new combinator like `h` for each delayed subexpression, and the only graph nodes built are applications of these combinators to the delayed expressions' free variables (which may themselves be delayed expressions).

A moment's reflection reveals that we have come full circle and arrived at nothing more or less than a force-and-delay implementation. A delayed expression appears in the graph as a pointer to code (the combinator) along with an environment for its free variables (the arguments to which it is applied). So the graphs built are really just thunks. When such a graph is reduced it is overwritten with its value, but this is equivalent to the memoization of a thunk's value when forced. Given the high quality of target code that can be generated from G-code, it seems that the G-Machine LML compiler can be very competitive with the force-and-delay ALFL compiler. The same is probably true for other abstract machine based compilers.

From a pedagogical point of view, force-and-delay style lazy compilation is a bit easier to deal with, since the force-and-delay notation is succinct and avoids extraneous notions like combinator graphs. In the remainder of this thesis, therefore, we will take force-and-delay compilers as representative of the state of the art in lazy compilation, knowing that the abstract machine approach yields essentially equivalent results. In fact, the abstract machine formalism can impose limitations on object code quality not present in the force-and-delay style. For example, we saw in Section 2.2 how Orbit can avoid building thunk environments in the heap when it can show that the thunk does not escape upward—it compiles the thunk to fetch its free variables directly from its parent stack frames. There seems to be no way to accomplish a similar optimization in the G-machine framework, because activations do not have pointers to their parent frames. Of course, there is every reason to believe that a suitably modified G-machine or some other abstract machine could form the basis of a compiler as good as a force-and-delay compiler. For our purposes, the important fact is that the approaches are essentially equivalent despite their very different appearance.

2.5 Strictness Analysis

Delayed expressions are the principal source of overhead in lazy implementations, and so good lazy compilers expend a lot of effort in eliminating as much delaying as possible without disturbing the termination properties of the program. To discover opportunities for this optimization, current lazy compilers rely on various forms of ***strictness analysis***.

Strictness analysis is based on the notion of a strict function:

Definition 2.1 *A function f of n arguments is* strict *in its ith argument if*

$$f(x_1, \ldots, x_{i-1}, \bot, x_{i+1}, \ldots, x_n) = \bot$$

for all $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$.

As Mycroft points out [52], if a function f is strict in its ith argument it is always safe to evaluate that argument before calling f. Why? Assume that $f(e_1, \ldots, e_n)$ terminates; its value is therefore greater than $\bot$. Since f is strict in its ith argument, e_i must also be greater than $\bot$. But if e_i were never evaluated by f, it would carry no more information than $\bot$. So we conclude that e_i is always evaluated by f whenever f terminates, and so it is safe to evaluate e_i before calling f.[5] On the other hand, if f does not terminate, nothing is changed by evaluating e_i early, even if e_i also does not terminate.

Strictness analysis attempts to determine in which arguments, if any, the functions of a program are strict. When compiling a call to a function, delays are eliminated from strict argument positions as determined by the analysis (see the last derived rule in Figure 2.3). Of course, strictness is an undecidable property [52], so any method of strictness analysis will be to some degree an approximation; a great deal of research has been devoted to finding strictness analysis methods that are more precise and/or more efficient. Mycroft's original work as well as a number of other efforts [20, 36, 16] are based on ***abstract interpretation*** [22] of the source program. Another approach is ***backwards analysis*** (also called *context analysis*), which appears to have some practical advantages over abstract interpretation [39, 77].

If higher-order functions are allowed, strictness analysis becomes more complex. Calls where the name of a function is syntactically applied to all arguments (*i.e.*, a first-order call) can be

[5] Actually, we can only conclude that e_i can be evaluated sometime before the call to f *returns*. To also conclude that e_i can be evaluated before anything inside of f is computed, we must note that since f is free from side effects, no computation within f can possibly affect e_i before f returns. Evaluation of e_i before f begins is therefore indistinguishable from evaluation of e_i just before f returns. Note that this is not true for non-strict languages with side-effects like Id [53] (with unrestricted use of I-structures) or Josephs' language [45], and so strictness is not a valid criterion for early evaluation of arguments in those languages.

optimized as before, but a more complex analysis is needed to optimize a general application of one expression to another. In such a situation, the delay of the argument can be removed if it can be shown that all possible values for the function expression are strict in that argument. Abstract interpretation methods can be extended to handle higher-order functions [36, 16], but more work seems to be needed to make these techniques practical. It should be noted, however, that first-order strictness analysis can be used with higher-order languages, as long as pessimistic decisions are taken when general applications are encountered.

A more serious shortcoming of present-day strictness analysis is its ineffectiveness on non-flat domains (*i.e.*, data structures). Suppose a unary function f is strict in its argument according to the definition above, and that the argument expression e in some call to f returns a tuple. From our previous discussion, we know that we have to get more information from e than $\perp$ if f is to terminate. Unfortunately, we do not know *how much* information: we could evaluate e to $\langle \perp, \perp \rangle$, $\langle v_1, \perp \rangle$, $\langle \perp, v_2 \rangle$, or $\langle v_1, v_2 \rangle$ (and of course this continues recursively if v_1 or v_2 are themselves data structures). We are forced to take the most conservative position and only perform the minimum evaluation possible on e; in the force-and-delay framework, this means we can force e before the call, but not any of its components. Putting it another way, we can never remove delays from the arguments to a data constructor. This is a very serious defect considering the frequency with which calls to data structures are made in typical functional programs. Extending strictness analysis to eliminate delays from data structure creation is an active area of research [77].

Finally, we should note that some researchers [17, 35] have proposed the use of strictness *annotations*, which allow the programmer to direct the compiler to remove a particular delay regardless of the consequences.

Chapter 3

Lenient Evaluation

At the beginning of Chapter 2 we noted that lazy evaluation gives more expressive power to the programmer than does strict evaluation, because there are programs which produce answers under lazy evaluation but not under strict evaluation. As we have seen, though, this expressive power comes at the expense of run-time overhead, in the form of delayed computation. Given that there is a trade-off between expressiveness and efficiency, it is reasonable to ask whether there is some other evaluation strategy which is more expressive than strict evaluation yet more efficient than lazy evaluation.

To answer this question, we examine the ways in which lazy evaluation is more expressive than strict evaluation. We focus first on *non-strictness* in lazy evaluation—the property that arguments are passed unevaluated to procedures. We show how non-strictness allows fuller use of recursion than is possible under strict evaluation, and therefore gives the programmer the ability to specify more complicated patterns of data dependence. From there we see that lazy evaluation imposes stronger constraints on evaluation than needed simply to achieve non-strict behavior. These constraints stem from *laziness*, which seeks to prevent evaluation that does not contribute to the final answer. Laziness is responsible for yet a different kind of expressive power: the ability to manipulate "infinite" data structures. The next logical step is to imagine implementing non-strictness but not necessarily achieving laziness. We call this *lenient evaluation*, and examine how it is achieved and how it allows for reduced overhead compared to lazy evaluation.

3.1 Expressive Power from Non-Strictness

To illustrate how non-strictness allows a more unrestricted use of recursion, we present this small example:

```
{a = cons 2 (hd a);
 in
   a}
```

We would like this program to return ⟨2,2⟩, but this requires that (hd a) be passed to cons unevaluated—we cannot evaluate (hd a) before calling cons, as a has no value until cons returns. The need for non-strictness extends to the calling of procedures, as a slight modification of this program shows:

```
{f x y = cons x y;
 a = f 2 (hd a);
 in
   a}
```

Neither of these programs is correctly executed by a strict interpreter, because the strict interpreter tries to evaluate (hd a) before a has a value. Depending on the implementation, the strict interpreter will either complain about a being undefined, or simply deadlock.

The reader unfamiliar with strict functional languages may wonder at this point how letrec is implemented in those languages. We quote from the Scheme manual [59]:

> The [variables on the left hand sides] are bound to fresh locations holding undefined values; the [expressions on the right hand sides] are evaluated in the resulting environment (in some unspecified order); each variable is assigned to the result of the corresponding [right hand side]; the [final expression] is evaluated in the resulting environment; and the value of the [final expression] is returned. Each binding of a variable has the entire letrec expression as its region, making it possible to define mutually recursive procedures.
>
> One restriction on letrec is very important: it must be possible to evaluate each [right hand side] without referring to the value of any variable. If this restriction is violated, then the effect is undefined, and an error may be signalled during evaluation of the [right hand sides]. ...In the most common uses of letrec, all the [right hand sides] are lambda expressions and the restriction is satisfied automatically.

Lambda expressions on the right hand sides satisfy the restriction automatically because evaluating them only means constructing a closure for them, requiring the *locations* of the left hand side variables but not their values.

In the rest of this section, we present some more examples of programs which exploit the free use of recursion allowed by non-strictness; all of them make use of non-lambda right hand sides in letrec expressions.

Before proceeding to the examples, however, we should clarify what we mean by the term *non-strictness*, a term frequently used in the literature but not defined very precisely. We will use the term to describe a particular semantic property of a functional programming language, and by extension to describe a functional language implementation which achieves that property. By non-strict we do not simply mean anything that is different from strict semantics; that would not be a very useful definition. Instead, we note that the distinguishing feature of strict semantics is that arguments to procedures and data constructors are completely evaluated before the procedure body or data constructor is invoked. By non-strict, then, we mean that arguments are not necessarily evaluated before a procedure or data constructor is invoked. But more than this, we *require* that an implementation invoke the procedure or data constructor before the arguments are evaluated if this is the only way progress can be made. In the example above, progress could not be made unless **cons** was called before evaluating the argument (hd a). The semantic property that thus becomes available to the programmer is the assurance that computation will be delayed whenever necessary to allow the continued execution of the program.

3.1.1 Cyclic Data Structures

Cyclic data structures are often useful, but are impossible to construct in a functional language without non-strictness. Here is a simple example of constructing a circular list:

```
{a = cons 1 (cons 2 (cons 3 a));
 in
   a}
```

Non-strictness is required because of the reference to **a** on the right hand side of **a**'s binding.

A less trivial example constructs a doubly-linked list. An ordinary list is a collection of cells, where each cell is a two-tuple containing that cell's element and the next cell: $c_i = \langle v_i, c_{i+1} \rangle$. The cells of a doubly linked list are three-tuples which contain the next cell and also the previous cell: $c_i = \langle v_i, c_{i-1}, c_{i+1} \rangle$. The following program constructs a doubly-linked list from an ordinary list:

```
doubly_link l prev =     % Initial call: (doubly_link l nil)
  if nil? l then
    nil
  else
    {this = 3_tuple (hd l) prev next;
     next = doubly_link (tl l) this;
     in
       this};
```

Again, non-strictness allows the mutual recursion between the definitions of `this` and `next`.

3.1.2 Dynamic Programming

In dynamic programming, a table of some sort is constructed where most elements are defined in terms of other elements already computed. Here is a program which generates a list of factorials from 1 to n using dynamic programming:

```
make_fact_list n =
  {fact_list = cons 1 (gen_fact_list 2 n);
   gen_fact_list i n =
     if i > n then
       nil
     else
       cons (i * (nth (i - 1) fact_list))
            (gen_fact_list (i + 1) n);
   in
     fact_list}
```

Here, each element of the list is defined in terms of the previous element; this is done by using the `nth` function to read the $(i-1)$th element when computing the ith element. As a result, `make_fact_list` only does $O(n)$ multiplications, compared to the $O(n^2)$ that would be required if it simply computed $i!$ for each value of i.

In this example, the data structure is not cyclic, but the data dependences in the program are: `fact_list`'s definition calls `gen_fact_list`, but `gen_fact_list` examines the value of `fact_list`. The circular dependences are necessary in functional programming, because the programmer cannot take the imperative approach of allocating an empty structure and filling in the components one at a time. The nearest functional equivalent of the imperative approach would *copy* the entire data structure at each step, at a prohibitive cost in time and space.[1]

[1] In some instances, the compiler could optimize the copying program into one which updates a structure in place; see [32, 30] for details.

Non-strictness allows the data structure to be constructed only once even though its definition "reads itself". This dynamic programming technique shows up in such well-known examples of functional programming as Turner's "paraffins" program [73] and the "wavefront" program for matrices [8], and can even be used to solve the "backpatching" program cited by advocates of logic programming [58].

The factorial example is somewhat artificial in that dynamic programming is not necessary to generate a list of factorials efficiently:

```
non_dynamic_fact_list n =
  {fact_list = (gen_fact_list 1 1 n);
   gen_fact_list i prod n =
     if i > n then
       nil
     else
       cons (i * prod)
            (gen_fact_list (i + 1) (i * prod) n);
   in
     fact_list}
```

On the other hand, this kind of table generation is often programmed using bulk data constructors, as in the following program which uses a general purpose `make_list` primitive:

```
yet_another_fact_list n =
  {fact_list = make_list f 1 n;
   f i =
     if i == 1 then
       1
     else
       i * (nth (i-1) fact_list);
   in
     fact_list}
```

`(make_list f 1 n)` constructs a list each of whose elements is $f(i)$, for i from 1 to n. Primitives like this are more commonly used to construct arrays—indeed, they may be the only option available to the functional programmer [8]—but to use these primitives efficiently, non-strictness is once again essential.

The point of all this, of course, is not to compute factorials more efficiently, but to illustrate how non-strictness allows more efficient programs than are possible under strict evaluation.

3.1.3 Conditional Dependence

This program is a bit contrived, but illustrates that the applications of non-strict recursion are not limited to data structures:

```
conditional_example x =
  {a = if x > 0 then bb else 3;
   b = if x < 0 then aa else 4;
   aa = a + 5;
   bb = b + 6;
   in
     aa+bb};
```

The order in which this program's subexpressions are evaluated depends on the value of x. If x is positive, we first evaluate b (4), then bb (10), then a (10), then aa (15), and finally the answer (25). If x is negative, however, we first evaluate a (3), then aa (8), then b (8), then bb (14), and the answer (22).

This program has the surprising behavior that the order in which the two additions are performed depends on the value of x, even though those two additions lie outside the conditionals.

3.2 Expressive Power from Laziness

In the last section, we saw that non-strictness allows more complicated patterns of data dependences than are allowed under strict evaluation: we could make cyclic data structures, define structures in terms of themselves, even have apparent cyclic data dependences that "untangle" themselves at run time. In all of these programs, evaluation of some expressions needed to be delayed until their evaluation was possible—strict evaluation would always try to evaluate them too early, when some of their free variables were still undefined. Lazy evaluation delays *every* expression, and delays each one until the last possible moment, when execution cannot proceed further without it. So it should come as no surprise that all of the programs in the last section execute correctly under lazy evaluation, since lazy evaluation will certainly introduce sufficient delays to insure that the delayed expressions are executable. In fact, if there is any evaluation order which will produce an answer from a program, lazy evaluation will also produce an answer from that program [75]. So lazy evaluation enjoys all the expressive power that comes with non-strictness.

But lazy evaluation provides some additional expressive power beyond that discussed in the previous section: the power to manipulate "infinite" objects. The most famous example of this is the functional sieve of Erastothenes for finding prime numbers. We quote the description of the sieve given by Henderson [29]:

> Make a list of the integers, commencing at 2. Repeat the following process of marking the numbers in the list:
>
> 1. The first unmarked number of the list is prime, call it p.
> 2. Mark the first unmarked number in the list and each pth number thereafter whether previously marked or not (here you are marking all multiples of p).
> 3. Repeat from 1.

Typically, the sieve is used to find all primes less than some number n, so the initial list is a list from 2 to n. An imperative algorithm might start with an array, and at Step 2 scan through the array marking off array elements (say by setting them to zero). Of course, this approach does not carry over into the functional framework, so the usual solution uses a list of numbers, and "marking" multiples is accomplished by making a new list with the multiples removed:

```
sieve l =
  if nil? l then
    nil
  else
    cons (hd l)
         (sieve (remove_multiples (hd l) (tl l)));
```

Here `(remove_multiples i l)` returns a list containing those elements of input list `l` which are not multiples of `i`. If we want a list of all primes less than 100, then, we need only call `sieve` on a list of the integers from 2 to 100.

Now suppose that instead of all primes less than n, we want the first n primes. This is a bit harder to do with the sieve, since we do not know how big to make the original list of integers. Here is where lazy evaluation enters the picture: we simply start with a list of *all* the integers.

```
{ints_from x = cons x (ints_from (x+1));
 all_primes = sieve (ints_from 2);
 in
   first_n_elements n all_primes}
```

`(ints_from 2)` is a list of all integers starting with 2, and so `all_primes` is the list of all the primes, from which we simply extract the first n elements. Now if we tried executing this program under strict evaluation, we would go into an infinite loop as we actually tried to build the entire list of integers. But lazy evaluation does no computation not required for the final answer, so no more than the first n elements of **all_primes** are evaluated, and no more iterations of **ints_from** are performed than the value of the nth prime (541, if $n = 100$).

This is a very different sort of expressive power than we saw in the previous section. In that section, we saw how non-strictness allowed complicated patterns of data dependence. In the sieve, the data dependences are straightforward, but the *control flow* is not. **Ints_from** appears to be an infinite loop, because there is no conditional to break the recursion. **Sieve**, too, is apparently infinite: the **nil?** predicate in its conditional never returns true because the input list is infinite, and so the conditional there is superfluous (and is omitted in [29]). The control over these recursions comes not from code within the procedures themselves, but from the procedure that reads the **all_primes** list, namely, **first_n_elements**. Lazy evaluation has effectively induced the control structure of **first_n_elements** onto **sieve** and **ints_from**.

3.3 Lenience: Non-strictness Without Laziness

We have seen that there are two distinct kinds of expressive power available in lazy evaluation not found in strict evaluation. One is the ability to specify complicated patterns of data dependence, specifically, data dependences which have the syntactic appearance of cycles but which are resolved because they go through different components of the same data structure, or because the cycles are broken by conditionals. The other is the ability to express complicated control structure, where the execution of procedures which produce a data structure is controlled by the procedures which use that data structure. Both require some delaying of evaluation: the former (*non-strictness*) requires that expressions be delayed until all of their input data are available (already computed or obtainable by forcing some other delayed expressions), while the latter (*laziness*) requires that they be delayed until no further progress can be made without them. Since laziness delays expressions until the last possible moment, it automatically introduces enough delay to achieve non-strictness.[2]

[2] We wish to reiterate a rather subtle point alluded to in Section 2.5. Lazy evaluation delays expressions until the last possible moment, but with strictness analysis some arguments to some procedures may be evaluated early, before the call is made. It is tempting to think that this works because strictness analysis shows that an argument is always forced, *i.e.*, known to contribute to the answer before the call is made. While this is true, a stronger condition is required to eliminate the delay: the argument must be evaluable (have all its input data available) before the call. As we pointed out in the footnote on page 28, if an argument is always forced it must be evaluable before the call since no information can propagate to the argument from the call until it returns.

We define *lenient* evaluation to be an evaluation strategy which achieves non-strictness but not necessarily laziness. For the implementation, this implies that expressions must be delayed until their input data are available, but not necessarily any longer. For the programmer, it means unrestricted use of letrec and the attendant flexibility in data dependences, but not the consumer-directed control structure required for the finite use of infinite data structures. More operationally, it means that conditionals are the only means of controlling whether or not expressions are evaluated, and should be included whenever there is the possibility of infinite loops. The control structure available under lenient evaluation, therefore, is exactly what is available under strict evaluation.

We should point out that strict languages like Scheme provide explicit force and delay operators to the programmer, so that infinite object algorithms can be implemented. The same option is available to lenient implementations. A variation is to provide explicit delays but *implicit* forces, making it transparent to the consumer of a data structure whether it is infinite or not. Programming style and methodology for lenient languages with explicit delay constructs is a topic of current research [28].

3.3.1 An Example of Lenient Object Code

Lenient evaluation does not delay expressions as long as does lazy evaluation, so we expect it to have less delay-related overhead. Specifically, lenient evaluation allows more delays to be eliminated at compile time, and in other cases allows separate delayed expressions to be combined. To illustrate, let us return to the fact_list example from Section 3.1.2.

```
make_fact_list n =
  {fact_list = cons 1 (gen_fact_list 2 n);
   gen_fact_list i n =
     if i > n then
       nil
     else
       cons (i * (nth (i - 1) fact_list))
            (gen_fact_list (i + 1) n);
   in
     fact_list}
```

Consider how a lazy compiler would compile the internal procedure gen_fact_list. The best lazy compiler would introduce forces and delays as follows:

The same implication does not hold for the delayed expressions of a letrec; all of the right-hand sides of the program in Section 3.1.3 are forced, but they still must be delayed for the dependences to be unraveled.

```
gen_fact_list i n =
  if i > n then
    nil
  else
    cons (delay (i * (nth (i - 1) fact_list)))
         (delay (gen_fact_list (i + 1) n));
```

Strictness analysis reveals nth strict in all its arguments, and gen_fact_list strict in i and n. Thus, no delays appear in the calls to these procedures, nor is there need to force i or n (since all calls to gen_fact_list will evaluate them before the call). The object code generated would be as follows (in informal "quads" notation):

```
       function gen_fact_list (i, n, fact_list)
       Temp1   := i > n
       if Temp1 goto L1
       Ans     := allocate 2
       Temp2   := ⟨Thunk for Th1 closed over i, fact_list, Ans⟩
       Temp3   := ⟨Thunk for Th2 closed over i, n, Ans⟩
       Ans[1]  := Temp2
       Ans[2]  := Temp3
       return Ans
L1:    return nil

       function Th1 (i, fact_list, cell)
       Temp1   := i - 1
       Temp2   := call nth(Temp1, fact_list)
       Ans     := i * Temp2
       cell[1] := Ans
       return Ans

       function Th2 (i, n, cell)
       Temp1   := i + 1
       Ans     := call gen_fact_list(Temp1, n)
       cell[2] := Ans
       return Ans
```

This notation omits the details of procedure linkage and thunk representation. We assume that the compiler chooses the best linkage and thunk representation possible (see Section 2.1), but it should be noted that both thunks in this example escape upward, and so require expensive heap-allocated environments.

Now consider generating lenient code for gen_fact_list. When we reach the "else" arm of the conditional we know that the values of i and n are available, since they were needed by the predicate. On the other hand, fact_list is not available, since its value is computed by gen_fact_list and will only be available after it returns. The expression (i * (nth (i - 1) fact_list)) must be delayed, therefore, but (gen_fact_list (i + 1) n) need not.

```
gen_fact_list i n =
  if i > n then
    nil
  else
    cons (delay (i * (nth (i - 1) fact_list)))
         (gen_fact_list (i + 1) n);
```

The object code is as follows:

```
        function gen_fact_list (i, n, fact_list)
        Temp1   := i > n
        if Temp1 goto L1
        Temp2   := i + 1
        Temp3   := call gen_fact_list(Temp2, n)
        Ans     := allocate 2
        Temp4   := ⟨Thunk for Th1 closed over i, fact_list, Ans⟩
        Ans[1]  := Temp4
        Ans[2]  := Temp2
        return Ans
L1:     return nil

        function Th1 (i, fact_list, cell)
        Temp1   := i - 1
        Temp2   := call nth(Temp1, fact_list)
        Ans     := i * Temp2
        cell[1] := Ans
        return Ans
```

We see that for this example, lenient evaluation has removed half the overhead needed for lazy evaluation.

3.4 Semantics or Implementation Technique?

We have discussed at length the relative expressive power of strict, lenient, and lazy implementations of functional languages. Our concern with expressive power is mainly for expository reasons. By examining functional programs from which we expect a certain kind of behavior, we can see what an implementation must do to achieve that behavior. We can then capture the behavioral differences in succinct operational terms: lenient implementations add non-strictness to strict implementations, lazy implementations add laziness. The operational notions correspond directly to kinds of expressive power: non-strictness allows more complex data dependences, laziness allows more complex control structure. This view has allowed us to emphasize the distinction between non-strictness and laziness, which heretofore have been considered synonymous in the literature.

But even though strict, lenient, and lazy implementations differ in expressive power, it is not entirely accurate to think of them as having three distinct semantics. More to the point is that there is one semantics for functional languages, which strict, lenient, and lazy implementations implement with varying degrees of faithfulness.[3] So strict, lenient, and lazy evaluation can be thought of as compilation techniques rather than as separate classes of languages, which differ in their approach to subexpression scheduling. Strict compilation bases scheduling on the syntactic call graph, lenient compilation employs an analysis of data dependence to schedule based on availability of data, while lazy compilation schedules based on whether an expression is needed for the final answer. As we have remarked earlier, strict, lenient, and lazy compilation are respectively more faithful to the standard semantics, but also respectively introduce more run-time overhead.

3.5 Compilation as Ordering: Sequential Threads

As we discussed in Chapter 2, existing lazy compilers are based on one of two approaches. Force-and-delay compilers view compilation as introducing force and delay primitives into the program to convert it to strict semantics. Abstract machine compilers try to generate code which emulates the behavior of an abstract graph reduction architecture. As we have seen, both yield approximately equivalent code, but the way in which they go about it is heavily influenced by the basic view they take.

The starting point for the compilation method presented in this thesis is simply that compilation is a process of choosing the order in which subexpressions will be evaluated. Unlike imperative programs, functional programs do not give any explicit indication of subexpression ordering; this is why they are termed "declarative". But sequential code is by definition ordered, so if sequential code is to be produced from functional programs, the compiler must take ordering decisions. Because this ordering information is not explicit in the source code, the

[3] Of course, it is always possible to devise a formal semantics which exactly models a given implementation. For example, Stoy [68] gives a semantics for strict lambda calculus through a modification to the standard semantics which artificially "strictifies" all lambda expressions.

compiler must perform considerable analysis to make these decisions correctly, and in fact most of the compiler technology that is unique to non-strict functional languages is connected with making ordering decisions.

While sequential code demands that ordering decisions be taken at compile time, non-strictness can require that some be deferred until run time. The program in Section 3.1.3 is a good example: the relative ordering of the two statements `aa = a + 5` and `bb = b + 6` depends on input data, and so the compiler cannot sequentialize them even though they lie outside the conditionals which create the variation according to input. The product of compiling a non-strict program, therefore, will not be a single piece of sequential code but a collection of sequential *threads*.

How can we characterize a sequential thread? Certainly a sequential thread is a segment of sequential code, where the instructions are ordered at compile time. Furthermore, the relative order in which different threads execute is *not* fixed at compile time, but allowed to vary according to input data. Of course, this description could apply as well to code segments between conditional branch instructions in ordinary imperative code, or to code generated for individual procedures and subroutines. The key feature is that sequential threads are generated from program portions that do not have any explicit control transfers between them; the run-time switching between threads is directed only by the communication of values between threads. That is, when an executing thread needs a value that is to be computed by another thread, it must suspend its own execution and allow that other thread to proceed. The hallmark of sequential threads, therefore, is scheduling controlled by *tests to determine whether another subexpression has been evaluated*. Code which consists of sequential threads will always make use of *presence bits* of one sort or another. We will use the term "multi-threaded" to describe object code in which a compiled procedure consists of a collection of sequential threads.

With this definition of threads in mind, the force-and-delay code produced by lazy compilers is easily seen to be a type of multi-threaded code. Each *delay* results in a small piece of sequential code, invoked when some other piece of code needs the value it computes. So thunks are just sequential threads, and removing a *delay* from a subexpression amounts to embedding one thread in another. (These comments apply equally well to abstrıct machine based compilers.) Seen in this light, strictness analysis and all of the other compilation techniques unique to lazy evaluation are methods of producing as large threads as possible, minimizing the overhead of switching between them.

3.6 Code Generation Options

By taking a view of non-strict compilation as choosing a set of sequential threads for a program, we immediately separate two classes of concern. The first is deciding what set of threads will be produced from a given program. This is primarily a semantic issue, as the threads must give an ordering among subexpressions which correctly implements the meaning of the program according to the semantics. The second concern is how the multi-threaded mechanism is to be implemented on the target architecture. These issues are mostly pragmatic; they include such details as whether presence bits are implemented in hardware or software, whether threads may execute concurrently, how threads are scheduled, *etc.* This separation is not present in existing lazy compilers, because those compilers take as their starting point a particular implementation of multiple threads (either as forces and delays or as abstract machine code) which fixes both the semantics and most of the run-time mechanisms.

In contrast, the bulk of our compilation method deals with sequential threads very abstractly, so that it addresses the semantic issues independent of the implementation mechanisms. As we have suggested in our extensive discussion of lenient evaluation, we will describe how to produce an appropriate set of threads for lenient semantics. Because lenient evaluation captures the notion of non-strictness in isolation from laziness, the exposition of our compilation method will reveal how non-strictness is responsible for most of the difficulty in taking ordering decisions at compile time. In fact, we will show how lazy code may also be produced in our framework, and it will simply turn out to be a further refinement of the threads produced for lenient evaluation. In this sense, our compilation method is neutral toward the issue of lenient vs. lazy evaluation: either option is supported.

Many options also exist in the mechanisms chosen to implement multi-threaded code. We will discuss a number of these, including whether presence bits are in hardware or software, whether concurrency is supported, and whether threads are scheduled eagerly or on demand. All of these choices turn out to be more or less orthogonal, and all may be chosen independently of whether the threads generated implement lenient or lazy semantics (although demand-driven scheduling is required for lazy evaluation).

So we see that lenient evaluation provides not only an alternative semantics to lazy evaluation, but also a completely new perspective on compiling non-strict programming languages. With the motivation and goals of lenient evaluation firmly in mind, we proceed to the theory and practice of lenient compilation.

3.7 A Footnote: Non-Sequentiality

Before proceeding, we should say a word or two about *non-sequential* programming languages and how they relate to our discussion of multi-threaded code. "Sequentiality" has several different meanings, but the most relevant meaning here is Huet-Lévy sequentiality [12], which for our purposes says that when a thread suspends for lack of some value, it can always identify the thread which will produce that value. Without this guarantee, an implementation might have to evaluate many threads simultaneously in order to be sure of evaluating the one which produces the needed value; if it arbitrarily chose one to which to devote its full attention, it might happen to choose a thread which diverges. Of course, actual parallel hardware is not needed, but to implement a non-sequential language there must be at least a *simulation* of parallelism. A well-known example of a non-sequential construct is "parallel OR", a binary function which returns true if one of its arguments is true, even if the other argument fails to terminate. In this thesis, however, we will only consider functional languages which are sequential in the sense of Huet and Lévy, and so we need not worry about simulating parallelism.

One is tempted to consider the kind of multi-threaded code we have discussed as somehow "non-sequential", since it does not consist of a single sequential thread as is found in most programming language implementations. But, as we have seen, the use of multi-threaded code does not imply non-sequentiality in the sense described above. It is worth noting, however, that compiled code from a non-sequential language *would* likely be multi-threaded.

Chapter 4

Functional Quads

In Section 1.1 we introduced a small kernel functional language, and in Chapters 2 and 3 we described in general terms how programs are evaluated under strict, lenient, and lazy evaluation. We will now formalize all of this, and describe syntax and semantics very precisely.

The centerpiece of our study is a very minimal functional language which we call "functional quads". Functional quads was designed to meet the following goals:

- It should serve as a model of both lenient and lazy evaluation; hence, it must be non-strict.
- It should have primitive constructs for each of the functional language features normally treated as primitive by functional language implementations. Specifically, its primitive constructs should include scalar types (numbers, booleans, *etc.*), arithmetic and other scalar primitives, conditionals, data structures, first-class functions, and "letrec" recursion. It should also reflect the fact that many implementations can treat "first-order" function calls (a known function applied to all arguments simultaneously) more efficiently than higher-order calls.
- It should be minimal, in the sense that it does not include features which can *and ordinarily would* be implemented in terms of more primitive features.
- It should have a well-defined operational semantics which accurately models the behavior of realistic functional language implementations. Specifically, there must be a natural correspondence between operations in the operational semantics and in a functional language implementation, and the operational semantics should accurately model sharing of computation which takes place in an implementation.
- The language and its operational semantics should have a formal structure which makes it easy to study the relationships between the individual computations which comprise the execution of a program.

We should point out that when we speak of functional language implementations we are of course referring to *compiled* implementations, not interpreters.

The mathematical basis of functional quads is as an *abstract reduction system* [37, 49]. In this respect, it is similar to the lambda calculus, and the reader may wonder why we need to introduce a new model of functional computation when the lambda calculus is so well known. The reason is that the lambda calculus fails rather miserably in meeting the second and fourth goals above: in lambda calculus, the *only* primitive constructs are functions and function application, and so the other primitive features mentioned above must be simulated in lambda calculus through the use of functions. Numbers, for example, may be simulated with Church numerals, and recursion through the Y combinator [11]. Such simulations are far removed from their typical implementation as machine arithmetic and cyclic references, and furthermore do not have the same sharing properties as in an implementation. There are, of course, many examples in the literature of extensions to lambda calculus to include primitive data types and functions [11, 48], as well as graphical representations which model certain kinds of sharing [72, 78], but these are still somewhat removed from real implementations, and do not share the wide acceptance and plentiful theory of lambda calculus. The fifth goal above would require still another set of extensions to lambda calculus (*e.g.*, Lévy's labeled lambda calculus [51]). In short, since lambda calculus itself does not meet our needs, we are better off constructing a system that meets them exactly. Since we are modeling functional computation, of course, our system will not be very different in spirit from lambda calculus.

Our preoccupation with operational semantics comes about because the questions raised by sequential implementation—namely, in what order machine instructions will execute—are fundamentally operational in nature. Working with an abstract system instead of actual machine code, however, allows us to abstract away unimportant details, including details of how and where memory is allocated, how presence bits are maintained, procedure linkage and environment representation, *etc.* The term "functional quads" is derived by analogy to the quads notation [3] commonly used as an abstraction of imperative code for von Neumann machines, which hides similar details. Our intuition is that functional quads can play as universal a role as sequential quads, serving as the basis for all kinds of functional language compilers, whether for sequential or parallel architectures, with lenient or lazy semantics, for von Neumann, dataflow, or reduction machines.

In the first part of this chapter, we present the syntax and operational semantics of functional quads along with a discussion of how functional languages relate to functional quads and how functional quads relate to sequential code. In the second part, we investigate the theoretical

$$\begin{array}{lcll}
\mathit{Scalar} & ::= & \mathit{Number} \mid \mathtt{true} \mid \mathtt{false} & \\
\mathit{Struct} & ::= & <\ \mathit{StructTag}^{(n)}\ \underbrace{,\ \mathit{Identifier}^{(0)} \ldots ,\ \mathit{Identifier}^{(0)}}_{n}\ > & n \geq 0 \\
\mathit{Partial} & ::= & (\ \mathit{Identifier}^{(n)}\ \underbrace{\mathit{Identifier}^{(0)} \ldots \mathit{Identifier}^{(0)}}_{i}\ ) & 0 \leq i < n \\
\mathit{Value} & ::= & \mathit{Scalar} \mid \mathit{Struct} \mid \mathit{Partial} & \\
\mathit{Primary} & ::= & \mathit{Identifier}^{(0)} \mid \mathit{Value} & \\
 & & & \\
\mathit{Simple} & ::= & \mathit{Primary} \mid \mathtt{const}\ \mathit{Value} \mid \mathit{Primary}\ \mathit{Op}\ \mathit{Primary} \mid & \\
 & & \mathtt{if}\ \mathit{Primary}\ \mathtt{then}\ \mathit{Block}\ \mathtt{else}\ \mathit{Block} \mid & \\
 & & \mathtt{sel\_}t\_i\ \mathit{Primary} \mid \mathtt{is\_}t\mathtt{?}\ \mathit{Primary} \mid & \\
 & & \mathit{Primary}\ \mathit{Identifier}^{(0)} & \\
\mathit{Op} & ::= & \mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \mid \mathtt{/} \mid \mathtt{==} \mid \mathtt{<} \mid \ldots & \\
 & & & \\
\mathit{Block} & ::= & \{\ \mathit{Binding}\ ;\ \mathit{Binding}\ ;\ \ldots\ \mathtt{in}\ \mathit{Identifier}^{(0)}\ \} & \\
 & & & \\
\mathit{Binding} & ::= & \mathit{Identifier}^{(0)} = \mathit{Simple} \mid & \\
 & & \mathrm{Identifier}^{(n)}\ \underbrace{\mathit{Identifier}^{(0)} \ldots \mathit{Identifier}^{(0)}}_{n} = \mathit{Block} & n \geq 1 \\
\mathit{State} & ::= & \mathit{Binding}\ ;\ \mathit{Binding}\ ;\ \ldots &
\end{array}$$

- A *State* must also be name-consistent, as defined in the text.

Figure 4.1: Syntax of Functional Quads

properties of functional quads as an abstract reduction system. Among our results are some very strong assertions about the relationships between computations performed in different executions of the same program, and about the equivalence of intermediate results obtained in different executions. These results in turn expose the parallelism inherent in functional quads and the relationship between lenient and lazy evaluation.

Functional quads and its reduction system was inspired in large part by the rewrite rule operational semantics for Id given by Arvind, Nikhil, and Pingali [8].

4.1 Syntax

The syntax of functional quads is given in Figure 4.1. Some important details:

- The syntactic category *Identifier* is partitioned into an infinite number of sets, each corresponding to a different *arity*. A parenthesized superscript indicates the arity of an identifier; *e.g.*, *Identifier*$^{(2)}$ is an identifier in the arity 2 set. As the grammar for bindings indicates, the arity indicates how many formal parameters are present in an identifier's definition; the arity will affect how identifiers will be rewritten by the abstract reduction semantics. Arity should not be confused with *type*, which is a semantic property, not a syntactic one.

- Structure tags are also partitioned into sets according to arity, where the arity of a tag indicates the number of components a structure with that tag has. For each n-ary structure tag t there are keywords **sel_*t*_1**, ..., **sel_*t*_*n***, and **is_*t*?** which correspond to selectors and a predicate. No special syntax for constructors is needed as the syntax for *Struct* serves this purpose. Structure tags have no connection with identifiers, and are not first class.
- A *State* is a complete program; by convention it should contain a binding for the special identifier $\Diamond^{(0)}$, whose value is to be considered the result of the program. Typically, a program will consist of a number of bindings defining functions along with a binding for $\Diamond^{(0)}$ which applies one of these functions to some arguments.

Beyond being partitioned according to arity, the syntactic set *Identifier* has a dyadic function *newid* defined over it, which allows the generation of new identifiers from old ones. Specifically, if $A^{(0)}$ and $B^{(i)}$ are identifiers of arity zero and $i \geq 0$, respectively, then $newid(A, B)$ is an identifier of arity i uniquely determined by A and B. There is nothing mathematically unorthodox about this; we are merely asserting that the set *Identifier* has a structure wherein certain members are related to others through the *newid* relation. We will use *newid* when we give the semantics of function application, in which we need new identifiers to construct a copy of the called function, distinguishable from all other calls; through *newid*, each new identifier will be uniquely determined by the caller and the old identifier. Readers interested in the mathematical details of *Identifier* and *newid* may consult the appendix to this chapter (Section 4.8).

The actual syntactic set *State* is a subset of that generated by the grammar in Figure 4.1, for we also require that a state be *name-consistent.* The semantics given in the next section give a "letrec" interpretation to the syntactic sets *Block* and *State*; name-consistency is simply a group of restrictions on identifiers to make sure that scoping rules are properly obeyed and preserved by the semantics. A state is name-consistent if it satisfies these three conditions:

1. All identifiers are defined in scope, that is, any identifier appearing on the right hand side of a binding must also appear on the left hand side of some binding in an enclosing block or in the state.
2. All identifiers appearing on left hand sides are pairwise distinct. We really mean *all* left hand side identifiers: a left hand side identifier, no matter how deeply nested in blocks it appears, must be distinct from every other left hand side identifier, whether in the same block, an enclosing block, an enclosed block, or a non-overlapping block.

3. If a state or block contains a binding of the form

$$X = \textit{Primary Identifier};$$

then no other left hand side in that state or block, nor in enclosed blocks, contains an identifier that is $\mathit{newid}(X,Y)$ for any Y.

Condition (1) above requires that any identifier used in an expression have an associated definition, while conditions (2) and (3) make sure that there are no multiple definitions. (The motive for condition (3) will become apparent when we discuss the semantics of function calls in the next section.) For the sake of simplicity, the latter two conditions are actually a bit more stringent than is necessary to make everything work out; this is inconsequential since the conditions only affect the choice of identifiers, which in any event is arbitrary as far as the meaning of the program is concerned.

To illustrate functional quads, here is a definition for factorial, expressed in Section 1.1's kernel language and in functional quads:

Kernel Language

```
fact x =
  if x <= 0 then
    1
  else
    x * (fact (x -1));
```

Functional Quads

```
fact^{(1)} x^{(0)} =
  {p^{(0)} = x^{(0)} <= 0;
   res^{(0)} =
     if p^{(0)} then
       {in 1}
     else
       {xx^{(0)} = x^{(0)} - 1;
        fxx^{(0)} = (fact^{(1)}) xx^{(0)};
        xfxx^{(0)} = x^{(0)} * fxx^{(0)};
        in
          xfxx^{(0)}};
   in
     res^{(0)}};
```

A functional quads program to compute the factorial of five, therefore, would be:

$$\Diamond^{(0)} = \mathtt{fact}^{(1)}\ 5;\ \mathtt{fact}^{(1)}\ \mathtt{x}^{(0)} = \{\ldots\};$$

In general, we can easily translate a kernel program into function quads by flattening blocks where they do not belong, introducing new ones where they are required, and introducing new identifiers for unnamed subexpressions; we discuss this in more detail in Section 4.3.

Since the arity of an identifier can be inferred by looking at the binding which defines it, we will henceforth omit most arity superscripts. The origin of the name "functional quads" should be clear: the restricted syntax for expressions gives programs an appearance akin to the sequential quads notation used to describe sequential object code, where each line describes a single computation. It should be emphasized, however, that unlike sequential quads there is no significance to the *order* of bindings in functional quads; we discuss this further in Section 4.4.

4.2 Semantics

We give the semantics of functional quads operationally, as an abstract reduction system [37, 49].

Definition 4.1 *An* abstract rewriting system (ARS) *is a structure* $\langle \Sigma, \vdash \rangle$ *consisting of a set* Σ *and a binary relation* $\vdash$ *on* Σ.

For our purposes, Σ is the set of all name-consistent states and $\vdash$ is the *one-step reduction* relation, to be defined below. The idea is that $a \vdash b$ if b is the state obtained from performing one step of evaluation on a. We will define $\vdash$ in such a way that "one step" of evaluation will be the application of a primitive, or the selection of a conditional, or the application of a function, or the substitution of a value.

We define $\vdash$ through *rewrite rules*, which concisely describe the pairs of states such that $a \vdash b$. For example, there is the following rewrite rule:

$$X = Y;\ Y = V \implies X = V;\ Y = V$$

Each binding on the left hand side of the rule is to be matched against a separate binding of a state a. If such a match is found, then $a \vdash b$ where b is the state constructed by replacing the matched bindings of a with the bindings given by the right hand side of the rule. The bindings which match the left hand side need not appear in the same order as in the rule, nor need they be consecutive, and the bindings which replace them may be added to the state in any order and at any position. For example, the rule above implies that among other pairs, $\vdash$ holds for the following pairs of states:

$$\Diamond\ \texttt{= i; j = i + 5; i = 3;} \quad \vdash \quad \Diamond\ \texttt{= 3; i = 3; j = i + 5;}$$

Here we have matched $\Diamond$ with X, i with Y, and 3 with V.

Below we present the complete set of rewrite rules, with explanations. Throughout, V denotes any value, $X^{(0)}$, $Y^{(0)}$, and $Z^{(0)}$ any identifier of arity 0 (omitting the superscript when apparent from context), $F^{(n)}$ any identifier of arity $n > 0$, P any primary, and B any binding.

$$X = Y;\ Y = V;\ \Longrightarrow\ X = V;\ Y = V; \tag{R1a}$$

$$X = Y\ Op\ P;\ Y = V;\ \Longrightarrow\ X = V\ Op\ P;\ Y = V; \tag{R1b}$$

$$X = P\ Op\ Y;\ Y = V;\ \Longrightarrow\ X = P\ Op\ V;\ Y = V; \tag{R1c}$$

$$\begin{array}{l} X = \texttt{if}\ Y\ \texttt{then \{...\} else \{...\};} \\ Y = V; \end{array} \Longrightarrow \begin{array}{l} X = \texttt{if}\ V\ \texttt{then \{...\} else \{...\};} \\ Y = V; \end{array} \tag{R1d}$$

$$X = \texttt{sel\_}t\texttt{\_}i\ Y;\ Y = V;\ \Longrightarrow\ X = \texttt{sel\_}t\texttt{\_}i\ V;\ Y = V; \tag{R1e}$$

$$X = \texttt{is\_}t\texttt{?}\ \ Y;\ Y = V;\ \Longrightarrow\ X = \texttt{is\_}t\texttt{?}\ \ V;\ Y = V; \tag{R1f}$$

$$X = Y\ Z;\ Y = V\ \Longrightarrow\ X = V\ Z;\ Y = V \tag{R1g}$$

Collectively, these rules allow an identifier to be substituted by the value to which it is bound, for all contexts in which the grammar allows a *Primary* in an expression. Because only values are substituted, the computation which reduces an identifier to a value is shared among all references to that identifier. The choice of when *Primary* appears in the grammar as opposed to *Identifier*$^{(0)}$, and therefore the choice of substitution rules, is carefully based on semantic grounds, as we discuss in Section 4.4.

$$X = \texttt{const}\ V\ \Longrightarrow\ X = V \tag{R2}$$

The **const** statement and this rewrite rule do not give functional quads any additional expressive or computational power, but are included as a technical convenience for the benefit of the material in Chapter 5.

$$X = \texttt{if true then \{}B_{t,1}\texttt{;...;}B_{t,n}\ \texttt{in}\ Y_t\texttt{\} else \{...\};}\ \Longrightarrow\ \begin{array}{c} X = Y_t; \\ B_{t,1};\ldots;B_{t,n}; \end{array} \tag{R3}$$

There is also an analogous rule for **if false** .... After execution of this rule, the selected arm becomes part of the state, and so its bindings become subject to execution. The identifiers bound in the new bindings added to the state cannot conflict with bindings already there, because of the pairwise distinctness aspect of name-consistency.

$$X = V_1 + V_2;\ \Longrightarrow\ X = V_3; \tag{R4}$$

where $V_3 = V_1 + V_2$. There are similar rules for -, *, >=, *etc.*

$$X = \texttt{sel\_}t\texttt{\_}i\ \texttt{<}t\texttt{,}Y_1\texttt{,...,}Y_i\texttt{,...,}Y_n\texttt{>;}\ \Longrightarrow\ X = Y_i; \tag{R5}$$

$$X = \texttt{is\_}t\texttt{?}\ \ \texttt{<}t\texttt{,}Y_1\texttt{,...,}Y_n\texttt{>;}\ \Longrightarrow\ X = \texttt{true;} \tag{R6a}$$

$$X \texttt{ = is\_}t\texttt{? } V\texttt{;} \quad \Longrightarrow \quad X \texttt{ = false;} \tag{R6b}$$

for any V which is not a *Struct* with tag t.

$$X \texttt{ = (}F^{(n)}\ Y_1\ \ldots\ Y_{i-1}\texttt{)}\ Y_i\texttt{;} \quad \Longrightarrow \quad X \texttt{ = (}F^{(n)}\ Y_1\ \ldots\ Y_{i-1}\ Y_i\texttt{);} \tag{R7}$$

where $1 \leq i < n$.

$$\begin{array}{l} X \texttt{ = (}F^{(n)}\ Z_1\ \ldots\ Z_{n-1}\texttt{)}\ Z_n\texttt{;} \\ F^{(n)}\ Y_1\ \ldots\ Y_n \texttt{ = \{}B_{F1}\texttt{;}\ldots\texttt{;}B_{F_m}\texttt{ in } Z_F\texttt{\};} \end{array} \quad \Longrightarrow \quad \begin{array}{l} X \texttt{ = } Z'_F\texttt{;} \\ F^{(n)}\ Y_1\ \ldots\ Y_n \texttt{ = \{}\ldots\texttt{\};} \\ B'_{F1}\texttt{;}\ldots B'_{Fm}\texttt{;} \\ Y'_1 \texttt{ = } Z_1\texttt{;}\ldots\texttt{;}Y'_n \texttt{ = } Z_n\texttt{;} \end{array} \tag{R8}$$

where the primes indicate consistent α-renaming of all identifiers appearing on left hand sides within the body of $F^{(n)}$, together with the formals, such that they are given unique names not appearing anywhere else in the state. (By "all identifiers appearing on left hand sides" we are including formals of internal definitions and the binding lists of all enclosed blocks, so that the only identifiers unaffected by the renaming are free variables of the function F.) The choice of α-renaming is not arbitrary: for each identifier $Y^{(i)}$ that is to be renamed, it is renamed to $newid(X,Y^{(i)})$ (where X is the same X as in the statement of the rule). In this way, we preserve a connection between the caller and the new computation added to the state. (We point out that condition (3) of name-consistency (page 51) insures that the new bindings added to the state by this rule do not conflict with any already present.)

As an exercise, we invite the reader to verify that name-consistency is preserved by all of the rules above.

All of these rules have the effect of replacing all or part of an expression which occurs on the right hand side of a binding. In this way, this abstract reduction system bears a strong resemblence to a term rewriting system, except that while the replacements in a term rewriting system are context-free, in this system a replacement depends upon other components of the state. In a term rewriting system, the subexpression that is replaced is called a *redex*, and we will adopt that terminology to refer to the portion of the state which changes; in Rules R1a through R1g, the redex is the occurence of Y on the right hand side of the binding for X, while in Rules R2 through R8 the redex is the entire right hand side of the binding for X. We will sometimes use the notation $x \vdash_\alpha y$ to indicate that a particular redex α within x is rewritten to arrive at state y (a notion we will formalize in Section 4.5).

Executing a program is modeled by the abstract reduction system as successive application of rewrite rules to an initial state until no more rewriting is possible. Here is an example, in which the selected redex is underlined at each step (note that this is just one of many possible reduction sequences from this initial state):

```
    ◇ = sel_cons_1 b; b = (f) a; a = 3 + 4; f x = {y = <cons,x,y>; in y};
⊢
    ◇ = sel_cons_1 b; b = yy; a = 3 + 4; f x = {...}; yy = <cons,xx,yy>; xx = a;
⊢
    ◇ = sel_cons_1 b; b = <cons,xx,yy>; a = 3 + 4; f x = {...}; yy = <cons,xx,yy>; ...
⊢
    ◇ = sel_cons_1 <cons,xx,yy>; b = <cons,xx,yy>; a = 3 + 4; f x = {...}; ...
⊢
    ◇ = xx; b = <cons,xx,yy>; a = 3 + 4; f x = {...}; yy = <cons,xx,yy>; xx = a;
⊢
    ◇ = xx; b = <cons,xx,yy>; a = 7; f x = {...}; yy = <cons,xx,yy>; xx = a;
⊢
    ◇ = xx; b = <cons,xx,yy>; a = 7; f x = {...}; yy = <cons,xx,yy>; xx = 7;
⊢
    ◇ = 7; b = <cons,xx,yy>; a = 7; f x = {...}; yy = <cons,xx,yy>; xx = 7;
```

Here we have implicitly assumed that *newid*(b, x) = **xx** and *newid*(b, y) = **yy**. Notice that the call to procedure **f** is executed before the argument **a** is reduced to a value, illustrating the use of identifiers in functional quads models non-strictness. Data structures are non-strict because they are considered values even though they contain identifiers whose values have not yet been computed. Similarly, rules R7 and R8 make functions non-strict because they apply even when the arguments to functions are non-value identifiers.

In saying that the elements of Σ in the ARS $\langle\Sigma, \vdash\rangle$ are states, we are actually glossing over a minor technical point. In our description of rewrite rules, we noted that the order in which bindings appeared in the state is immaterial, and similarly there was complete freedom in how new bindings were added to the state. If a state a has a different ordering of its bindings but is otherwise syntactically equal to another state b, then a and b are indistinguishable as far as the reduction relation $\vdash$ is concerned, and we should really consider them to be identically the same. To be extremely precise, then, we should say that each element of Σ is not a state but an equivalence class of states which are syntactically equal but for permutation of the state bindings, and when we write something like

```
◇ = i; j = i + 5; i = 3;  ⊢  ◇ = 3; i = 3; j = i + 5;
```

it is tacitly understood that the states *on either side of the* $\vdash$ symbol are just representatives from their respective equivalence classes. This said, we shall hunceforth disregard it entirely, and simply ignore the order in which state bindings happen to be written.[1] Many readers will recognize that a similar equivalence class argument crops up in lambda calculus, where

[1] We could, if we liked, extend this equivalence over permutation to the bindings of other blocks contained within a state, but there is no reason to for the purposes of our theory.

$\lambda a.a$ and $\lambda b.b$ are considered the *same* lambda expression even though they are syntactically different (see [11], Appendix C). On the other hand, in functional quads we shall *not* use this sort of equivalence class argument to deal with the issue of α-renaming, hence the states `◇ = a; a = 5;` and `◇ = b; b = 5;` are *different* states. Our use of *newid* obviates the need for such equivalence.

4.3 From a Functional Language to Functional Quads

We have given functional quads a syntax similar to that of the kernel language we introduced in Section 1.1, and so it may appear that the relationship between the two is no more deep than concrete syntax. In fact, the conversion from a functional language to functional quads should be considered a process of compilation: a functional language and functional quads each have their own well-defined semantics, and so the translation from one to the other is a matter of achieving in the functional quads semantics the meaning of the original functional language program. This translation may be straightforward or not, depending on how closely the two semantics match.

In describing the kernel langauge in Section 1.1 and in using it in examples (Chapter 3), we intentionally assumed a semantics that mirrors the semantics of functional quads. Translating from that particular kernel language to functional quads, therefore, is mostly a matter of introducing identifiers and blocks to conform to the restricted syntax of functional quads. The following translation schema $\mathcal{Q}$ translates a kernel language definition into a functional quads definition by adding identifiers. We assume that the kernel program has already been α-renamed so that each identifier is uniquely defined, and we use T to denote a new unique identifier and the symbol § to denote concatenation of bindings:

$$\mathcal{Q}\llbracket F\ Y_1\ \ldots\ Y_n\ \texttt{=}\ E\rrbracket \;=\; F\ Y_1\ \ldots\ Y_n\ \texttt{=}\ \texttt{\{}\mathcal{Q}\llbracket T_1\ \texttt{=}\ E\rrbracket\ \texttt{in}\ T_1\texttt{\}}$$

$$\begin{aligned}
\mathcal{Q}\llbracket X\ \texttt{=}\ C\rrbracket &= X\ \texttt{=}\ C\\
\mathcal{Q}\llbracket X\ \texttt{=}\ F\rrbracket &= X\ \texttt{=}\ (F^{(n)})\\
\mathcal{Q}\llbracket X\ \texttt{=}\ Y\rrbracket &= X\ \texttt{=}\ Y^{(0)}\\
\mathcal{Q}\llbracket X\ \texttt{=}\ E_1\ Op\ E_2\rrbracket &= \mathcal{Q}\llbracket T_1\ \texttt{=}\ E_1\rrbracket\,\S\,\mathcal{Q}\llbracket T_2\ \texttt{=}\ E_2\rrbracket\,\S\,X\ \texttt{=}\ T_1\ Op\ T_2\\
\mathcal{Q}\llbracket X\ \texttt{=}\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\rrbracket &= \mathcal{Q}\llbracket T_1\ \texttt{=}\ E_1\rrbracket\S\\
&\ X\ \texttt{=}\ \texttt{if}\ T_1\ \texttt{then}\ \texttt{\{}\mathcal{Q}\llbracket T_2\ \texttt{=}\ E_2\rrbracket\ \texttt{in}\ T_2\texttt{\}}\\
&\ \qquad\qquad\quad \texttt{else}\ \texttt{\{}\mathcal{Q}\llbracket T_3\ \texttt{=}\ E_3\rrbracket\ \texttt{in}\ T_3\texttt{\}}\\
\mathcal{Q}\llbracket X\ \texttt{=}\ \texttt{make\_}t\ E_1\ \ldots\ E_n\rrbracket &= \mathcal{Q}\llbracket T_1\ \texttt{=}\ E_1\rrbracket\,\S\cdots\S\,\mathcal{Q}\llbracket T_n\ \texttt{=}\ E_n\rrbracket\,\S\,X\ \texttt{=}\ \texttt{<}t\texttt{,}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{>}
\end{aligned}$$

$$
\begin{aligned}
Q\llbracket X \mathtt{=} \mathtt{sel\_}t\mathtt{\_}i\ E\rrbracket &= Q\llbracket T_1 \mathtt{=} E\rrbracket \,\S\, X \mathtt{=} \mathtt{sel\_}t\mathtt{\_}i\ T_1 \\
Q\llbracket X \mathtt{=} \mathtt{is\_}t\mathtt{?}\ E\rrbracket &= Q\llbracket T_1 \mathtt{=} E\rrbracket \,\S\, X \mathtt{=} \mathtt{is\_}t\mathtt{?}\ T_1 \\
Q\llbracket X \mathtt{=} E_1\ E_2\rrbracket &= Q\llbracket T_1 \mathtt{=} E_1\rrbracket \,\S\, Q\llbracket T_2 \mathtt{=} E_2\rrbracket \,\S\, X \mathtt{=} T_1\ T_2 \\
Q\llbracket X \mathtt{=} \mathtt{\{}B_1\mathtt{;}\ldots\mathtt{;}B_n\ \mathtt{in}\ E\mathtt{\}}\rrbracket &= Q\llbracket B_1\rrbracket \,\S \cdots \S\, Q\llbracket B_n\rrbracket \,\S\, Q\llbracket X \mathtt{=} E\rrbracket
\end{aligned}
$$

This is by no means the best translation possible. For example, extra identifiers are introduced if the original program contains a binding like a = b + c; this is easily corrected by adding some more rules to the translation. There are, however, some other choices facing the designer of a translation that are much less trivial in nature.

Some choices amount to source-to-source optimizations. A good example is common subexpression elimination: if two identifiers in the same block are bound to the same expression, one can be eliminated (and the references to it suitably renamed). Another important optimization is "fetch elimination" [71], which bypasses a fetch from a data structure when the value stored there can be identified. An example:

```
...                            ...
a = <tuple,x,y>;         →     a = <tuple,x,y>;
b = sel_tuple_1 a;             b = x;
...                            ...
```

This optimization is particularly important when tuples are used as a substitute for multiple values. For a discussion of other traditional compiler optimizations in a functional setting, see [71] and [64].

Other choices in the translation to functional quads try to match the functional quads program to the capabilities and/or requirements of the compilation phases which follow. For example, an implementation may be able to compile better procedure calls when a known function is applied to all arguments. Thus, we would want to add a statement:

$$
\begin{aligned}
Q\llbracket X \mathtt{=} F\rrbracket &= X \mathtt{=} \mathtt{(}F^{(n)}\mathtt{)} \\
Q\llbracket X \mathtt{=} F\ E_1\ \ldots\ E_n\rrbracket &= Q\llbracket T_1 \mathtt{=} E_1\rrbracket \,\S \cdots \S\, Q\llbracket T_n \mathtt{=} E_n\rrbracket \,\S\, X \mathtt{=} \mathtt{(}F^{(n)}\ T_1\ \ldots T_{n-1}\mathtt{)}\ T_n
\end{aligned}
$$

so that a single first-order application statement is generated instead of n partial application statements. Another implementation-dependent transformation is the promotion of internal definitions to top-level via *lambda lifting* [42]; this may be required if an implementation has no primitive way of dealing with internal definitions.

Still other choices affect the sharing and/or scheduling of computation. It is possible, for instance, to achieve greater sharing in some situations by performing *mfe abstraction* [38].

Another option involves the scheduling of computation in conditionals. The translation schema above enforces the property that code within an arm of a conditional is not evaluated until the predicate becomes a value. If "eager" conditionals are desired, however, an alternate translation of conditionals can be used:

$$\begin{aligned} \mathcal{Q}[X \;\texttt{=}\; \texttt{if}\; E_1 \;\texttt{then}\; E_2 \;\texttt{else}\; E_3] \;=\; & \mathcal{Q}[T_1 \;\texttt{=}\; E_1]\,\S\,\mathcal{Q}[T_2 \;\texttt{=}\; E_2]\,\S\,\mathcal{Q}[T_3 \;\texttt{=}\; E_3]\S \\ & X \;\texttt{=}\; \texttt{if}\; T_1 \;\texttt{then}\; \texttt{\{in}\; T_2\texttt{\}}\; \texttt{else}\; \texttt{\{in}\; T_3\texttt{\}} \end{aligned}$$

To summarize, functional quads has a particular semantics, which the translation from the functional source language must take into account. All of the decisions related to assigning meaning to source language constructs are encoded into the functional quads for a program, and so our task in describing the generation of sequential code is limited to faithfully implementing the operational semantics given for functional quads.

4.4 Functional Quads vs. Sequential Quads

Functional quads has been carefully designed so that its computation steps correspond closely with computation steps in the familiar sequential quads which model sequential object code. In the previous section we saw how functional languages can be translated into functional quads, and that this is fairly straightforward if the original language's semantics are similar to that of functional quads (most importantly, if the original language is non-strict). The translation from sequential quads into target code is well-known von Neumann compiler technology, involving register allocation, procedure linkage optimization, *etc.* The only difference between functional and sequential quads is that in functional quads the ordering of subexpressions and delaying of computation is implicit, but in sequential quads it is explicit. As we discussed in Chapter 3, the heart of functional langauge compilation lies precisely in determining the ordering of subexpression evaluation and in deciding what to delay, so the meat of a functional language compiler is concentrated into the translation from functional quads to sequential quads.

For example, here is a functional quads fragment:

```
...; x = 2 * 3; z = x + y; y = 4 * 5; ...
```

Each binding can be directly translated into a sequential quads statement, but in addition we must choose their relative ordering. One possible translation is this fragment:

```
...
y := 4 * 5
x := 2 * 3
z := x + y
...
```

Another possible translation would interchange the first two statements. Correctness of a functional quads to sequential quads translation requires the statement ordering chosen to be consistent with a possible reduction sequence of the original functional quads program. The sequence above, for example, is consistent with the following reduction sequence:

$$\texttt{...; x = 2 * 3; z = x + y; y = }\underline{\texttt{4 * 5}}\texttt{; ...}$$

$$\vdash$$

$$\texttt{...; x = }\underline{\texttt{2 * 3}}\texttt{; z = x + y; y = 20; ...}$$

$$\vdash$$

$$\texttt{...; x = 6; z = }\underline{\texttt{x}}\texttt{ + y; y = 20; ...}$$

$$\vdash$$

$$\texttt{...; x = 6; z = 6 + }\underline{\texttt{y}}\texttt{; y = 20; ...}$$

$$\vdash$$

$$\texttt{...; x = 6; z = }\underline{\texttt{6 + 20}}\texttt{; y = 20; ...}$$

$$\vdash$$

$$\texttt{...; x = 6; z = 26; y = 20; ...}$$

There is a one-to-one correspondence between expressions in a functional quads program and its sequential quads counterpart, and also between variables. Furthermore, executing a sequential quads statement is modeled in functional quads as the reduction of some substitution rules (R1a through R1g) followed by a computational rule (R2 through R8). Above, for example, the sequential quads statement `z := x + y` was modeled by the third, fourth, and fifth reductions in the functional quads sequence. We can examine the sequential quads statement at a finer level, however, and note that it is really composed of two operand fetches and an addition—and each of these has an exact counterpart in functional quads. This explains why substitution was limited to the seven cases covered by rules R1a through R1g: they cover the cases where the corresponding sequential quads must fetch from the corresponding variable. If other substitution rules were added to functional quads, they would not correspond to any movement of data in the corresponding sequential quads program, and could lead to infinite sequences of such vacuous reductions. For example, if a substitution rule were added which allowed substitution for a variable appearing within a data structure, a cyclic structure could lead to the following infinite reduction sequence:

$$\texttt{a = <cons,a,b>} \overset{?}{\vdash} \texttt{a = <cons,<cons,a,b>,b>} \overset{?}{\vdash} \texttt{a = <cons,<cons,<cons,a,b>,b>,b>} \overset{?}{\vdash} \ldots$$

This reduction sequence does not correspond to any sequence of computations that would take place in an actual implementation.

The technique of translating functional quads into sequential quads is taken up in the four chapters that follow. In the remainder of this chapter, we establish some mathematical properties of the functional quads reduction system, which form the foundation of our compilation method.

4.5 Mathematical Properties of Reduction

We now formalize some of the concepts we introduced in Section 4.2. The following two definitions are standard [49]:

Definition 4.2 *A* normal form *of an abstract reduction system* $\langle\Sigma,\vdash\rangle$ *is an* $a \in \Sigma$ *such that there is no* $b \in \Sigma$ *for which* $a \vdash b$.

Definition 4.3 *A* reduction sequence *in an ARS* $\langle\Sigma,\vdash\rangle$ *is a sequence of elements from* Σ $a_0, a_1, \ldots$ *such that* $a_i \vdash a_{i+1}$, $i \geq 0$. *A* terminating reduction sequence of length n *is a finite reduction sequence* $a_0, \ldots, a_n$ *such that* a_n *is a normal form.*

We also wish to formalize the notion of identifying a particular redex within a state.

Definition 4.4 *A* redex specifier *for functional quads is a pair* $\alpha = \langle X, \rho\rangle$ *where* X *is an identifier of arity zero and* ρ *is the name of a rewrite rule (an element of the set* $\{R1a, R1b, \ldots, R8\}$*). A redex* $\alpha = \langle X, \rho\rangle$ exists *in a state* S_0 *if* S_0 *contains a binding of the form* X = E; *which matches rewrite rule* ρ *(perhaps together with other bindings from* S_0*, depending on the rule). If a redex* α *exists in a state* S_0*, then* $S_0 \vdash_\alpha S_1$ *where* S_1 *is the state obtained from applying the specified rule to the specified redex.*

A redex specifier can denote at most one redex within a given state because name-consistency guarantees the uniqueness of identifiers bound in a state.

All of the theory we will develop in the framework of functional quads depends on the following three properties of the functional quads reduction system (the names of these properties are not standard):

Property 4.5 (Commutivity) *Let* α *and* β *be distinct redexes in* a*, and let* b *and* c *be such that* $a \vdash_\alpha b$ *and* $a \vdash_\beta c$. *Then there exists* d *such that* $b \vdash_\beta d$ *and* $c \vdash_\alpha d$.

This property says that given two redexes, rewriting them consecutively yields equivalent results regardless of order. This very strong property implies, among other things, confluence (the Church-Rosser property) and that all reduction sequences to normal form are of equal length.[2]

Property 4.6 (Redex Uniqueness) *Let α be a redex in a. If $a \vdash_\alpha b$, then there is no c such that $b \vdash^* c$ and α exists in c.*

In other words, the same redex (where "same" means having the same reduction specifier α) cannot be reduced twice. This implies that the redex specifiers defined above are adequate to identify a particular step in a reduction sequence.

Property 4.7 (Strong Dependence) *For all a, b, c, d such that $a \vdash^* b \vdash_\alpha c \vdash_\beta d$, if β does not exist in b then α is reduced before β in every reduction sequence that begins with a and includes β.*

This says that if the reduction of a redex α introduces a new redex β, then reduction of α is a necessary precondition for the creation of β.

As we mentioned, the commutivity property implies confluence, which in turn implies that normal forms are unique, an important property indeed if functional quads is to be a model of computation. Adding the redex uniqueness property will allow us to show that in every reduction sequence of a given program to normal form (indeed, to any arbitrary derivable form) the same set of redexes are reduced. We will not make use of the strong dependence property until the next chapter, where it will allow us to infer sufficient constraints upon the order in which redexes are reduced from a consideration of all possible orderings.

In the remainder of this section we prove that functional quads has each of the three properties above, and also prove some results that follow from the properties.

Theorem 4.8 (Figure 4.2a) *Functional quads has the Commutivity property.*

Proof. Let $\alpha = \langle X_\alpha, \rho_\alpha \rangle$ and $\beta = \langle X_\beta, \rho_\beta \rangle$. We consider two cases depending on whether $X_\alpha = X_\beta$:

Case 1 ($X_\alpha \neq X_\beta$). If $a \vdash_\alpha b$, the only way b differs from a is that new bindings may have been added and that the right hand side of X_α's binding will have changed. Now the redex β is completely unaffected by this, since the binding for X_β is unchanged, as are any other bindings

[2] A weaker version of this property which does not identify the redexes α and β is called the *Diamond Property* [11], or *WCR*^{1} [49].

$$\begin{tikzcd}
a \arrow[r, "\beta"] \arrow[d, "\alpha"'] & c \arrow[d, dashed, "\alpha"] \\
b \arrow[r, dashed, "\beta"'] & d
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
a \arrow[r, "*"] \arrow[d, "*"'] & c \arrow[d, dashed, "*"] \\
b \arrow[r, dashed, "*"'] & d
\end{tikzcd}$$

(b)

Figure 4.2: Diagrams of (a) Commutivity; (b) Confluence

involved in reducing β (such as $Y = V$; or $F\ Y_1\ \ldots\ Y_n = \{\ldots\}$;, which cannot contain redexes and are therefore not affected by α). So β exists in b, and reducing it in b makes the same changes to the state as reducing it in a. By symmetry, α exists in c (where $a \vdash_\beta c$), and reducing it in c makes the same changes as reducing it in a. The net changes made to the state by reducing both α and β are independent of the order in which they are performed, and so $b \vdash_\beta d$ and $c \vdash_\alpha d$.

Case 2 $(X_\alpha = X_\beta)$ In this case, it must be that $\rho_\alpha = \text{R1b}$ and $\rho_\beta = \text{R1c}$ (or vice versa), and the same argument about non-interference in Case 1 applies. ∎

Another way of looking at it is that reducing a redex never duplicates an existing redex, nor does it remove any redex other than the one reduced. We are also depending on the way new identifiers are introduced by Rule R8; the use of *newid* insures that they are created in a way which does not depend on the relative order in which various R8 redexes are reduced.

Corollary 4.9 (Confluence (Figure 4.2b))

$$\forall a,b,c \quad a \vdash^* b \wedge a \vdash^* c \Rightarrow \exists d\, b \vdash^* d \wedge c \vdash^* d$$

Proof. Standard [49] (see also the proof of Lemma 4.12). ∎

Corollary 4.10 *Normal forms in functional quads are unique.*

Theorem 4.11 *Functional quads has the Redex Uniqueness property.*

Proof. Figure 4.3 enumerates all the syntactic possibilities for a binding of the form $X = E$;, and shows which rules can transform one into the other. On no path in this diagram does the

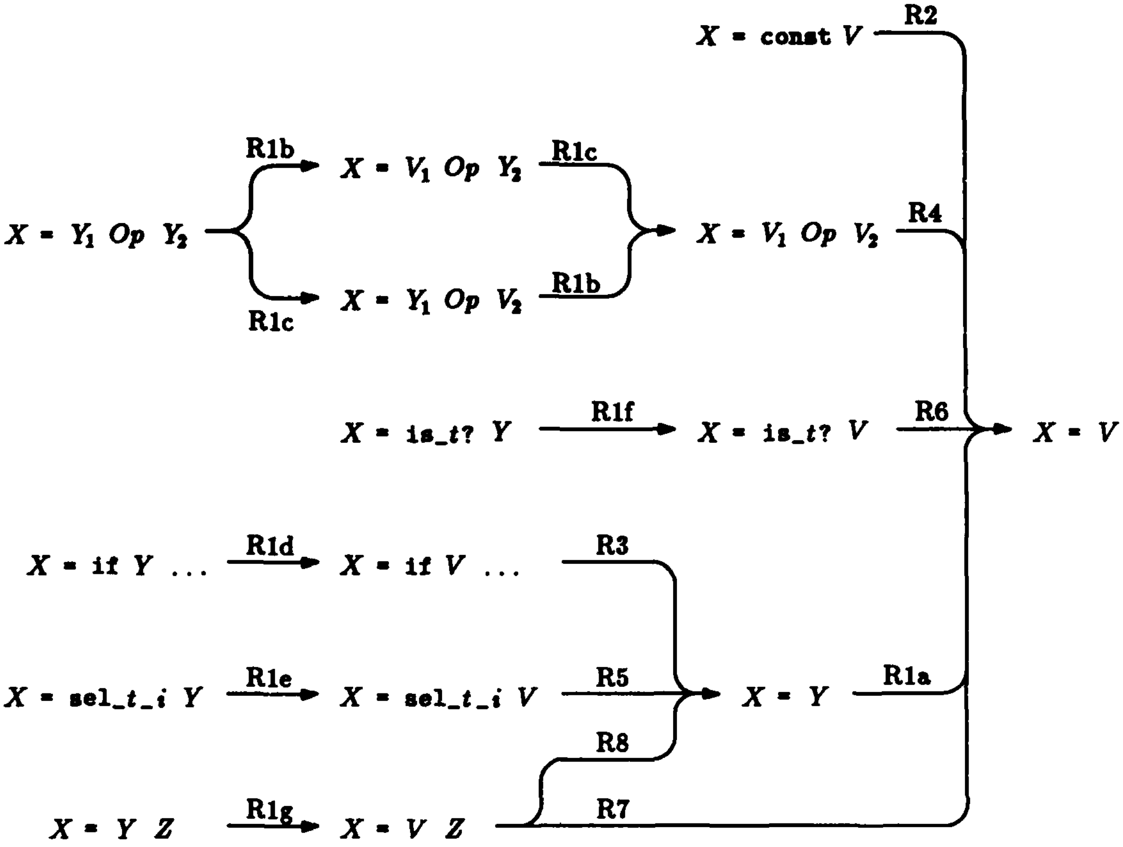


Figure 4.3: Summary of Possible Rewritings of Bindings

same rule appear twice, so if $\alpha_1 = \langle X, \rho_1 \rangle$ and $\alpha_2 = \langle X, \rho_2 \rangle$ are two redexes reduced in some reduction sequence, $\rho_1 \neq \rho_2$. ∎

The following lemma strengthens Corollary 4.9 (confluence) by identifying the redexes needed to unite two states.

Lemma 4.12 *Let $a \vdash_{\alpha_1} \cdots \vdash_{\alpha_n} b$ and $a \vdash_{\beta_1} \cdots \vdash_{\beta_m} c$ be two finite reduction sequences. Then there exists d such that $b \vdash_{\delta_1} \cdots \vdash_{\delta_l} d$ and $c \vdash_{\gamma_1} \cdots \vdash_{\gamma_k} d$, and furthermore the following hold:*

$$\begin{aligned}(\gamma_1, \ldots, \gamma_k) &= (\alpha_1, \ldots, \alpha_n) - \{\beta_1, \ldots, \beta_m\} \\ (\delta_1, \ldots, \delta_l) &= (\beta_1, \ldots, \beta_m) - \{\alpha_1, \ldots, \alpha_n\}\end{aligned}$$

where the notation $(x_1, \ldots, x_n) - \{y_1, \ldots, y_m\}$ indicates the sequence obtained by removing elements of the set $\{y_1, \ldots, y_m\}$ from the sequence $(x_1, \ldots, x_n)$, maintaining the same order between the x's that remain.

Proof. We construct d by using Theorem 4.8 (commutivity) to "tile" the reduction diagram, as shown in Figure 4.4a. Consider the uppermost row of tiles. There are two cases depending on

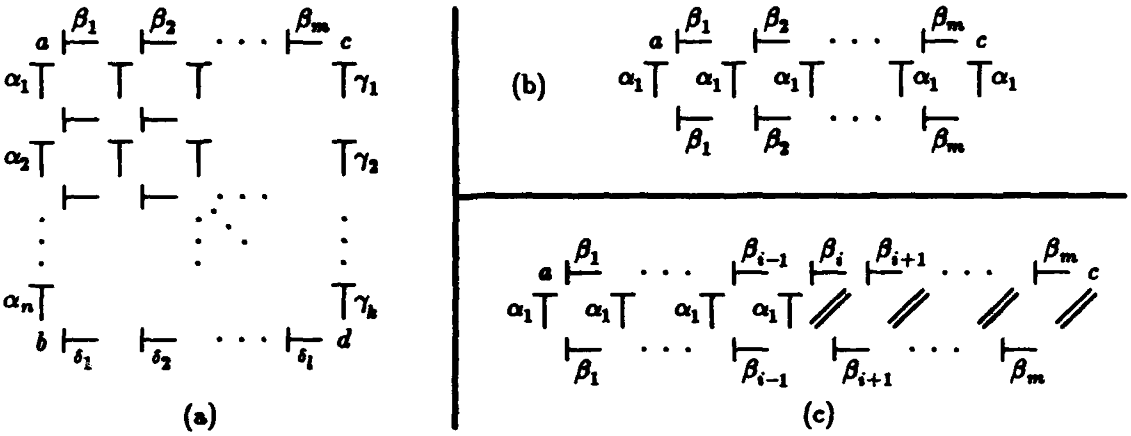


Figure 4.4: Proof of Lemma 4.12

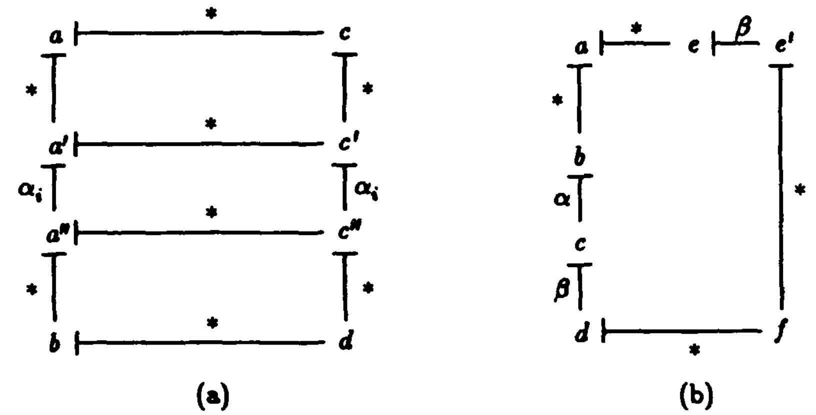


Figure 4.5: Proof of Theorems 4.13 and 4.15

whether $\alpha_1 = \beta_i$ for some i. If not, then $\gamma_1 = \alpha_1$ (Figure 4.4b). If so, then there is no γ reduction in the first row (Figure 4.4c). Continuing this argument for the remaining $n-1$ rows gives the γ-sequence result (note that the bottom edge of each row will contain all β's which are not the same as any α already executed; Theorem 4.11 guarantees that none of the α's which follow will be the same as a β that is missing from the bottom edge). A symmetric argument gives the δ-sequence result. ∎

Theorem 4.13 *Let $a \vdash_{\alpha_1} \cdots \vdash_{\alpha_n} b$ be a finite reduction sequence. Then all reduction sequences from a to b are permutations of $(\alpha_1, \ldots, \alpha_n)$.*

Proof. Suppose the theorem were false; then there is a reduction sequence $a \vdash_{\beta_1} \cdots \vdash_{\beta_m} c$ where $b = c$ and either $\{\alpha_1, \ldots, \alpha_n\} - \{\beta_1, \ldots, \beta_m\} \neq \emptyset$ or $\{\beta_1, \ldots, \beta_m\} - \{\alpha_1, \ldots, \alpha_n\} \neq \emptyset$. Consider the former; there is some α_i which is not equal to any β. We can use the tiling argument from Lemma 4.12 to construct the diagram in Figure 4.5a. Since α_i is absent from the paths $a \vdash^* c$ and $a \vdash^* a'$, by the lemma it is absent from the paths $a' \vdash^* c'$ and $c \vdash^* c'$. Hence the reduction from c' to c'' must be of α_i, and so we have $c \vdash^* \vdash_{\alpha_i} \vdash^* d$. But $b = c$, so we also have $a \vdash^* \vdash_{\alpha_i} \vdash^* b \vdash^* \vdash_{\alpha_i} \vdash^* d$, violating the redex uniqueness theorem. Contradiction. A symmetric argument handles the case where there is some β_i not equal to any α. ∎

One of the consequences of this theorem is that all reduction sequences from a given state to normal form are of the same length, and so whether normal form is reached is independent of reduction strategy. We discuss this further in the next section.

We will also find the converse of the previous theorem useful:

Theorem 4.14 *Let $a \vdash_{\alpha_1} \cdots \vdash_{\alpha_n} b$ be a finite reduction sequence and let $a \vdash_{\beta_1} \cdots \vdash_{\beta_n} c$ be another reduction sequence where $(\beta_1, \ldots, \beta_n)$ is a permutation of $(\alpha_1, \ldots, \alpha_n)$. Then $b = c$.*

Proof. By Lemma 4.12 there exists d such that $b \vdash^* d$ and $c \vdash^* d$, but by the lemma both of these sequences must be empty. ∎

These two theorems allow us to sensibly speak of the set of redexes reduced in reducing a state S_0 to another state S_1; we will denote this set by $A^{S_0 \vdash^* S_1}$.

Theorem 4.15 *Functional quads has the Strong Dependence property.*

Proof. We are given $a \vdash^* b \vdash_\alpha c \vdash_\beta d$ where β does not exist in b; so performing the reduction $b \vdash_\alpha c$ *creates* the redex β. There are a limited number of possible combinations for α and β; for example, if α is an R1b redex, then β must be an R4 redex. Suppose the strong dependence property does not hold; then there is a sequence $a \vdash^* e \vdash_\beta e'$ where the reduction from a to e does not include α. We can apply Lemma 4.12 to unite d and e', so that we have $d \vdash^* f$ and $e' \vdash^* f$, where α is reduced somewhere along $e' \vdash^* f$ (see Figure 4.5b). But in fact, since β was reduced in arriving at e', α cannot possibly exist in any successor of e'; continuing the example above, if β is an R4 redex then after reducing β the right hand side of β's binding is a value, and will never contain an R1b redex. Contradiction. (We leave the details of the other cases as an exercise for the reader.) ∎

This theorem does not say that α will always create β; for example, if β is an R4 redex then in general there is an R1b redex and an R1c redex that must precede it, and only the second of these will create β. What the theorem *does* say is that there are no disjunctive preconditions for a redex—a redex cannot be created by the reduction of *either* of two other redexes, for example. We will not make any use of the strong dependence property until the next chapter.

4.6 Termination, Weak Normal Forms, and Lazy Evaluation

We have seen that normal forms produced from a given program are always the same, regardless of the evaluation order used in their derivations. We now examine the question of whether the choice of evaluation order affects whether a normal form is reached at all. Answering this question will allow us to contrast lenient and lazy evaluation within the functional quads framework.

In fact, Theorem 4.13 showed that in functional quads, all derivations to a normal form are of equal length, and so evaluation order has no effect on whether normal form is reached. This property seems a bit surprising at first; we know that lazy evaluation of fuuctional programs terminates on programs where other evaluation orders fail, but it seems that in functional quads the evaluation order makes no difference. The problem is not with the functional quads reduction system, but simply in what we consider a normal form.

In functional quads, computations (redexes) are never removed from the state, even when their values do not contribute to the final answer. To illustrate:

```
◇ = (k 5) p; p = (loop) 3; loop v = {a = (loop) v in a}; k x y = {in x};
```

This program has no normal form, because each attempt to reduce (`loop` x) adds another call to `loop` to the state. On the other hand, it is possible to reduce this program to a state that contains a binding ◇ `= 5`. So we can obtain states that are not normal forms but nevertheless have an answer. We call these states *weak normal forms*.

Definition 4.16 *A* weak normal form *in functional quads is a state containing a binding* ◇ = V*, where* V *is any value.*

Unlike normal forms, weak normal forms are not necessarily unique. For example, the preceding program has the following two weak normal forms, among others:

```
◇ = 5; p = (loop) 3; xx = 5; yy = p; loop v = {...}; k x y = {in x};

◇ = 5; p = aa; vv = 3; xx = 5; yy = p; aa = (loop) vv; loop v = {...}; ...
```

While weak normal forms are not unique, we certainly expect the answers contained in weak normal forms to be unique. This is expressed in the following theorem:

Theorem 4.17 (Unique Values) *Let S_0, S_1, and S_2 be states such that $S_0 \vdash^* S_1$ and $S_0 \vdash^* S_2$, where S_1 contains a binding $X = V_1;$ and S_2 contains a binding $X = V_2;$ for some identifier X. Then $V_1 = V_2$ (syntactic equality).*

Proof. By Corollary 4.9, there must be a state S_3 such that $S_1 \vdash^* S_3$ and $S_2 \vdash^* S_3$. Now there is no rewrite rule which modifies a binding of the form $X^{(0)} = V;$ (see Figure 4.3), so S_3 must contain a binding $X = V_3;$ and furthermore $V_1 = V_2 = V_3$. ■

Corollary 4.18 *The answers contained in different weak normal forms of the same state are equal.*

Proof. Let X be $\Diamond$ in Theorem 4.17. ■

We see, therefore, that it makes sense to stop evaluation when a weak normal form is reached, since the answer contained in a weak normal form will not be altered by further evaluation.[3] If a program has no normal form, however, there is no guarantee that a particular evaluation sequence will reach a weak normal form. So if we are interested in weak normal forms, evaluation order becomes significant. We generally specify an evaluation order through an *evaluation strategy*, which is an algorithm for choosing the next redex to reduce, given a state. At each step, the strategy either identifies a redex to reduce, or indicates that no further reduction is to be done.[4] We can give a characterization of program execution in terms of weak normal forms and evaluation strategies, as follows:

Definition 4.19 *An* execution under strategy $\mathcal{S}$ *of a program S_0 is a reduction sequence $S_0, S_1, \ldots$ where the redex chosen at each step is determined by the strategy $\mathcal{S}$. A* terminating execution under strategy $\mathcal{S}$ *is a finite execution under strategy $\mathcal{S}$ $S_0, \ldots, S_n$ where strategy $\mathcal{S}$ identifies no redexes in S_n. Terminating executions are further classified as* non-deadlocking, *if S_n is a weak normal form, and* deadlocking *otherwise.*

In other words, deadlock means termination with no answer.

We will mainly concern ourselves with the *lenient strategy*, which simply says that at any step, *any* of the available redexes may be evaluated next, unless the state is already in weak normal form, in which case the strategy terminates. How does this correspond to lenient

[3] If the answer in a weak normal form is a data structure, then the variables it contains may or may not have values. If desired, these can be further evaluated, and Theorem 4.17 guarantees their uniqueness.

[4] An alternative way of defining a strategy is as a function from an initial state to the sequence of states resulting from reducing according to the strategy [11].

evaluation, which we described in Chapter 3? We recall that lenient evaluation scheduled a subexpression based on whether it was executable, in the sense of having enough of its input data available to proceed. Furthermore, an arm of a conditional is evaluated only after the predicate is known. Examining the rewrite rules for functional quads, we see that both of these requirements are met automatically by the construction of the $\vdash$ relation. Thus, choosing any available redex corresponds to lenient evaluation. Stopping at a weak normal form reflects the fact that we are only interested in answers.

The *lazy strategy* for functional quads chooses a redex that is required to produce an answer, and so models lazy evaluation. We can describe the lazy strategy as a case analysis on the current state:

$$\mathcal{L}[\![\Diamond \texttt{ = } E;\ \ldots]\!] \;=\; \mathcal{N}[\![E]\!][\![\Diamond \texttt{ = } E;\ \ldots]\!]$$

$$\begin{array}{rcl}
\mathcal{N}[\![V]\!][\![S]\!] & = & \text{[Terminate]} \\
\mathcal{N}[\![E]\!][\![S]\!] & = & E, \qquad \text{if } E \text{ is a redex, otherwise:} \\
\mathcal{N}[\![X^{(0)}]\!][\![\ldots;\ X^{(0)} \texttt{ = } E;\ \ldots]\!] & = & \mathcal{N}[\![E]\!][\![\ldots;\ X^{(0)} \texttt{ = } E;\ \ldots]\!] \\
\mathcal{N}[\![V_1 \texttt{ + } P_2]\!][\![S]\!] & = & \mathcal{N}[\![P_2]\!][\![S]\!] \\
\mathcal{N}[\![P_1 \texttt{ + } P_2]\!][\![S]\!] & = & \mathcal{N}[\![P_1]\!][\![S]\!] \\
\mathcal{N}[\![\texttt{if } P \texttt{ then } \ldots]\!][\![S]\!] & = & \mathcal{N}[\![P]\!][\![S]\!] \\
\mathcal{N}[\![P\ X]\!][\![S]\!] & = & \mathcal{N}[\![P]\!][\![S]\!] \\
\mathcal{N}[\![\texttt{sel\_}t\texttt{\_}i\ P]\!][\![S]\!] & = & \mathcal{N}[\![P]\!][\![S]\!] \\
\mathcal{N}[\![\texttt{is\_}t\texttt{?}\ P]\!][\![S]\!] & = & \mathcal{N}[\![P]\!][\![S]\!]
\end{array}$$

$\mathcal{L}$ basically traces its way back from the answer until it finds a redex needed to make further progress. The strategy has the same desirable termination property as lazy evaluation, namely:

Theorem 4.20 *If a program S_0 has a weak normal form, execution under the lazy strategy always terminates.*

Proof. At every step, $\mathcal{L}$ chooses a redex which must be reduced in any reduction sequence that ends in a weak normal form (demonstratable through case analysis). So any reduction sequence to a weak normal form is a superset of the sequence chosen by $\mathcal{L}$; if there is a finite sequence to a weak normal form, then the $\mathcal{L}$ sequence is finite too. (See [12] for a more general discussion of needed redexes.) ∎

The function $\mathcal{L}$ can be used to obtain a strategy for reducing any state variable to a value, not just $\Diamond$.

To summarize, the behavior of a lenient or lazy implementation of a functional language is modeled in functional quads as finding weak normal forms under the lenient or lazy evaluation strategy, respectively.[5] Weak normal forms from the same program contain the same answers, so the choice of strategy does not affect the result of a program, as long as a weak normal form is found. Whether a weak normal form is found *does* depend on the strategy: the lazy strategy always finds one when one exists, while the lenient evaluation strategy is only guaranteed of finding a weak normal form when the program has a normal form, which is true as long as the program does not contain any infinite loops.

4.7 Denotational Semantics

The theoretical foundation of lenient compilation is based on a syntactic system, the abstract reduction system for functional quads. Nevertheless, we will see in Chapter 6 that semantic reasoning about functional quads programs is often useful. We will therefore give a denotational semantics for functional quads here. To be perfectly rigorous, we should accompany the semantics with a proof that the reduction system and the semantics are congruent. Such an endeavor is well beyond the scope of this thesis, however, and so we will content ourselves with an intuitive conviction that the semantics accurately model the reduction system.

The semantics given below is modeled after the semantics for Haskell given by Johnsson [44].

Notation

$x \rightsquigarrow A$	Inject x into domain A.
$x \vert A$	Project x onto domain A.
$x \in A$	True if x is in A.
$x \downarrow i$	Select the ith component of tuple x.
$x \rightarrow y\ ;\ z$	If x then y else z.

Domains

$$\begin{aligned}
E &= \mathit{Bool} + \mathit{Int} + S + F + \{\mathit{unbound}\} \\
S &= \mathit{StructTag} \times E^* \\
F &= E \rightarrow E \\
\mathit{Env} &= \mathit{Ide} \rightarrow E
\end{aligned}$$

[5] For completeness, we point out that there is also an evaluation strategy for functional quads which models strict evaluation. We will not discuss it in detail, but as an example we point out that it will not select an R8 redex unless all of the identifiers Y_1 through Y_n are bound to values. Hence, there will be programs for which the lenient strategy terminates but the strict strategy deadlocks.

Semantic Functions

$$\begin{aligned}
(\rho_1 \nabla \rho_2)i &= \rho_2\, i = \mathit{unbound} \rightarrow \rho_1\, i \,;\, \rho_2\, i \\
(\rho[e/i_1])i_2 &= i_1 = i_2 \rightarrow e \,;\, \rho\, i_2 \\
\rho_0 &= \lambda i.\mathit{unbound}
\end{aligned}$$

$$\begin{aligned}
\mathcal{K} &: \mathit{Scalar} \rightarrow E \\
\mathcal{P} &: \mathit{Simple} \rightarrow E \\
\mathcal{E} &: \mathit{Simple} \rightarrow \mathit{Env} \rightarrow E \\
\mathcal{D} &: \mathit{Binding} \rightarrow \mathit{Env} \rightarrow \mathit{Env}
\end{aligned}$$

Semantic Equations

$$\mathcal{P}\llbracket \Diamond\mathtt{=}\ E;B_1;\ldots;B_n \rrbracket = \mathcal{E}\llbracket \mathtt{\{}B_1;\ldots;B_n;\ \mathtt{in}\ E\mathtt{\}} \rrbracket\rho_0$$

$$\begin{aligned}
\mathcal{E}\llbracket C \rrbracket\rho &= \mathcal{K}\llbracket C \rrbracket \\
\mathcal{E}\llbracket X \rrbracket\rho &= \rho\, X \\
\mathcal{E}\llbracket \mathtt{const}\ V \rrbracket\rho &= \mathcal{E}\llbracket V \rrbracket\rho \\
\mathcal{E}\llbracket P_1\ \mathit{Op}\ P_2 \rrbracket\rho &= f_{\mathrm{Op}}(\mathcal{E}\llbracket P_1 \rrbracket\rho, \mathcal{E}\llbracket P_2 \rrbracket\rho) \\
\mathcal{E}\llbracket \mathtt{if}\ P\ \mathtt{then}\ Bk_1\ \mathtt{else}\ Bk_2 \rrbracket\rho &= \mathcal{E}\llbracket P \rrbracket\rho|\mathit{Bool} \rightarrow \mathcal{E}\llbracket Bk_1 \rrbracket\rho \,;\, \mathcal{E}\llbracket Bk_2 \rrbracket\rho \\
\mathcal{E}\llbracket P_1\ P_2 \rrbracket\rho &= (\mathcal{E}\llbracket P_1 \rrbracket\rho|F)(\mathcal{E}\llbracket P_2 \rrbracket\rho) \\
\mathcal{E}\llbracket \mathtt{<}t\mathtt{,}P_1\mathtt{,}\ldots\mathtt{,}P_n\mathtt{>} \rrbracket\rho &= \langle t, \mathcal{E}\llbracket P_1 \rrbracket\rho, \ldots, \mathcal{E}\llbracket P_n \rrbracket\rho \rangle \leadsto E \\
\mathcal{E}\llbracket \mathtt{sel\_}t\mathtt{\_}i\ P \rrbracket\rho &= (\mathcal{E}\llbracket P \rrbracket\rho|S) \downarrow 1 = t \rightarrow (\mathcal{E}\llbracket P \rrbracket\rho|S) \downarrow (i+1) \,;\, \bot \\
\mathcal{E}\llbracket \mathtt{is\_}t\mathtt{?}\ P \rrbracket\rho &= (\mathcal{E}\llbracket P \rrbracket\rho|S) \downarrow 1 = t \rightarrow \mathit{true} \,;\, \mathit{false} \\
\mathcal{E}\llbracket \mathtt{\{}B_1;\ldots;B_n;\ \mathtt{in}\ P\mathtt{\}} \rrbracket\rho &= \mathcal{E}\llbracket P \rrbracket(\rho \nabla \mathit{fix}\lambda\rho'.\mathcal{D}\llbracket B_1;\ldots;B_n; \rrbracket(\rho \nabla \rho'))
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}\llbracket B_1;\ldots;B_n; \rrbracket\rho &= \mathcal{D}\llbracket B_1 \rrbracket\rho \nabla \cdots \nabla \mathcal{D}\llbracket B_n \rrbracket\rho \\
\mathcal{D}\llbracket X\ \mathtt{=}\ E \rrbracket\rho &= \rho_0[\mathcal{E}\llbracket E \rrbracket\rho/X] \\
\mathcal{D}\llbracket X\ X_1\ \ldots X_n\ \mathtt{=}\ Bk \rrbracket\rho &= \rho_0[(\lambda x_1.\cdots\lambda x_n.\mathcal{E}\llbracket Bk \rrbracket(\rho[x_1/X_1,\ldots,x_n/X_n])) \leadsto E/X]
\end{aligned}$$

We should point out that the use of *unbound* in this semantics is purely a convenience, as the syntactic restrictions on functional quads programs will prevent the meaning of any subexpression from being *unbound*.

Most of the arguments we present in the following chapters make no use of the denotational semantics. We will refer to it, however, when relating strictness analysis to the kind of dependence analysis we will develop for functional quads. Mainly, we will assert that the meaning of a variable that has not been reduced to a value is $\bot$, and that the meaning of a data structure with tag t none of whose component variables have been reduced to values is $\langle t, \bot, \ldots, \bot \rangle$.

4.8 Appendix: The Structure of the Set *Identifier*

In Section 4.1 we stated that the syntactic set *Identifier* has a particular structure: it is partitioned into sets according to arity, and there is a function *newid* defined over it for producing new identifiers from old ones. In this appendix, which the reader may skip, we give a detailed account of this structure.

What exactly does the function *newid* do? Mathematically speaking, it does not actually "create" anything, but instead is just a ternary relation defined over the set *Identifier*. Given the set *Identifier*, all we require is that *newid* satisfy the following three axioms:

- (*Uniqueness*) If $newid(A, B) = newid(C, D)$, then $A = C$ and $B = D$. In other words, the same identifier cannot be constructed from two different pairs.
- (*Newness*) If $newid(A, B) = C$, then there is no way to obtain either A or B from C by successive applications of *newid*. That is, the following relation:

  $$\{ (X, Z) \mid \exists Y \text{ s.t. } newid(X, Y) = Z \vee newid(Y, X) = Z \}$$

  is acyclic.
- (*Arity Preservation*) If $newid(A, B) = C$, then B and C have the same arity.

The easiest way to visualize *newid* is as a pairing operation, so that $newid(\mathtt{a}^{(0)}, \mathtt{f}^{(3)}) = (\mathtt{a},\mathtt{f})^{(3)}$. Formally, let $s_i = \{a_i, b_i, \ldots\}$ be disjoint sets of symbols for all $i \geq 0$. Then define sets $\sigma_0, \sigma_1, \ldots$ inductively as follows:

$$\begin{aligned} x \in s_i \quad &\Rightarrow \quad x \in \sigma_i \\ \chi \in \sigma_0, v \in \sigma_i \quad &\Rightarrow \quad \langle \chi, v \rangle \in \sigma_i \end{aligned}$$

In this approach, each σ_i is the set of identifiers of arity i, the set of all identifiers is $\bigcup_{i \geq 0} \sigma_i$, and the *newid* operation is simply defined as $newid(\chi, v) = \langle \chi, v \rangle$.

Of course, it is more palatable to think of identifiers just as symbols, not pairs; so what we are really asserting is that the set of symbols *Identifier* together with the *newid* relation is isomorphic to the union of the σ_i sets and the pairing operation defined above. We also note that the easiest way to satisfy condition (3) of name-consistency (see page 51) is to construct programs only from the identifiers isomorphic to the sets $s_0, s_1, \ldots$; none of these identifiers are $newid(X, Y)$ for any X or Y.

Chapter 5

The Analysis Framework

We remarked in Section 4.4 that the meat of compiling sequential code from non-strict programming languages is in choosing an appropriate ordering on subexpression execution. As we have seen, it is not always possible to choose a total ordering at compile time, so the best we can do is to compile code into sequential threads, where instructions within a thread are totally ordered but the relative ordering of threads is determined at run time.

In this chapter, we develop the theoretical basis for compiling non-strict programs into sequential threads. We have already established functional quads as a model of non-strict program execution; our non-strict compilation theory uses the functional quads reduction system to obtain information about subexpression ordering. An outline of this theory is reflected in the path on the left side of the chart in Figure 5.1, where we begin with a procedure definition, expressed in functional quads. We construct a "test program" which applies the function to some input, and from observing the ordering relationships that hold for all possible reduction sequences, we arrive at the *requirement graph* for the test program. We repeat for all possible inputs, then summarize the results in the *function requirement graphs*, which indicate which ordering relationships hold for some inputs and which hold for all inputs. A procedure called *constraint computation* converts this ordering information into constraints upon object code in the form of two *constraint graphs*: one that indicates which instructions *must* be placed in separate threads, and one that indicates how instructions placed in the same thread are to be ordered. With the original functional quads program and the constraint graphs, multi-threaded sequential code can be generated. The theory guarantees that for any input to the procedure, there is some interleaving of the threads which corresponds to a legal reduction sequence of the original functional quads program on that input.

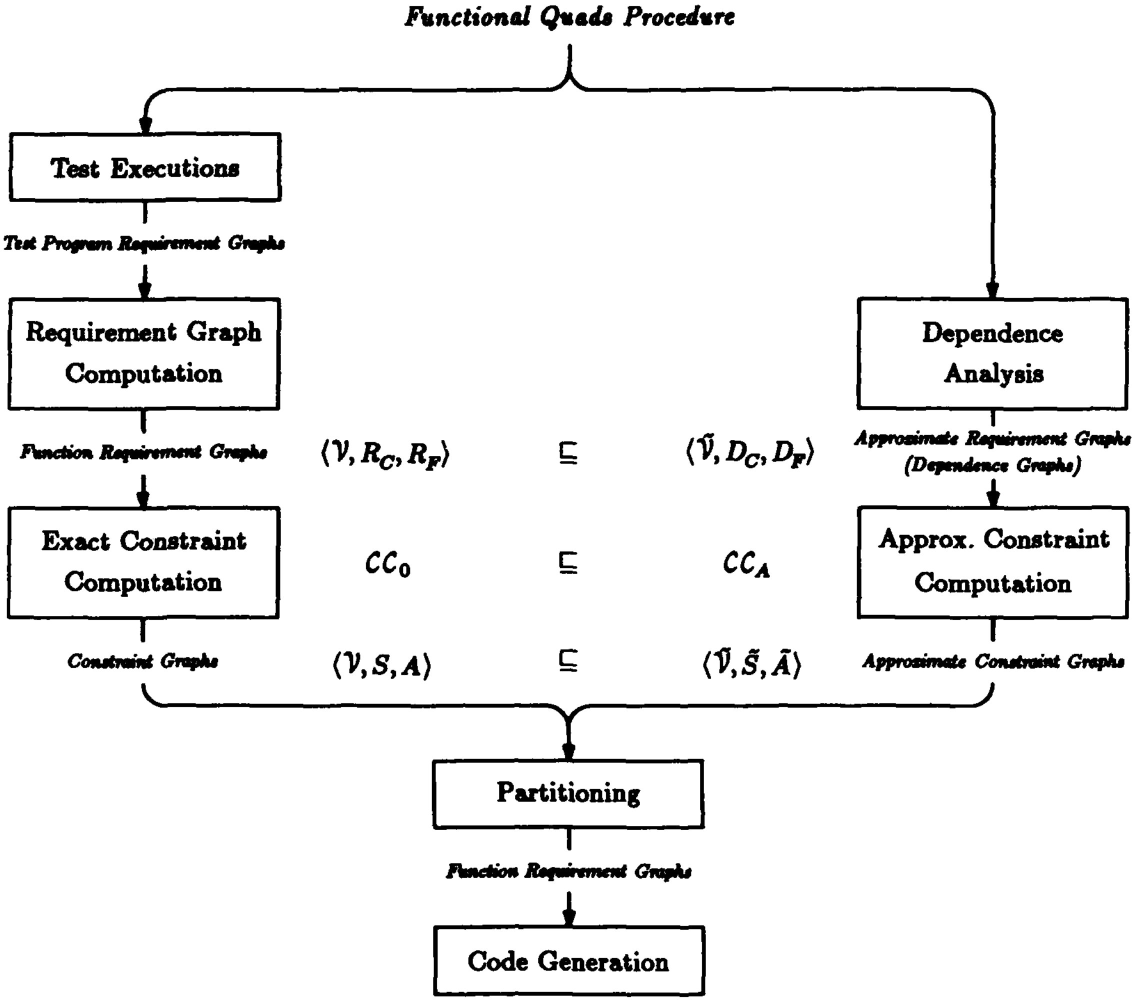

Functional Quads Procedure
Test Executions
Test Program Requirement Graphs
Requirement Graph Computation
Dependence Analysis
Function Requirement Graphs
$\langle \mathcal{V}, R_C, R_F \rangle$
$\sqsubseteq$
$\langle \tilde{\mathcal{V}}, D_C, D_F \rangle$
Approximate Requirement Graphs (Dependence Graphs)
Exact Constraint Computation
CC_0
$\sqsubseteq$
CC_A
Approx. Constraint Computation
Constraint Graphs
$\langle \mathcal{V}, S, A \rangle$
$\sqsubseteq$
$\langle \tilde{\mathcal{V}}, \tilde{S}, \tilde{A} \rangle$
Approximate Constraint Graphs
Partitioning
Function Requirement Graphs
Code Generation

Figure 5.1: Overview of Non-strict Compilation

The theoretical formulation of non-strict compilation outlined above draws its inferences about program behavior only from considering what actually happens when a procedure is executed for various inputs. In this way, it captures the minimum necessary constraints upon code generation, without any bias or prejudice toward a particular method for analyzing programs; all of the constraints derived are ones that were actually observed for some input. Not surprisingly, the theoretical formulation is not a practical method for compilation: obtaining requirement graphs for all possible inputs is clearly an undecidable proposition, and even if the requirement graphs were obtainable their conversion to constraint graphs is NP-complete.

While the theoretical model is not a usable compilation algorithm, it does provide a standard by which the correctness and effectiveness of actual compilers can be judged. Our characterization of non-strict compilation in terms of constraint graphs provides a particularly nice way of making this judgment: any real method of generating constraint graphs must generate at least as many constraints as the theoretical model yields. This notion is formalized, and used as the basis for several safety/correctness proofs.

The analysis that might take place in a real compiler is diagrammed in the path on the right side of Figure 5.1. Through dependence analysis, approximate function requirement graphs are constructed directly, without consideration of test programs (because they are the product of dependence analysis, we will usually call approximate function requirement graphs *function dependence graphs*). An approximate algorithm for constraint computation is then employed to convert the dependence graphs into constraint graphs, which are used in code generation just as are the exact constraint graphs produced by the theoretical model. The discussion of dependence analysis, approximate constraint computation, and code generation is found in Chapters 6, 7, and 8, respectively. The reader should bear in mind, however, that the design of the left side of Figure 5.1 was motivated by the needs of the right side. Some aspects of the theoretical model we present in this chapter may seem mathematically extraneous; they are there to maintain a close connection with what actually takes place in dependence analysis. Perhaps the best way to view the requirement theory developed in this chapter is as a *formal model of data dependence.*

The plan of this chapter is essentially that of the left side of Figure 5.1: we discuss the requirement relation and requirement graphs, constraint computation, and conclude with a discussion of complexity and the definition of approximate constraints.

5.1 Required Reductions

Definition 5.1 *Let* $\mathrm{A}^{S_0 \vdash^* S_1}$ *be the set of reductions for a program* S_0 *to some derivable form* S_1*,* $S_0 \vdash^* S_1$*. Then for* $\alpha, \beta \in \mathrm{A}^{S_0 \vdash^* S_1}$*,* β requires α *if* α *is reduced before* β *in every reduction sequence from* S_0 *to* S_1*.*

We note that requirement is transitive and antisymmetric, and so requirement forms a partial order on the set $\mathrm{A}^{S_0 \vdash^* S_1}$.

Theorem 5.2 *Let* $\alpha_1, \ldots, \alpha_n$ *be a permutation of* $\mathrm{A}^{S_0 \vdash^* S_1}$*. Then* $S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_n} S_1$ *is a valid reduction sequence if and only if the permutation is consistent with the requirement relation on* $\mathrm{A}^{S_0 \vdash^* S_1}$*.*

Proof. The only if side is true by definition. To prove the if side, suppose $S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_n} S_1$ is consistent with the requirement relation but not a valid reduction sequence. Let α_i $(1 \leq i \leq n)$ delimit the valid prefix of this sequence; that is, let i be such that $S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_{i-1}} S' \vdash^* S_1$ is a valid sequence but $S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_i} S'' \vdash^* S_1$ is not. Now by Theorem 4.13 the former sequence must be of the form

$$S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_{i-1}} S' \vdash_{\alpha_{a_1}} \cdots \vdash_{\alpha_{a_{k-1}}} S_a \vdash_{\alpha_{a_k}} S_b \vdash_{\alpha_i} S_c \vdash^* S_1$$

where $i < a_1, \ldots, a_k \leq n$. But because the original sequence is consistent with the requirement relation, for all a_j it is not the case that α_{a_j} must precede α_i, and so by the contrapositive of Theorem 4.15 (strong dependence), α_i must exist in S_a, and we can commute α_{a_k} and α_i to obtain:

$$S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_{i-1}} S' \vdash_{\alpha_{a_1}} \cdots \vdash_{\alpha_{a_{k-1}}} S_a \vdash_{\alpha_i} S_d \vdash_{\alpha_{a_k}} S_c \vdash^* S_1$$

Continuing this process eventually leads to a valid sequence $S_0 \vdash_{\alpha_1} \cdots \vdash_{\alpha_i} S'' \vdash^* S_1$. Contradiction. ∎

We therefore see that the requirement relation gives a necessary and sufficient ordering condition for the construction of an execution sequence from one state to another. Naturally, we are particularly interested in the case of the requirement relation from an initial program to its normal form. We are also interested in the requirement relation from an initial program to some weak normal form, but given that a program has many weak normal forms we need to establish some connection between their respective requirement relations. The following theorem does the trick:

Theorem 5.3 *Let $S_0 \vdash^* S_1 \vdash^* S_2$ and let $\mathrm{A}^{S_0 \vdash^* S_1}$ and $\mathrm{A}^{S_0 \vdash^* S_2}$ be the set of redexes reduced in an execution $S_0 \vdash^* S_1$ and $S_0 \vdash^* S_2$, respectively, with associated requirement relations $\Pi^{S_0 \vdash^* S_1}$ and $\Pi^{S_0 \vdash^* S_2}$. Then (a) $\mathrm{A}^{S_0 \vdash^* S_2} \supseteq \mathrm{A}^{S_0 \vdash^* S_1}$; and (b) $\Pi^{S_0 \vdash^* S_1}$ is equal to the restriction of $\Pi^{S_0 \vdash^* S_2}$ onto $\mathrm{A}^{S_0 \vdash^* S_1}$.*

Proof. (a) Choose any path from S_0 to S_1 and any path from S_1 to S_2. The first path consists of elements of $\mathrm{A}^{S_0 \vdash^* S_1}$; the concatenation of the paths give the elements of $\mathrm{A}^{S_0 \vdash^* S_2}$, which therefore contains $\mathrm{A}^{S_0 \vdash^* S_1}$.

(b) Suppose there were $\beta_1, \beta_2 \in \mathrm{A}^{S_0 \vdash^* S_1}$ such that $(\beta_1, \beta_2) \in \Pi^{S_0 \vdash^* S_2}$ but $(\beta_1, \beta_2) \notin \Pi^{S_0 \vdash^* S_1}$. Then there exists a sequence $S_0 \vdash^* S_1$ where β_2 precedes β_1, and so β_2 precedes β_1 in some sequence $S_0 \vdash^* S_1 \vdash^* S_2$. But then $(\beta_1, \beta_2) \notin \Pi^{S_0 \vdash^* S_2}$; contradiction. Conversely, suppose

there were $\beta_1, \beta_2 \in \mathrm{A}^{S_0 \vdash^* S_1}$ such that $(\beta_1, \beta_2) \notin \Pi^{S_0 \vdash^* S_2}$ but $(\beta_1, \beta_2) \in \Pi^{S_0 \vdash^* S_1}$. So there is a sequence from S_0 to S_2 that looks something like this:

$$\beta\alpha\beta\beta_2\beta\alpha\beta\beta_1\alpha\alpha\beta\alpha$$

where each β denotes an element of $\mathrm{A}^{S_0 \vdash^* S_1}$ and each α denotes an element of $\mathrm{A}^{S_1 \vdash^* S_2}$. Now because $S_0 \vdash^* S_1$ and by the contrapositive of Theorem 4.15 (strong dependence), we can always commute a β with an α, so that we get construct a sequence of all β's followed by all α's, with β_2 preceding β_1. By Theorem 4.14, the β portion of this sequence is $S_0 \vdash^* S_1$. Contradiction. ∎

This theorem says that additional execution of a program adds no new requirement relationships between the redexes that were already reduced.

Corollary 5.4 *The same requirement relationship holds between two redexes α and β in any execution sequence beginning with a state S_0 that includes them both.*

Proof. Immediate by applying Corollary 4.9 (confluence). ∎

5.2 Program Requirement Graphs

When constructing object code, we do not really care about all of the reductions performed by the functional quads reduction system, mainly because substitution reductions happen "automatically" in sequential quads (see the discussion in Section 4.4). To focus our attention on the actual computation performed in a reduction sequence, we introduce the following definition:

Definition 5.5 *Identifier $X^{(0)}$ is* reduced at step i *in a reduction sequence $S_0, S_1, \ldots$ if S_{i-1} contains a binding $X^{(0)} = E$ and S_i contains a binding $X^{(0)} = V$, where V is a value and E is a non-value expression. By convention, $X^{(0)}$ is reduced at step 0 if S_0 contains $X^{(0)} = V$.*

Alternatively, by considering Figure 4.3 we can say that X is reduced at step i in a sequence $S_0 \vdash_{\alpha_1} S_1 \vdash_{\alpha_2} \cdots$ if $\alpha_i = \langle X, \rho \rangle$, where ρ is either R1a, R2, R4, R6, or R7. The terminology is a little funny here: when we say X is reduced at step i, we really mean that the expression on the right hand side of the binding for X is reduced at step i. Because functional quads requires a separate binding for each subexpression, this terminology gives us a way of saying when each subexpression of the program is reduced to a value. Thus do we use identifiers as proxies for the expressions to which they are bound.

Definition 5.6 X_1 precedes X_2 *in an execution sequence* $S_0, S_1, \ldots$ *if there exists* i, j *such that* X_1 *is reduced at step* i*,* X_2 *is reduced at step* j*, and* $i < j$*.* X_2 follows X_1 *if and only if* X_1 *precedes* X_2*.*

Given that, we redefine requirement as a relation on identifiers instead of on redexes:

Definition 5.7 X_2 requires X_1 *in a program* S_0 *if for every non-deadlocking execution of* S_0 *in which* X_2 *is reduced,* X_1 *precedes* X_2*. By convention, if there is* no *non-deadlocking execution of* S_0 *in which* X_2 *is reduced,* X_2 *does* not *require any* X_1*.*

In this definition, we are using the term "non-deadlocking execution" in the sense of Definition 4.19 under the lenient strategy; we restrict our attention to non-deadlocking executions because we only care about compiling programs correctly when they produce answers. Notice that both precedence and requirement on identifiers are transitive, antisymmetric relations. We also note that Theorem 5.3 may allow us to determine requirement relations between identifiers without reducing a program to normal form, so that we can determine requirements between identifiers even for programs which have no normal form.

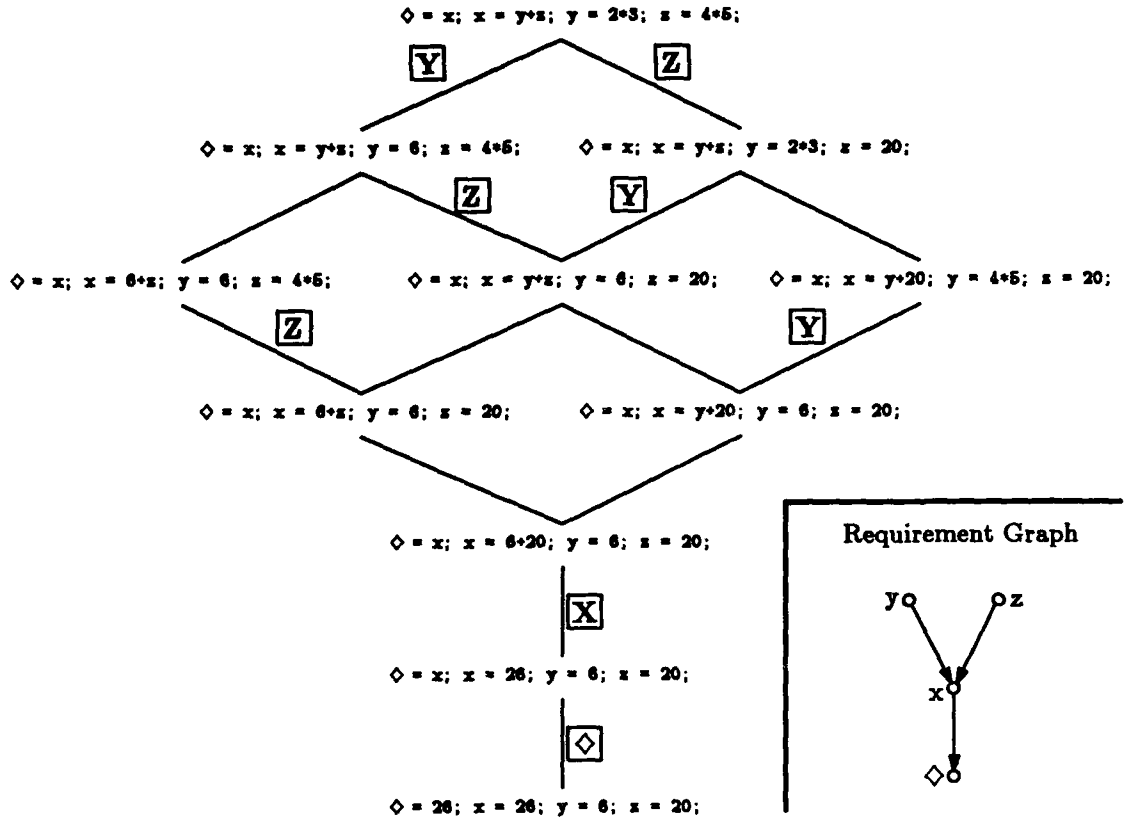
To illustrate these concepts, Figure 5.2 shows all possible reductions of a small program. Each path from the top to the bottom of the figure is a possible reduction sequence, and a step is labeled with an identifier if that identifier is reduced at that step. Now there are some paths for which y precedes z, and some where the reverse is true, but in every path, y and z precede x, and x precedes ◇. So x requires y and z, and ◇ requires x, y, and z. We can summarize these results in a *requirement graph*, a directed graph with a vertex for every variable (*i.e.*, arity 0 identifier) of a program, and a path for every requirement relationship.

We first define the vertex set.

Definition 5.8 *The* local variables *of a binding list,* $\mathcal{LV}[\![B_1;\ldots;B_n;]\!]$*, is the set of variables defined as follows:*

$$\begin{aligned}
\mathcal{LV}[\![B_1;\ldots;B_n;]\!] &= \mathcal{LV}[\![B_1]\!] \cup \cdots \cup \mathcal{LV}[\![B_n]\!] \\
\mathcal{LV}[\![X^{(0)} = E;]\!] &= \{X_0\} \cup \mathcal{LV}[\![E]\!] \\
\mathcal{LV}[\![F^{(n)}\ X_1\ \ldots X_n = \{\ldots\}]\!] &= \emptyset \\
\mathcal{LV}\left[\!\!\left[\begin{array}{l}\texttt{if}\ P\ \texttt{then}\ \{B_{t,1};\ldots;B_{t,n};\ \texttt{in}\ Y_t\} \\ \qquad\quad \texttt{else}\ \{B_{e,1};\ldots;B_{e,m};\ \texttt{in}\ Y_e\}\end{array}\right]\!\!\right] &= \mathcal{LV}[\![B_{t,1};\ldots;B_{t,n};]\!] \cup \mathcal{LV}[\![B_{e,1};\ldots;B_{e,n};]\!] \\
\mathcal{LV}[\![E]\!] &= \emptyset, \qquad \text{if } E \text{ is not a conditional}
\end{aligned}$$

The first line applies whether the binding list is a state or is extracted from a block. The active variables *of a program* S_0*,* $\mathcal{AV}[\![S_0]\!]$*, is the subset of* $\mathcal{LV}[\![S_0]\!]$ *such that* $X \in \mathcal{AV}[\![S_0]\!]$ *only if* X *is reduced in some lenient execution of* S_0*.*



```
◇ = x; x = y+z; y = 2*3; z = 4*5;
◇ = x; x = y+z; y = 6; z = 4*5;
◇ = x; x = y+z; y = 2*3; z = 20;
◇ = x; x = 6+z; y = 6; z = 4*5;
◇ = x; x = y+z; y = 6; z = 20;
◇ = x; x = y+20; y = 4*5; z = 20;
◇ = x; x = 6+z; y = 6; z = 20;
◇ = x; x = y+20; y = 6; z = 20;
◇ = x; x = 6+20; y = 6; z = 20;
◇ = x; x = 26; y = 6; z = 20;
◇ = 26; x = 26; y = 6; z = 20;
```

Figure 5.2: Lenient Rewritings of a Program

The local variables of a program are all variables which could be assigned a value during execution of the program, excluding those variables created when a function is invoked (*i.e.*, a Rule R8 rewrite). Because of the restricted syntax of functional quads, every subexpression of the program has a corresponding local variable. The active variables of a program further exclude those local variables which are never reduced to values in any execution of the program (because they are in a conditional arm that is never selected, for example). It is very important to recognize that the local variable set of a program is a *static* property, which depends only on the syntactic form of the program, while the active variable set is a *dynamic* property, whose composition depends on which variables get reduced during execution.

Definition 5.9 *The* requirement graph of program S_0 *is a directed graph* $\vec{R}^{S_0} = (V, R^{S_0})$, *where* $V = \mathcal{AV}[\![S_0]\!]$ *and* R^{S_0} *is the smallest set such that* $u \overset{R^{S_0}}{\longrightarrow}{}^{+} v$ *iff* v *requires* u *in* S_0.

The vertices of the requirement graph include only those variables which are reduced in some execution of S_0. The edges comprise the transitive reduction of the requirement relation for that program, where the transitive reduction of a graph is the least graph such that its transitive closure is the same as the transitive closure of the original graph. In other words, the requirement graph contains no edges that can be inferred from the transitivity of the requirement relation. We note that because the requirement relation is acyclic, the transitive reduction is unique [1], and so the requirement graph is always well-defined. The requirement graph for the program in Figure 5.2 is shown in that figure's inset.

Before moving on, we should address a small technical detail in our definition of program requirement graphs. If a program starts out with a binding $X = V$, where V is a value, then every other identifier will appear to require X, simply because X is a value in step 0. Similarly, if such a binding appears in an inner block, any binding that becomes a value after the block is added to the state will appear to require it. We remedy this with a restriction: programs for which we construct requirement graphs should not contain a binding of the form $X = V$ either in a state or in an inner block. This restriction is easily met by changing such bindings to X = `const` V.

5.3 Function Requirement Graphs

The program requirement graph only gives information about a particular program. Note that our definition of a program includes what would ordinarily be considered input data, the input data being contained in the bindings of the initial state. Compilers generally do not have the benefit of having a program's input data; instead, for each *function* it must compile code which behaves properly regardless of its arguments. To do this, we will test a function's behavior on a particular set of inputs to get a requirement graph for that input. We will then combine the requirement graphs obtained from testing the function on every possible input, to arrive at graphs which summarize the behavior of the function independent of input.

Definition 5.10 *Given a function $F^{(n)}$ with definition:*

$F\ X_1\ \ldots\ X_n$ = {B_1; ... B_m; `in` P};

a test program *for a function $F^{(n)}$ on input $(V_1, \ldots, V_N)$ is the following program:*

$\Diamond$ = P; X_1 = `const` V_1; ...; X_n = `const` V_n; B_1; ...; B_n

The requirement graph of F on input $(V_1, \ldots, V_n)$, $\vec{R}^F_{(V_1,\ldots,V_n)}$, *is the requirement graph of the test program for F on $(V_1, \ldots, V_n)$.*

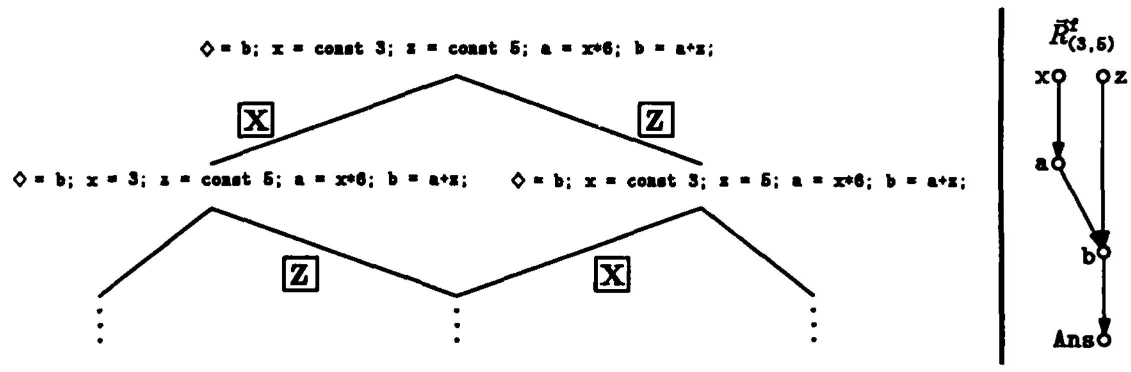

◇ = b; x = const 3; z = const 5; a = x*6; b = a+z;
X
Z
◇ = b; x = 3; z = const 5; a = x*6; b = a+z;
◇ = b; x = const 3; z = 5; a = x*6; b = a+z;
Z
X
R^f_(3,5)
x
z
a
b
Ans

Figure 5.3: Test execution of `f x z = {a = x*6; b = a+z; in b}` on (3,5)

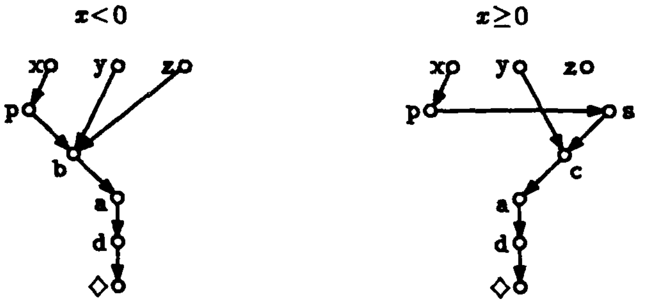
An example of a test program and its associated requirement graph is shown in Figure 5.3. Essentially, the test program simulates the state that results after a call to F is expanded, and so its requirement graph gives the requirement relations that hold when F is called with the tested input.

The definition of test program and requirement graph above holds even if procedure calls (rule R8 reductions) take place during execution of the test program. Because the vertex set of the requirement graph is a subset of F's local variables, the requirement graph will not contain vertices corresponding to bindings added to the state when procedure calls are executed, even if those calls represent recursive calls to F. On the other hand, the computation that goes on within the body of the called procedure *will* be reflected in the requirement graph, because that computation can create requirement relations between the portions of F which compute arguments for the call and the portions of F which receive the results. One minor technicality: if F makes procedure calls, then naturally the test program for F must include the definitions of all procedures that might be called by F or its children.

For any given function F, all requirement graphs on various inputs will have the same set of vertices, but may have different edges due to conditionals. For example, the following function:

```
f x y z =
  {p = x < 0;
   a = if p then
         {b = y + z in b}
       else
         {s = const 6; c = y + s in c};
   d = a + 7
   in
     d};
```

yields one of the two following requirement graphs, depending on the sign of x:



Note that the vertex sets are not identical; the second graph, for example, has no vertex for b, since b is not reduced in any execution where the first argument is positive.

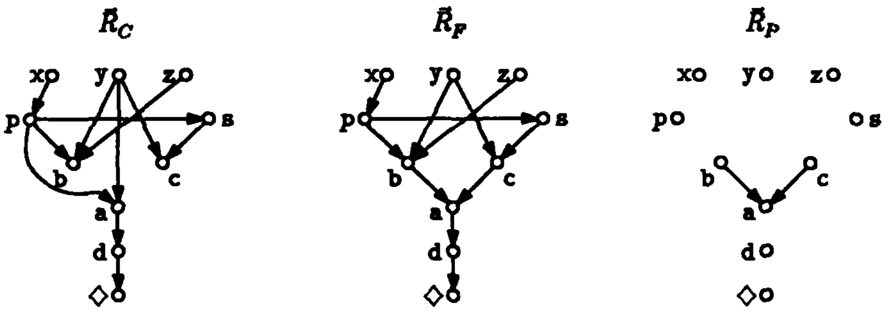
Let $\vec{\mathcal{R}}^F$ be the set of $\vec{R}^F_{(V_1,\ldots,V_n)}$ for all $(V_1,\ldots,V_n)$ such that there is a non-deadlocking execution of the test program for F on $(V_1,\ldots,V_n)$. We then define the following three graphs:

$$\begin{aligned}
R_C &= \text{transitive reduction of } \{\,(u,v) \mid u \xrightarrow{R}{}^{+} v \text{ in all } (V,R) \in \vec{\mathcal{R}} \text{ s.t. } v \in V\,\} \\
R_F &= \{\,(u,v) \mid \exists (V,R) \in \vec{\mathcal{R}} \text{ s.t. } u \xrightarrow{R} v\,\} = \bigcup_{(V,R)\in\vec{\mathcal{R}}} R \\
R_P &= R_F - R_C
\end{aligned}$$

$$\mathcal{V} = \bigcup_{(V,R)\in\vec{\mathcal{R}}} V$$

$$\begin{aligned}
\vec{R}_C &= (\mathcal{V}, R_C) \\
\vec{R}_F &= (\mathcal{V}, R_F) \\
\vec{R}_P &= (\mathcal{V}, R_P)
\end{aligned}$$

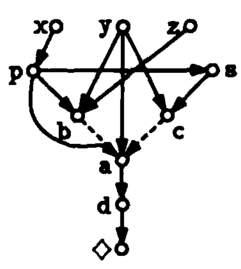
(We have omitted the F superscripts everywhere.) $\vec{R}_C$ is the *certain requirement graph*, $\vec{R}_F$ is the *full requirement graph*, and $\vec{R}_P$ is the *potential requirement graph*. Basically, paths in the certain requirement graph say when a requirement relation exists for *all* arguments, while in the full requirement graph they say when one exists for *some* set of arguments. The potential requirement, being the difference between the two, indicates requirement for some but not all arguments—this graph exists only as a convenience, since it carries no additional information. For the earlier example, we get the following graphs:



If the reader is puzzled by the asymmetry in the definitions of R_C and R_F, we point out that if R_C were simply the intersection of all edge sets in $\vec{\mathcal{R}}$, in the above example $\vec{R}_C$ would not contain paths from p and y to a. $\vec{R}_C$, $\vec{R}_F$, and $\vec{R}_P$ describe the requirement relations for a function in the following way:

1. $u \xrightarrow{R_C}{}^{+} v$ if and only if v requires u for all arguments.
2. $u \xrightarrow{R_F}{}^{+} v$ if and only if v requires u for some set of arguments; moreover,
3. For all argument sets $(V_1, \ldots, V_n)$ there exists a subset $R \subseteq R_F$ such that $u \xrightarrow{R}{}^{+} v$ if and only if v requires u given input $(V_1, \ldots, V_n)$.

There is one additional property which is not immediately obvious: $\vec{R}_P$ contains no paths that are present in $\vec{R}_C$ (we leave the proof to the reader; it depends on the fact that every $\vec{R} \in \vec{\mathcal{R}}$ is a transitive reduction). Throughout the remainder of the thesis, we will use a graphical convention which combines $\vec{R}_C$, $\vec{R}_F$, and $\vec{R}_P$ into a single figure, by using solid lines to denote edges in R_C, and dashed lines for R_P. For the previous example, this gives:



An edge in R_F can be inferred from the presence of either a solid or dashed line.[1]

We should point out that in consolidating all the requirement graphs in $\vec{\mathcal{R}}$ into just three graphs, $\vec{R}_C$, $\vec{R}_F$, and $\vec{R}_P$, we inevitably lose information. Specifically, we know that for each set of arguments there is a subset of R_F which reflects the resulting requirement relation, but there are also many subsets of R_F which do not correspond to any set of arguments. Thus, we lose the information regarding *correlations* between different potential requirements. This is one fundamental limitation of our analysis framework.

5.4 Constraint Computation

The requirement graphs for a function summarize the set of legal execution sequences for that function given various inputs. We now show how to take this information and convert it into constraints upon generating object code. The constraints are in the form of two more graphs: the *separation graph*, which indicates when two subexpressions must be assigned to different sequential threads, and the *a priori ordering constraint*, which indicates the relative ordering between subexpressions when assigned to the same thread.

At first glance, it would seem that ordering object code is simple: $\vec{R}_F$ contains a path from u to v whenever v requires u for some input, and so we have that u must precede v in sequential code. Generating object code would be nothing more than a topological sort of $\vec{R}_F$. Unfortunately, life is not so simple, for while the requirement graph for any *particular* input to a function must be acyclic, $\vec{R}_F$ *can* contain cycles.

Figure 5.4 shows a small function (similar to the one from Section 3.1.3) along with its requirement graphs (again, using the solid/dashed convention). The cycles arise because for negative values of **x**, a and **aa** must be evaluated before b and bb, while for positive values of **x** the reverse is true. As we discussed in Section 3.6, the code which computes **aa** and bb must be placed in separate threads, so that either relative ordering is possible at run time. On the other hand, a and **aa** may be placed in the same thread, but if they are, a must precede **aa**. In general, cycles in $\vec{R}_F$ will result in separation constraints.

To convert the requirement graphs into ordering constraints, we note that for every input to the function that results in a non-deadlocking execution, there is a subset of R_F which reflects

[1] Technically, this convention is ambiguous because there may be some edges in R_C but not in R_F. For any such edge, however, there will always be a *path* in $\vec{R}_F$, and as far as the analysis is concerned we could just as well add these edges to R_F. In fact, we could have unioned R_C into the definition of R_F, but there is no mathematical reason to do so, and it seems superfluous to do so just for the sake of the graphical convention.

```
conditional_example x =
  {p = x > 0;
   a = if p then bb else 3;
   b = if p then 4 else aa;
   aa = a + 5;
   bb = b + 6;
   c = a + b;
   in
     c};
```

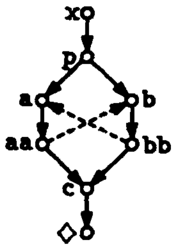


Figure 5.4: A Function with a Cyclic R_F

the requirements for that input, but there are also extraneous subsets which do not correspond to any input. So we must first restrict our attention to those subsets which are reasonable, by defining the *admissible requirement subsets*, $\mathcal{R}_{ADM}$, as follows:

$$\begin{aligned}\mathcal{R}_{ADM} &= \{\, R \mid R \subseteq R_F \wedge R^+ \supseteq R_C^+ \wedge R \text{ is acyclic} \,\}\\ &= \{\, R \cup R_C \mid R \subseteq R_F \wedge R \cup R_C \text{ is acyclic} \,\}\end{aligned}$$

Recall that for any *particular* input, the requirement graph must be acyclic; a cyclic requirement graph would not be reasonable. On the other hand, we do not need to consider any subsets of R_F which do not include all the paths of $\vec{R}_C$, since we know that the requirements in $\vec{R}_C$ exist for all inputs. So the reasonable subsets of R_F are as defined above.

Given the reasonable requirement relations, we can consider which instructions must precede one another during execution. As an intermediate step, we construct *precedence graphs*:

$$\begin{aligned}P_C &= R_C^+\\ P_F &= \{\, (u,v) \mid \exists R \in \mathcal{R}_{ADM} \text{ s.t. } u \xrightarrow{R}{}^+ v \,\} = \bigcup_{R \in \mathcal{R}_{ADM}} R^+\\ P_P &= P_F - P_C\end{aligned}$$

$\vec{P}_C = (\mathcal{V}, P_C)$, $\vec{P}_F = (\mathcal{V}, P_F)$, and $\vec{P}_P = (\mathcal{V}, P_P)$ are the *certain precedence graph*, the *full precedence graph*, and the *potential precedence graph*, respectively, and have an edge from u to v when u must precede v for all inputs, for some input, and for some but not all inputs, respectively.

For every pair of subexpressions, there are six cases of precedence (plus three symmetric variations), as shown in Figure 5.5. In the figure, a solid line denotes an edge in P_C, while a dashed line denotes an edge in P_P; interpreting these cases reveals how to construct the

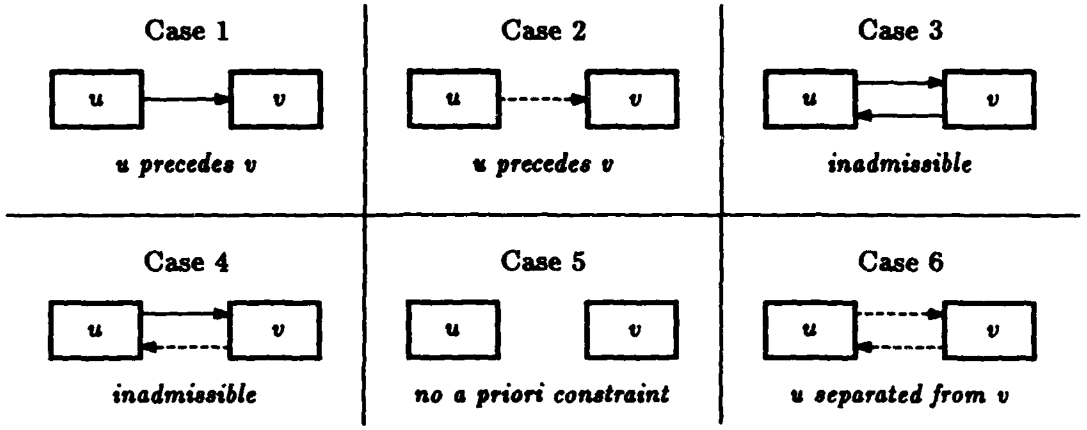


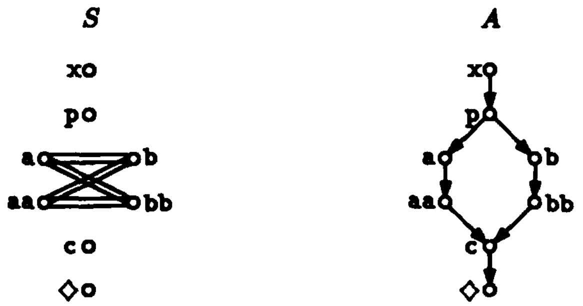
Figure 5.5: Partitioning Constraints Implied by Precedence Graphs

separation graph and the *a priori* ordering constraint. In case 1, u must precede v for every input, so u should precede v if assigned to the same thread. In case 2, u must precede v for some input, so again u should precede v when assigned to the same thread, so that the object code is properly ordered for that input. Cases 3 and 4 will not occur because they are inadmissible: an edge in P_C from u to v implies a path in $\vec{R}_C$, so if there were path in the opposite direction in R, R would be cyclic. In case 5 there is simply no constraint at all, *a priori* (see the discussion below). Finally, in case 6, u must precede v for some input, but v must precede u for some other input. In this case u and v must be placed in separate threads so that either ordering is possible at run time. Mathematically, the separation and ordering constraints are:

$$\begin{aligned} S &= \{\, \{u,v\} \mid (u,v),(v,u) \in P_P \,\} \\ A &= \{\, (u,v) \mid (u,v) \in P_F \wedge (v,u) \notin P_F \,\} \end{aligned}$$

The *separation graph* $\vec{S} = (V, S)$ is an undirected graph with an edge between pairs of subexpressions that *must* be assigned to different threads. The *a priori ordering constraint*, $\vec{A} = (V, A)$ has a directed edge between u and v if u must precede v when assigned to the same thread.

To illustrate, here are the constraints inferred from the precedence graphs for the function in Figure 5.4 (the transitive edges in A have been omitted for clarity):



To partition this program into threads, we need only color the separation graph,[2] each color representing a thread. Each thread is then ordered according to the ordering constraint. For the example program, we see that at least two threads are needed, and that the a's must be in a different thread from the b's, as we expect. More details are given in the next section.

Before proceeding to partitioning, there are two possible points of confusion we would like to address. First, the reader may be confused by the large number of graphs we have introduced. There are really only two pairs of graphs which are of importance: the function requirement graphs $\vec{R}_C$ and $\vec{R}_F$, and the constraint graphs $\vec{S}$ and $\vec{A}$. These graphs have direct analogs in the actual compilation process described in succeeding chapters (see also Figure 5.1); all of the other graphs we introduced in this chapter were merely intermediate steps in the requirement and constraint graphs' definitions.

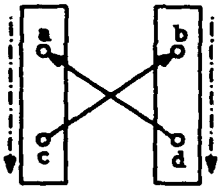
The second point of confusion is the repeated appearance of transitive reduction and transitive closure in the definitions we have given. We started with requirement relations for test programs, took their transitive reduction to arrive at test program requirement graphs, combined these into function requirement graphs $\vec{R}_C$ and $\vec{R}_F$ (with an intermediate closure/reduction step), and finally took transitive closures in defining the precedence graphs upon which the constraint graphs $\vec{S}$ and $\vec{A}$ were based. Why were we so concerned with defining $\vec{R}_C$ and $\vec{R}_F$ as transitive reductions when we were going to take their transitive closures anyway? This is where the mathematics of the theoretical model was influenced by what goes on in a real compiler based on these techniques. Remember that we intend to construct approximations of $\vec{R}_C$ and $\vec{R}_F$ directly from the source program. In so doing, we will consider *local* relationships: how a subexpression makes use of the values produced by the variables it references; the transitive information is redundant, and so we should not have to obtain it during analysis. In

[2] A *k-coloring* of an undirected graph assigns an integer i, $1 \leq i \leq k$, to each vertex such that adjacent vertices are assigned different integers.

fact, we will give algorithms for converting requirement graphs into constraint graphs which avoid the taking of transitive closures altogether. Again, the point is that $\vec{R}_C$, $\vec{R}_F$, $\vec{S}$, and $\vec{A}$ are the important graphs, and all of the transitive closure and reduction is simply the most mathematically convenient way of defining them precisely.

5.5 Partitioning

We stated earlier that partitioning is simply a matter of k-coloring $\vec{S}$ and sorting the threads according to $\vec{A}$. There is, however, a small hitch in this procedure: if there is no edge between u and v either in S or in A, then there is no *a priori* constraint on their relative ordering if assigned to the same thread, but that does not mean that every such pair can be ordered independently. To illustrate what is meant, consider the following situation:



Here we havι four subexpressions a, b, c, and d, with no separation constraints and two *a priori* ordering constraints, indicated by the solid arrows. They have been partitioned into two threads as shown, where the flow of control in each thread runs from top to bottom. The thread orderings are independently consistent with the ordering constraint, *but the net result is incorrect*, as there is no interleaving of these two threads which satisfies the ordering constraints that exist between threads. The problem is that when subexpressions are assigned to the same thread, new precedence relations are added by virtue of the sequential nature of a thread. This is why the ordering constraint is only an *a priori* ordering constraint, since assigning otherwise unconstrained subexpressions to the same thread propagates new precedence constraints through the graph.

The upshot is that the program is partitioned into threads by coloring the separation graph, but when the ordering within each thread is chosen care must be taken not to introduce any new cycles in the requirement graph. To formalize this, we first define what we mean by a partition.

Definition 5.11 *Let $\mathcal{V}$ be the vertex set in the requirement (and constraint) graphs of a procedure. A* partitioning *of $\mathcal{V}$ is a set $\Theta = \{\theta_1, \ldots, \theta_n\}$, where each* thread *$\theta_i$ is an ordered sequence of vertices $v_{i,1}v_{i,2}\cdots v_{i,|\theta_i|}$, $v_{i,j} \in \mathcal{V}$, and where each element of $\mathcal{V}$ appears in exactly one position in one $\theta_i \in \Theta$.*

Given requirement graphs $\vec{R}_C$ and $\vec{R}_F$ and constraint graphs $\vec{S}$ and $\vec{A}$, a legal partitioning of $\mathcal{V}$ must satisfy the following conditions:

1. If there is an edge $(v_{i,k}, v_{j,l}) \in S$, then $v_{i,k}$ and $v_{j,l}$ are assigned to separate threads (*i.e.*, $i \neq j$).
2. If $(v_{i,k}, v_{i,l}) \in A$ and $v_{i,k}$ and $v_{i,l}$ are assigned to the same thread, then $v_{i,k}$ precedes $v_{i,l}$ in the thread (*i.e.*, $k < l$); moreover,
3. Adding edges
$$\bigcup_{1 \leq i \leq |\Theta|} \; \bigcup_{1 \leq j \leq |\theta_i|-1} (v_{i,j}, v_{i,j+1})$$
to R_F must introduce no new cycles into R_F; that is, R_F must have the same set of strongly connected components before and after the edges are added.

We claim that if code is compiled from a partitioning satisfying these three conditions, then for any input to the program there is an interleaving of the threads that corresponds to a valid execution sequence for that program on that input. Interleaving is defined as follows:

Definition 5.12 *An* interleaving *of the threads in Θ is a permutation of the union of all elements of threads in Θ, $v_{t_1,i_1}v_{t_2,i_2}\cdots v_{t_n,i_n}$, such that for all $a < b \leq n$ if $t_a = t_b$ then $i_a < i_b$.*

In other words, statements taken from the same thread must appear in the interleaving in the same order as in the thread.

The correctness of $\vec{S}$, $\vec{A}$, and the conditions outlined above is proved in the following theorem.

Theorem 5.13 *Let Θ be a legal partition of $\mathcal{V}$ given requirement graphs $\vec{R}_C$ and $\vec{R}_F$, according to the definitions of S and A given earlier and satisfying the three conditions outlined above. Then for any admissible subset $R \in \mathcal{R}_{ADM}$ of R_F, there is an interleaving of the threads in Θ which is consistent with the partial order expressed by R.*

Proof. By construction of the appropriate interleaving. We begin with the set of threads Θ and the graph R, and "mark" vertices as they are added to the interleaving. At each step, some vertices will be marked, and we add one more vertex to the interleaving as follows (see Figure 5.6):

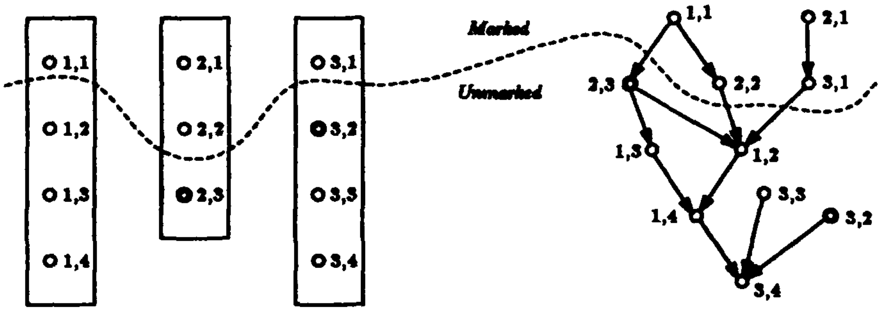


Figure 5.6: Construction of a Consistent Interleaving

1. Find all the unmarked vertices which are not pointed to in R by any unmarked vertex.
2. For each thread θ_i, find the first unmarked vertex $v_{i,f(i)}$ (that is, the vertex $v_{i,f(i)}$ such that $v_{i,j}$ is marked for all $j < f(i)$).
3. Choose any vertex from the intersection of (1) and (2); add this vertex to the interleaving and mark it.

Because in (1) we choose vertices not pointed at by unmarked vertices, we visit R in topological order. Likewise, in (2) we choose the first unmarked vertex in a thread so that we visit the threads in interleaved order. It remains to show that in step (3) there is always at least one vertex from which to choose.

We note that R is acyclic, and because the partitioning Θ adds no new cycles to R_F, $R \cup \bigcup_{i,j}(v_{i,j}, v_{i,j+1})$ is also acyclic; call this graph $\hat{R}$. (In order for $\hat{R}$ to be acyclic for all R, Θ must be consistent with the separation graph.) Now suppose that the intersection in step (3) were empty; then for all $1 \leq i \leq |\Theta'|$, Θ' being the set of threads that have not yet been completely marked, there exists j and k such that $(v_{j,k}, v_{i,f(i)}) \in R$, $k \geq f(j)$ (with the latter condition due to $v_{j,k}$ being unmarked). Now consider the reverse of $\hat{R}$. For all i, there is a path $v_{i,f(i)} \xrightarrow{\hat{R}^{-1}} v_{j,k} \xrightarrow{\hat{R}^{-1}}{}^{*} v_{j,f(j)}$ for some j. There are therefore arbitrarily long paths in the reverse of $\hat{R}$, and therefore it, along with the unreversed $\hat{R}$, is cyclic. Contradiction. ∎

In Chapter 7 we discuss efficient partitioning algorithms that satisfy the three conditions enumerated earlier.

5.6 Complexity and Approximations

We now have a complete theoretical basis for the production of sequential threads from non-strict programs. By modeling execution of the program in an abstract reduction system, functional quads, we arrive at requirement graphs for each function which summarize the necessary and sufficient conditions upon the relative orderings of subexpressions for every non-deadlocking execution of that function. By considering all reasonable patterns of requirement, we arrive at the separation and ordering constraints, which express the essential constraints upon the production of threads in order to preserve correct non-strict execution. Through coloring and topological sorting, these constraints can be employed to actually produce object code. At all stages of this process, we have taken care to isolate the minimum necessary constraints, so that our theoretical framework is not biased by any particular choice of algorithm or analysis technique.[3]

Understandably, the theoretical framework cannot be directly employed in an actual compiler. The requirement graphs for a function are uncomputable: to derive a requirement graph for a program we must examine all non-deadlocking executions, but to know that an execution is non-deadlocking we must know that it terminates, *i.e.*, solve the halting problem. Even if we were able to obtain the requirement graphs, we can show that converting them to the constraint graphs as defined is NP-complete (this is proved in the appendix to this chapter, Section 5.7).

Nevertheless, the framework is an important tool for understanding at an abstract level the technique of non-strict compilation, and also serves as a mathematical standard by which actual compiler algorithms may be judged. In this section, we define formally what it means to approximate the theoretical ideal.

Definition 5.14 *A* constraint set *is a triple $\langle V, S, A\rangle$ where (V, S) is an undirected graph, (V, A) is a directed graph, and $\{u,v\} \in S \Rightarrow (u,v),(v,u) \notin A$. A constraint set $\langle \tilde{V}, \tilde{S}, \tilde{A}\rangle$* approximates *another constraint set $\langle V, S, A\rangle$, notation $\langle \tilde{V}, \tilde{S}, \tilde{A}\rangle \sqsupseteq \langle V, S, A\rangle$, if the following hold:*

$$\begin{array}{rcl} V & \subseteq & \tilde{V} \\ \{u,v\} \in S & \Rightarrow & \{u,v\} \in \tilde{S} \\ (u,v) \in A & \Rightarrow & (u,v) \in \tilde{A} \vee \{u,v\} \in \tilde{S} \end{array}$$

In other words, an approximation to code generation constraints must have more constraints, although the approximation need not include an ordering constraint between two vertices that have a separation constraint in the approximation. Also, the approximation may include extra vertices, with arbitrary edges to and from them.

[3] As we have remarked, however, the framework has a fundamental limitation in that it does not preserve information about correlations between potential requirements.

Theorem 5.15 *$\sqsupseteq$ is a partial order;* i.e., *a reflexive, transitive, antisymmetric relation.*

Proof. Straightforward. ∎

We will use the approximation relation $\sqsupseteq$ to compare methods of obtaining requirement graphs and methods of converting requirement graphs to constraints. Mathematically, we can view the process of constraint computation as a *function* from requirement graphs to constraint graphs.

Definition 5.16 *A* requirement set *is a triple $\langle \mathcal{V}, R_C, R_F \rangle$ where $(\mathcal{V}, R_C)$ and $(\mathcal{V}, R_F)$ are directed graphs, $(\mathcal{V}, R_C)$ acyclic, and if $u \xrightarrow{R_C}{}^{*} v$ then $u \xrightarrow{R_F}{}^{*} v$. A* constraint computation function *is a function from requirement sets to constraint sets. The* standard constraint computation function, *CC_0, is defined as follows:*

$$CC_0(\langle \mathcal{V}, R_C, R_F \rangle) = \langle \mathcal{V}, S, A \rangle$$

$$\begin{array}{rrcl}
\textit{where} & S & = & \{\, \{u,v\} \mid (u,v),(v,u) \in P_P \,\} \\
 & A & = & \{\, (u,v) \mid (u,v) \in P_F \wedge (v,u) \notin P_F \,\} \\
 & P_C & = & R_C^+ \\
 & P_F & = & \{\, (u,v) \mid \exists R \in \mathcal{R}_{ADM} \ s.t.\ u \xrightarrow{R}{}^{+} v \,\} = \bigcup_{R \in \mathcal{R}_{ADM}} R^+ \\
 & P_P & = & P_F - P_C \\
 & \mathcal{R}_{ADM} & = & \{\, R \mid R \subseteq R_F \wedge R^+ \supseteq R_C^+ \wedge R \textit{ is acyclic} \,\} \\
 & & = & \{\, R \cup R_C \mid R \subseteq R_F \wedge R \cup R_C \textit{ is acyclic} \,\}
\end{array}$$

The standard constraint computation function CC_0 just summarizes the equations we presented in Section 5.4; it is the theoretical standard. We extend the definitior of $\sqsupseteq$ to apply to constraint computation functions and also to requirement graphs as follows:

Definition 5.17 *Given two constraint computation functions CC_1 and CC_2, $CC_1 \sqsupseteq CC_2$ if $CC_1(\langle \mathcal{V}, R_C, R_F \rangle) \sqsupseteq CC_2(\langle \mathcal{V}, R_C, R_F \rangle)$ for all requirement sets $\langle \mathcal{V}, R_C, R_F \rangle$. Given two requirement sets $\langle \mathcal{V}, R_C, R_F \rangle$ and $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle$, $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$ if $CC_0(\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle) \sqsupseteq CC_0(\langle \mathcal{V}, R_C, R_F \rangle)$, where CC_0 is the standard constraint computation function.*

In Chapter 6 we describe a method of deriving requirement graphs which approximate the actual requirement graphs for a function, and in Chapter 7 we discuss approximate constraint computation functions.

5.7 Appendix: NP-completeness of CC_0

We commented that computing constraint graphs from requirement graphs using the standard constraint computation function CC_0 is NP-complete. In this appendix we prove that fact.

To begin, we define a graph problem called *Constrained Connected Acyclic Subgraph*, equivalent to deciding membership in P_F, and show that it is NP-complete. The definition of the problem is as follows:

Constrained Connected Acyclic Subgraph (CCAS)

INSTANCE: Directed graph $G = (V, E)$ with specified vertices $s, t \in V$ and partition of E into "solid" edges E_S and "dashed" edges E_D, $E_S \cap E_D = \emptyset$, $E_S \cup E_D = E$.

QUESTION: Is there a subset $E_D' \subseteq E_D$ such that the graph $G' = (V, E_S \cup E_D')$ is acyclic and there is a path in G' from s to t?

We will prove that CCAS is NP-complete by reducing an arbitrary instance of Exact Cover by 3-Sets (X3C) to CCAS.[4] X3C is known to be NP-complete [26]; we reproduce below the definition given there:

Exact Cover by 3-Sets (X3C)

INSTANCE: Set X with $|X| = 3q$ and a collection C of 3-element subsets of X.

QUESTION: Does C contain an exact cover for X, *i.e.*, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'?

For a given instance of X3C, we construct an instance of CCAS in which the vertices V consist of a source s, a sink t, and a grid of pairs of vertices $u_{i,j}$ and $v_{i,j}$, in which each row of the grid corresponds to an element of X and in which each column corresponds to an element of C. (See Figure 5.7.)

Formally, let $N = |X|$ and $M = |C|$. Let $\textsc{row} : X \rightarrow \mathcal{N}$ be a bijection from X onto $[1, N]$ and $\textsc{col} : C \rightarrow \mathcal{N}$ be a bijection from C onto $[1, M]$. Furthermore, let $\textsc{row}^{-1}$ and $\textsc{col}^{-1}$ be the functional inverses of $\textsc{row}$ and $\textsc{col}$, respectively. V is then defined as:

$$V = \{s, t\} \cup \{\, u_{i,j}, v_{i,j} \mid 1 \leq i \leq N,\ 1 \leq j \leq M,\ \textsc{row}^{-1}(i) \in \textsc{col}^{-1}(j)\,\}$$

The basic idea is to provide entirely dashed paths which start at s, pass through every row in sequence, and arrive at t. A particular path will therefore cover every row (and therefore every $x \in X$), using some subset of the columns (*i.e.*, some subset of C), and each such path induces a subgraph consisting of the edges on the path plus all the solid edges. We include solid back edges between pairs of columns that represent non-disjoint 3-sets, so that any path

[4] The author is indebted to Serge Plotkin, who after consultation with David Johnson suggested the construction used for this reduction.

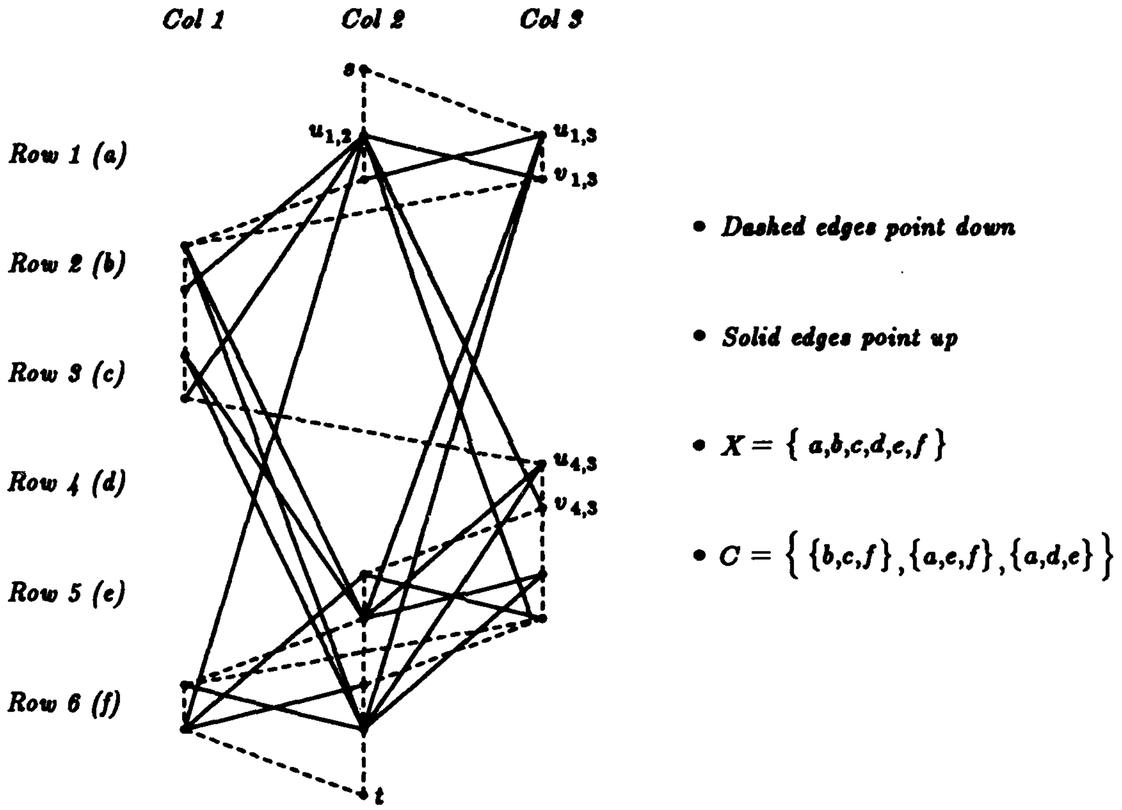


Figure 5.7: Construction Reducing X3C to CCAS

that does not correspond to an exact cover will have a back edge between two of its vertices. In this way, only those paths corresponding to exact covers will induce acyclic graphs.

The dashed edges are as follows:

$$\begin{aligned} E_D \;\; = \;\; & \{ (u_{i,j}, v_{i,j}) \mid u_{i,j}, v_{i,j} \in V \} \cup \{ (v_{i,j}, u_{i+1,k}) \mid v_{i,j}, u_{i+1,k} \in V \} \\ & \cup \{ (s, u_{1,j}) \mid u_{1,j} \in V \} \cup \{ (v_{N,j}, t) \mid v_{N,j} \in V \} \end{aligned}$$

There is a dashed edge between each u and its corresponding v, and a dashed complete bipartite graph between the vs of one row and the us of the next. In addition, there is an edge between s and every u of the first row, and between every v of the last row and t.

The solid edges are:

$$E_S = \bigcup_{\substack{c_1, c_2 \in C \\ c_1 \cap c_2 \neq \emptyset}} \{ (v_{i,\mathrm{COL}(c_1)}, u_{k,\mathrm{COL}(c_2)}) \mid \mathrm{ROW}^{-1}(i) \in c_1,\ \mathrm{ROW}^{-1}(k) \in c_2,\ k \leq i \}$$

Since solid edges go only from vs to us, there are no solid paths of length greater than one.

Lemma 5.18 *An instance of X3C has a solution if and only if the constructed instance of CCAS has a solution.*

Proof (Only If): Let C' be the solution to the instance of X3C; we construct a solution to the corresponding CCAS instance as follows. Since C' is an exact cover for X, there is a bijection f from X onto $[1, M]$ such that $f(x) = \text{ROW}(c)$ where $x \in c$ and $c \in C'$. Let $x_1, x_2, \ldots, x_N$ be the elements of X such that $\text{ROW}(x_1) = 1$, $\text{ROW}(x_2) = 2$, *etc.* Then the following is a path in G composed entirely of dashed edges:

$$s, u_{1,f(x_1)}, v_{1,f(x_1)}, u_{2,f(x_2)}, v_{2,f(x_2)}, \ldots, u_{N,f(x_N)}, v_{N,f(x_N)}, t$$

Such a path exists by construction. Let $E_D{}'$ consist of only those edges on the above path. It remains to show that $G' = (V, E_S \cup E_D{}')$ is acyclic. Since there are no solid paths of length greater than one, the graph is cyclic only if there is a solid edge between two vertices along the dashed path. But solid edges only occur between pairs of columns corresponding to non-disjoint members of C. Since C' is exact, it contains no such pair. The dashed path, therefore, does not pass through any pair of columns between which there is a solid edge, and so there can be no solid edge joining two vertices along the path. G' is therefore acyclic.

(If): A solution to CCAS is a set of dashed edges $E_D{}'$ and a path from s to t. Given a solution, we first construct a canonical solution in which the path is composed entirely of dashed edges. By construction, an entirely dashed path must take the form:

$$s, u_{1,\text{COL}(c_1)}, v_{1,\text{COL}(c_1)}, u_{2,\text{COL}(c_2)}, v_{2,\text{COL}(c_2)}, \ldots, u_{N,\text{COL}(c_N)}, v_{N,\text{COL}(c_N)}, t$$

where $c_1, c_2, \ldots, c_N \in C$ and are not necessarily distinct. If a path from s to t contains solid edges, it must contain one or more segments of the form:

$$\ldots, v_{i,\text{COL}(c_a)}, u_{j,\text{COL}(c_b)}, \ldots, v_{i,\text{COL}(c_c)}, w, \ldots$$

where $j \leq i$, and w is either a u vertex or the final t. The solid edge is $(v_{i,\text{COL}(c_a)}, u_{j,\text{COL}(c_b)})$, and the segment preceding $v_{i,\text{COL}(c_c)}$ may also contain solid edges. Intuitively, a solid edge causes the path to jump back from a v vertex to a lower-numbered u vertex (or to a u in the same row). But if the path ultimately reaches t, it must pass through every row between this lower-numbered row and the last row, and in particular it must pass through row i again, although perhaps in a different column (*i.e.*, c_a and c_c are not necessarily the same). Now if this segment is part of a solution to CCAS, then it can be replaced by just:

$$\ldots, v_{i,\text{COL}\,c_a}, w, \ldots$$

thereby deleting the solid edge(s). There is guaranteed to be an edge in E_D between $v_{i,\mathrm{COL}\,c_a}$ and w by construction; this edge may have to be added to E_D' if it was not already present. Adding this edge cannot introduce any new cycles since the two vertices involved were already connected by a path. By repeating this transformation, we arrive at a solution to the CCAS instance in which the path is entirely dashed.

Given an entirely dashed path, the solution to the X3C problem is:

$$C' = \{\, c \mid v_{i,COL(c)} \text{ is on the path} \,\}$$

Since the path passes through every row, the columns through which it passes must correspond to 3-sets which form a cover for X. Since the path passes through each row only once, the cover is exact. ∎

Theorem 5.19 *CCAS is NP-complete.*

Proof: A potential solution to an instance of CCAS is verified by checking for connectivity between s and t, and by checking for the acyclicness of G'. Since both tests can be performed in polynomial time, CCAS is in NP. An arbitrary instance of X3C can be converted to CCAS using the construction proved in Lemma 5.18. The conversion time is $O(M^2N)$, being dominated by the construction of N bipartite subgraphs, each of size $O(M)$. Since X3C is known to be NP-complete [26], and since the conversion time is polynomial, CCAS is NP-complete. ∎

Looking at the definitions of $\mathcal{R}_{ADM}$ and P_F, we see that deciding whether an edge (s,t) is in P_F is exactly the same as solving the CCAS problem where the solid edges are the edges of R_C and the dashed edges are the edges of R_F. To be completely rigorous, we note that to decide membership in S we do not necessarily need to decide membership in P_F: we just need to decide whether a pair of edges $(u,v),(v,u)$ are both in P_F. Thus, the problem for which we *really* need to show NP-completeness is the following:

Constrained Doubly Connected Acyclic Subgraph (CDCAS)

INSTANCE: Directed graph $G = (V,E)$ with specified vertices $s,t \in V$ and partition of E into "solid" edges E_S and "dashed" edges E_D, $E_S \cap E_D = \emptyset$, $E_S \cup E_D = E$.

QUESTION: Are there subsets $E_D' \subseteq E_D$ and $E_D'' \subseteq E_D$ such that the graph $G' = (V, E_S \cup E_D')$ has a path from s to t, the graph $G'' = (V, E_S \cup E_D'')$ has a path from t to s, and both G' and G'' are acyclic?

Theorem 5.20 *CDCAS is NP-complete.*

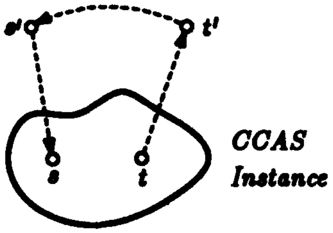


Figure 5.8: Construction Reducing CCAS to CDCAS

Proof. We reduce an instance of CCAS to CDCAS with the construction illustrated in Figure 5.8. Given the CCAS problem, we add new vertices s' and t', and dashed edges (s', s), (t, t'), and (t', s'), where s' and t' are to be the distinguished vertices in the constructed CDCAS problem. We claim that the constructed CDCAS problem has a solution if and only if the original CCAS problem has a solution. To prove the if side: to go from s' to t' use the CCAS subset together with (s', s) and (t, t'); to go from t' to s' use the CCAS subset with (t', s'). To prove the only if side: the CDCAS subset which goes from s' to t' must include the edges (s', s) and (t, t'), and so it also provides a path from s to t. ∎

Chapter 6

Dependence Analysis

In the last chapter, we defined requirement graphs for a function which summarize the ordering relationships that exist for all non-deadlocking executions of that function. As we noted, requirement graphs are uncomputable, since we need to know if a function deadlocks on a particular input. Even if we were willing to restrict our attention to functions which always terminate without deadlock, we would still have to simulate all possible executions on all possible inputs in order to compute the requirement graph, a clearly infeasible approach.

In this chapter we explore methods that approximate the requirement graph of a function through various analysis techniques, which we collectively call *dependence analysis*. We first extend our earlier discussion of approximate requirement graphs by giving criteria for judging the correctness of an approximation directly in terms of requirement graphs. We then develop dependence analysis techniques for functional programs. We begin by restricting our attention to simple cases of programs which do not use data structures and do not make procedure calls. We then extend the analysis to handle function calls, data structures, and finally higher-order functions. We find that strictness analysis and variants thereof are useful tools in performing dependence analysis. Given a proposed dependence analysis method, it is incumbent upon us to show that it yields safe approximations, and we will demonstrate such proofs for the simple cases. We conclude by showing how the ability to feed the result of a function back into its arguments in non-strict functions is largely responsible for the need to compile them into multiple threads.

6.1 Approximate Requirement Graphs

In Section 5.6 we defined what it means to have an approximation to a requirement graph: an approximate requirement graph yields no fewer constraints than the exact requirement graph when both are subjected to the standard constraint computation function. It will be convenient, however, to characterize approximate requirement graphs directly in terms of the exact requirement graph. We will give two such characterizations, each as sufficient conditions on an approximate requirement graph to guarantee safety.

For reference, we repeat here the definition of the standard constraint computation function:

$$\mathcal{CC}_0(\langle \mathcal{V}, R_C, R_F \rangle) = \langle \mathcal{V}, S, A \rangle$$

$$\begin{array}{lrcl} \text{where} & S & = & \{\, \{u,v\} \mid (u,v),(v,u) \in P_P \,\} \\ & A & = & \{\, (u,v) \mid (u,v) \in P_F \land (v,u) \notin P_F \,\} \\ & P_C & = & R_C^+ \\ & P_F & = & \{\, (u,v) \mid \exists R \in \mathcal{R}_{ADM} \text{ s.t. } u \xrightarrow{R}{}^{+} v \,\} \\ & P_P & = & P_F - P_C \\ & \mathcal{R}_{ADM} & = & \{\, R \mid R \subseteq R_F \land R^+ \supseteq R_C^+ \land R \,\text{is acyclic}\, \} \end{array}$$

6.1.1 Characterization I

Theorem 6.1 $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$ *if the following hold:*

$$\begin{array}{rcl} \tilde{\mathcal{V}} & \supseteq & \mathcal{V} \\ \forall u,v \in \mathcal{V},\ u \xrightarrow{\tilde{R}_C}{}^{+} v & \Rightarrow & u \xrightarrow{R_C}{}^{+} v \\ \exists \textit{ acyclic } R \subseteq R_F, R^+ \supseteq R_C^+, u \xrightarrow{R}{}^{+} v & \Rightarrow & \exists \textit{ acyclic } R' \subseteq \tilde{R}_F, R'^+ \supseteq \tilde{R}_C^+, u \xrightarrow{R'}{}^{+} v \end{array}$$

Proof. In proving this result, it is helpful to work with the intermediate graphs P_C, P_F, and P_P that are part of the definition of $\mathcal{CC}_0$. From the conditions given above, we immediately have:

$$\begin{array}{rcll} \forall u,v \in \mathcal{V},\ (u,v) \in \tilde{P}_C & \Rightarrow & (u,v) \in P_C & \\ (u,v) \in P_F & \Rightarrow & (u,v) \in \tilde{P}_F, & \qquad \text{and therefore} \\ (u,v) \in P_P & \Rightarrow & (u,v) \in \tilde{P}_P & \end{array}$$

Let $\langle \mathcal{V}, S, A \rangle = \mathcal{CC}_0 \langle \mathcal{V}, R_C, R_F \rangle$ and $\langle \tilde{\mathcal{V}}, \tilde{S}, \tilde{A} \rangle = \mathcal{CC}_0 \langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle$. We show that each of the conditions for $\langle \tilde{\mathcal{V}}, \tilde{S}, \tilde{A} \rangle \sqsupseteq \langle \mathcal{V}, S, A \rangle$ hold:

$\tilde{\mathcal{V}} \supseteq \mathcal{V}$

Immediate.

$\{u,v\} \in S \Rightarrow \{u,v\} \in \tilde{S}$

We have $\{u,v\} \in S \Rightarrow (u,v) \in P_P \Rightarrow (u,v) \in \tilde{P}_P$, and symmetrically we get $(v,u) \in \tilde{P}_P$. So $\{u,v\} \in \tilde{S}$.

$(u,v) \in A \Rightarrow (u,v) \in \tilde{A} \vee \{u,v\} \in \tilde{S}$

We have $(u,v) \in A \Rightarrow (u,v) \in P_F \Rightarrow (u,v) \in \tilde{P}_F$. There are two cases. If $(v,u) \notin \tilde{P}_F$ then immediately $(u,v) \in \tilde{A}$. If $(v,u) \in \tilde{P}_F$, then we also note that $(u,v) \in A \Rightarrow (v,u) \notin A \Rightarrow (v,u) \notin P_C \Rightarrow (v,u) \notin \tilde{P}_C$, and therefore $(v,u) \in \tilde{P}_P$. But $(v,u) \in \tilde{P}_F \Rightarrow (u,v) \notin \tilde{P}_C$ (because of the admissibility criterion), and so we have $(u,v) \in \tilde{P}_P$ and thus $\{u,v\} \in \tilde{S}$. ∎

6.1.2 Characterization II

The sufficient conditions for safety derived in the last section will be useful in some proofs to be developed later (see Section 6.10, for example), but do not provide very much intuition. We can, however, give conditions which do.

Theorem 6.2 *$\langle \tilde{V}, \tilde{R}_C, \tilde{R}_F \rangle \sqsupseteq \langle V, R_C, R_F \rangle$ if the following hold:*

$$\begin{aligned} \tilde{V} &\supseteq V \\ (\tilde{R}_C^+)|V &\subseteq R_C^+ \\ \tilde{R}_F &\supseteq R_F \end{aligned}$$

Proof. The first two conditions are the same as the first two conditions from Characterization I (the second is just stated differently). The third condition is just a special case of Characterization I's third condition, as we leave the reader to verify. ∎

These are much more intuitive, although they are more restrictive than the conditions in the last section. The first equation just says we need at least the vertices of the exact requirement graph. $\tilde{R}_C^+ \subseteq R_C^+$ says that we should not claim a certain requirement relationship between vertices in the exact graph unless it exists there too; if we did, we might make inadmissible a potential dependence subset that actually arises for some input. On the other hand, there is no harm in including extra transitive edges in $\tilde{R}_C$, which is why we have $\tilde{R}_C^+ \subseteq R_C^+$ instead of $\tilde{R}_C \subseteq R_C$. Similarly, $\tilde{R}_F \supseteq R_F$ says that at least we must claim a potential requirement relationship for every certain or potential requirement relationship that actually exists, for if we omit some we might fail to observe a necessary ordering constraint. We have $\tilde{R}_F \supseteq R_F$ instead of $\tilde{R}_F^+ \supseteq R_F^+$ because the latter does not guarantee that all paths in R_F will have *admissible* counterparts in $\tilde{R}_F$.

6.2 Outline of Dependence Analysis

We wish the computation of approximate requirement graphs to be simple and efficient, so that it can realistically be performed by a compiler. To this end, we will define functions which compute the graphs by a simple recursive descent on the syntax of the program, possibly using the results from other types of analysis (*e.g.*, strictness analysis, type deduction) as input. The rules we give will compute the approximate requirement graph of a function independently from other functions of the program, and so our method is compatible with separate compilation of functions (although the strictness and other analyses we require as input may not be compatible with separate compilation).

The term "approximate requirement graph" is a bit cumbersome, and so we will term the particular approximation described here ***dependence graphs***, in light of their being derived from a consideration of data dependence. We will give the dependence graphs their own symbols D_C and D_F (and D_P, for convenience), and we claim that for any function we have $\langle \tilde{\mathcal{V}}, D_C, D_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$ where $\vec{R}_C$ and $\vec{R}_F$ are the function's exact requirement graphs.

Given a function definition:

$$F^{(n)}\ X_1\ \ldots\ X_n = \{B_1;\ \ldots;\ B_m;\ \mathtt{in}\ X\};$$

the "top-level" of dependence analysis is as follows:

$$\mathcal{V} = \{X_0, \ldots, X_n\} \cup \mathcal{LV}[\![B_1;\ \ldots;\ B_m]\!] \cup \{\Diamond\}$$

$$\begin{aligned} D_C &= \mathcal{DC}[\![B_1;\ \ldots;\ B_m]\!] \cup \{(X, \Diamond)\} \\ D_F &= D_C \cup D_P \\ D_P &= \mathcal{DP}[\![B_1;\ \ldots;\ B_m]\!] \end{aligned}$$

$$\begin{aligned} \vec{D}_C &= (\mathcal{V}, D_C) \\ \vec{D}_F &= (\mathcal{V}, D_F) \\ \vec{D}_P &= (\mathcal{V}, D_P) \end{aligned}$$

The vertex set $\mathcal{V}$ can be recognized as just the local variables of a test program for F (the definition of local variables, $\mathcal{LV}$, was given on page 78). This, of course, is a superset of the vertices in the exact requirement graphs for F (it may include some extra vertices if F has some local variables that are never reduced to values). The auxiliary functions $\mathcal{DC}$ and $\mathcal{DP}$ which define the edges of the dependence graphs are a case analysis on the grammar of functional quads. We give the first cases below:

$$\begin{aligned} \mathcal{DC}[\![B_1;\ \ldots;\ B_m]\!] &= \mathcal{DC}[\![B_1]\!] \cup \cdots \cup \mathcal{DC}[\![B_1]\!] \\ \mathcal{DP}[\![B_1;\ \ldots;\ B_m]\!] &= \mathcal{DP}[\![B_1]\!] \cup \cdots \cup \mathcal{DP}[\![B_1]\!] \end{aligned}$$

So a function is analyzed on a binding-by-binding basis. We will give the remaining cases in the sections that follow, as we consider more and more complicated types of expressions. Before proceeding, however, we wish to introduce two restrictions on the programs we analyze which will simplify the presentation.

First, we restrict the input syntax so that values only appear in a **const** *V* binding; all instances of *Primary* in the grammar (Figure 4.1) should be considered *Identifier*. This is only a restriction on the program submitted as *input* to analysis; if we were to start rewriting this program, the substitution rules would propagate values to other positions within expressions. There is nothing deep about this restriction: it is merely to reduce the number of cases we consider. Without the restriction, for example, there would be four cases of an arithmetic expression, corresponding to the choice of *Value* or *Identifier* for each of two operands.

The second restriction has to do with internal definitions; that is, a binding of the form *Identifier*$^{(n)}$ *Identifier* ... *Identifier* = {...} occurring in the body of another function. In general, such a definition may have *free variables*: identifiers which are defined in the surrounding scope. For the purposes of identifying requirement relationships, such free variables behave like formal parameters, as every call to the function must implicitly pass in the free variables. There is, of course, considerable variation in how this implicit parameter passing may be implemented; the free variables could be passed in as ordinary parameters, or the function could be compiled to fetch them from an environment located in the heap or on the stack [50]. The implementation details are irrelevant as far as the requirement relations are concerned, and so we may as well treat free variables as additional formal parameters to the function. In other words, we will perform the *lambda lifting* transformation [42] which adds the extra parameters and moves the internal definitions to top level. We then need only consider top level definitions during analysis, even if the lambda lifting transformation is not actually used to generate code.

6.3 Arithmetic Expressions

We start by considering programs composed only of arithmetic expressions, so that the only kind of *Value* we allow is *Scalar*, and the only kinds of *Simple* are **const** *Value*, *Identifier*, and *Identifier Op Identifier*.

The cases we add to $\mathcal{DC}$ and $\mathcal{DP}$ are:

$$\begin{aligned}
\mathcal{DC}[\![X = \textbf{const}\ C]\!] &= \emptyset \\
\mathcal{DC}[\![X = Y]\!] &= \{(Y,X)\} \\
\mathcal{DC}[\![X = Y_1\ Op\ Y_2]\!] &= \{(Y_1,X),(Y_2,X)\}
\end{aligned}$$

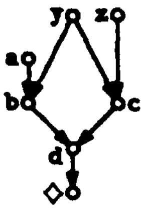
$$\begin{aligned}
\mathcal{DP}[\![X = \textbf{const}\ C]\!] &= \emptyset \\
\mathcal{DP}[\![X = Y]\!] &= \emptyset \\
\mathcal{DP}[\![X = Y_1\ Op\ Y_2]\!] &= \emptyset
\end{aligned}$$

Notice that, for this restricted class of programs, there are only certain dependences. This is to be expected, for without conditionals the order in which subexpressions are evaluated cannot depend on input data.

For example, given this program:

```
f y z =
  {a = const 6;
   b = a + y;
   c = y * z;
   d = b / c;
   in
     d};
```

The vertex set V is $\{\mathtt{y},\mathtt{z},\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\Diamond\}$, and the dependence graph is:



To see the intuition behind the rules, recall that we have a certain dependence only when there is a certain requirement, and that a certain requirement between u and v means that u must be reduced to a value before v can be. Clearly, then, an arithmetic operation requires both its operands, hence we add an edge to the left hand side variable from each of the two variables on the right hand side.

We now prove the safety of $\mathcal{DC}$ and $\mathcal{DP}$ as defined so far:

Theorem 6.3 *The rules above always yield safe dependence graphs.*

Proof. *(Sketch)* To show that $D_C \subseteq R_C$, note that all edges in D_C are from Y to X in statements like X = Y Op Z. The right hand side cannot be reduced as long as Y is not substituted, so Y always precedes X. Since $D_C \subseteq R_C$, $D_C^+ \subseteq R_C^+$. To show that $D_F \supseteq R_F$, we must show that if $(X,Y) \notin D_F$ then there is some execution where Y precedes X. From the definition of $\mathcal{DC}$, $(X,Y) \notin D_F$ means that Y is not a "syntactic descendant" of X, where by this we mean that Y's right hand side has no occurrence of X, nor of any identifier whose right hand side has an occurrence of X, *etc.* Now suppose we have an execution where X precedes Y. From it we can construct an execution where Y precedes X by using Theorems 4.8 (commutivity) and Theorem 4.15 (strong dependence) to commute redexes as follows. In the reductions between where X is reduced and Y is reduced, there are redexes involving syntactic descendants of X, those involving syntactic predecessors of Y, and those involving neither (there are none involving both, because then Y would be a syntactic descendant of X). All but the syntactic descendants of X can be commuted to appear before X, and all but the syntactic predecessors of Y can be commuted to appear after Y, leaving a valid sequence where Y immediately follows X. These can then be commuted to arrive at the desired sequence. ∎

In fact, from the proof we see that for this restricted class of programs, the dependence relation is exact. Because of the tedium of these safety proofs, we will not give them for other types of expressions, relying hereafter on intuition to convince ourselves of correctness.

6.4 Conditionals

The rules for conditional expressions are as follows:

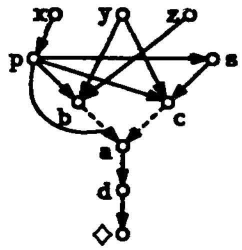
$$\begin{aligned}
\mathcal{DC}\left[\!\!\left[\begin{array}{l} X\ \texttt{= if}\ Y\ \texttt{then}\ \{B_{t,1}\texttt{;}\ldots\texttt{;}B_{t,n}\texttt{; in}\ Y_t\} \\ \qquad\qquad\quad \texttt{else}\ \{B_{e,1}\texttt{;}\ldots\texttt{;}B_{e,m}\texttt{; in}\ Y_e\} \end{array}\right]\!\!\right] = \ & \{(Y,Z) \mid Z \in \mathcal{LV}[\![B_{t,1};\ldots;B_{t,n};]\!]\} \cup \\
& \{(Y,Z) \mid Z \in \mathcal{LV}[\![B_{e,1};\ldots;B_{e,m};]\!]\} \cup \\
& \mathcal{DC}[\![B_{t,1};\ldots;B_{t,n};]\!] \cup \\
& \mathcal{DC}[\![B_{e,1};\ldots;B_{e,m};]\!] \cup \\
& \{(Y,X)\}
\end{aligned}$$

$$\mathcal{DP}\left[\!\!\left[\begin{array}{l}X\ \texttt{=}\ \texttt{if}\ Y\ \texttt{then}\ \texttt{\{}B_{t,1}\texttt{;}\ldots\texttt{;}B_{t,n}\texttt{;}\ \texttt{in}\ Y_t\texttt{\}}\\ \qquad\qquad\ \texttt{else}\ \texttt{\{}B_{e,1}\texttt{;}\ldots\texttt{;}B_{e,m}\texttt{;}\ \texttt{in}\ Y_e\texttt{\}}\end{array}\right]\!\!\right] = \begin{array}{l}\{(Y_t,X),(Y_e,X)\}\cup\\ \mathcal{DP}[\![B_{t,1};\ldots;B_{t,n};]\!]\cup\\ \mathcal{DP}[\![B_{e,1};\ldots;B_{e,m};]\!]\end{array}$$

Basically, we are asserting that nothing in either arm can become a value until the predicate is known, that the left hand side cannot become a value until the predicate is known, and that for some inputs the left hand side further requires the result of one arm, and for other inputs the other arm.

To illustrate, we present again the example from page 82 along with the dependence graph generated by the above rules:

```
f x y z =
  {p = x < 0;
   a = if p then
          {b = y + z in b}
        else
          {s = const 6;
           c = y + s in c};
   d = a + 7
   in
     d};
```



(We are cheating a bit by not introducing **const** statements for the scalars 0 and 7 that appear in the program; in so doing we are implicitly assuming the appropriate clause for $\mathcal{DC}$ that handles arithmetic expressions with one value operand.) We leave it as an exercise for the reader to compare this dependence graph with the requirement graphs on page 83 and use Theorem 6.2 to verify that the dependence graph is safe.

These rules for conditional expressions can be improved. For one, we note that we add many transitive edges to D_C, which are unnecessary. In the program above, for example, an edge between p and c is included, even though there are edges between p and s and between s and c. So we can amend the rule by saying that any transitive edges added to D_C can be removed. This has no effect at all on the quality of our analysis, but may allow a compiler to deal with significantly smaller data structures.

A more important improvement is that if there is any v such that there are paths in D_C from v to both Y_t and Y_e, then we should add an edge from v to X to D_C. In other words, if both arms of a conditional certainly depend on some variable, then so does the result of the conditional. In the example above, this amendment would add a solid edge from y to a, which was present in the requirement graph on page 83 but did not appear in our original dependence graph.

6.5 Interlude: Dependence Analysis vs. Strictness Analysis

We recall that a function f is strict in its ith argument if $f(v_1, \ldots, v_{i-1}, \perp, v_{i+1}, \ldots, v_n) = \perp$ for all $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$. Now suppose the function f is expressed in functional quads:

$$f\ X_1\ \ldots\ X_n\ \mathtt{=}\ \mathtt{\{}B_1\mathtt{;}\ \ldots\ B_m\mathtt{;}\ \mathtt{in}\ P\mathtt{\};}$$

A test program for f on input $v_1, \ldots, v_n$ is:

$$\Diamond\ \mathtt{=}\ P\mathtt{;}\ X_1\ \mathtt{=}\ \mathtt{const}\ v_1\mathtt{;}\ \ldots\mathtt{;}\ X_n\ \mathtt{=}\ \mathtt{const}\ v_n\mathtt{;}\ B_1\mathtt{;}\ \ldots\mathtt{;}\ B_n$$

In functional quads, an expression cannot directly gain any information from a variable; instead, the variable must be substituted into the expression, which in turn requires that the variable be reduced to a value. So, saying that a variable X is never reduced to a value is equivalent to saying that $X = \perp$ (see the denotational semantics and discussion in Section 4.7). Now in the test program for f above, $\Diamond$ is reduced to the value computed by the function, if it computes one. So we can rephrase the definition of strictness in terms of reducing identifiers in the test program, as follows: f is strict in its ith argument if $\Diamond$ cannot be reduced to a value as long as X_i has not been reduced to a value. In other words, f is strict in its ith argument if $\Diamond$ requires X_i for all inputs.

From this definition of strictness, we see that saying a function is strict in an argument is equivalent to saying that there is a path from that argument to the answer in the certain requirement graph for that function. We note that requirement and strictness are both undecidable properties of a function. But as dependence analysis yields an approximation to the requirement relation, it also yields an approximation to strictness, and is therefore a form of strictness analysis. Strictness analysis and this aspect of dependence analysis share the same safety condition: strictness analyzers are designed to never falsely claim strictness, while safe dependence analysis does not falsely claim a certain requirement relation. So we see that our certain dependence rules are just another way of doing strictness analysis.

In the next section, we will turn this argument on its head, and use strictness analysis to assist in adding certain dependence edges to a dependence graph. Before moving on to this, however, we wish to point out a connection between *potential* dependence and a variation on strictness analysis. We can define what it means for a function to ignore an argument as follows:

Definition 6.4 *A function f of n arguments* ignores *its ith argument if*

$$f(x_1,\ldots,x_{i-1},\bot,x_{i+1},\ldots,x_n) = f(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_n)$$

for all $x_1,\ldots,x_n$.

This is a property that can be analyzed using methods similar to strictness analysis; examples include the strictness analyzers of Wray [24] and of Hughes and Wadler [77]. If a function ignores its ith argument, then for no input does the result of the function require that argument, and so ignoring an argument implies a *lack* of a path from the argument to the result in the full requirement graph. To summarize:

Strictness Property	*Requirement Graph Property*
Strict in ith argument	$(X_i,\Diamond) \in R_C^+$
Ignores ith argument	$(X_i,\Diamond) \notin R_F^+$

6.6 First-Order Functions

In this section we consider first-order uses of functions. To do this, we allow partial applications to appear in a program, but only as part of a binding of the form:

$$X = (F^{(n)}\ Y_1\ \ldots\ Y_{n-1})\ Y_n;$$

We suppose that a strictness analyzer is available to us, which computes the following two functions:

$$\begin{aligned} \mathit{Strict}(f) &= \{\, i \mid f \text{ is inferred to be strict in its } i\text{th argument} \,\} \\ \mathit{Ignored}(f) &= \{\, i \mid f \text{ is inferred to ignore its } i\text{th argument} \,\} \end{aligned}$$

Either of these functions can be approximated by the empty set if unavailable. We then add the following two cases to $\mathcal{DC}$ and $\mathcal{DP}$:

$$\begin{aligned} \mathcal{DC}[\![X = (F^{(n)}\ Y_1\ \ldots Y_{n-1})\ Y_n;]\!] &= \{(Y_i,X) \mid i \in \mathit{Strict}(f)\} \\ \mathcal{DP}[\![X = (F^{(n)}\ Y_1\ \ldots Y_{n-1})\ Y_n;]\!] &= \{(Y_i,X) \mid i \notin \mathit{Strict}(f) \wedge i \notin \mathit{Ignored}(f)\} \end{aligned}$$

Again, this is fairly intuitive: X requires Y_i if F is strict in that argument, does not require it if F ignores that argument, and may require it otherwise. We can gain a deeper understanding by seeing how a function call is rewritten during execution of a program.

Suppose we are analyzing function F, which makes a procedure call to another procedure G, where F and G are defined as:

$$\begin{array}{l} F\ X_1\ \ldots\ X_{n_f} = \{B_{F1};\ \ldots;\ B_{Fm};\ \mathtt{in}\ X\}; \\ G\ Y_1\ \ldots\ Y_{n_g} = \{B_{G1};\ \ldots;\ B_{Gm};\ \mathtt{in}\ Y\}; \end{array}$$

where somewhere within the body of F there is a binding

$$Z = (G\ Z_1\ \ldots\ Z_{n_g-1})\ Z_{n_g};$$

Our analysis of F is trying to approximate the requirement relationships that hold for various test programs for F. Now consider such a test program at the point where the call to G is reduced. We have the following two steps:

$$\begin{array}{ll} & \ldots;\ Z = (G\ Z_1\ \ldots\ Z_{n_g-1})\ Z_{n_g};\ \ldots \\ \vdash & \\ & \ldots;\ Z = Y';\ Y'_1 = Z_1;\ \ldots;\ Y'_{n_g} = Z_{n_g};\ B'_{G1};\ \ldots;\ B'_{Gm};\ \ldots \end{array}$$

In expanding the call to G, we add bindings to the state which are isomorphic to the bindings that would exist in a test program for G. Suppose that G is strict in its ith argument; then in every test program for G the answer is never reduced to a value unless Y_i is. In the state above, then, Y' and therefore Z cannot be reduced to a value until Y'_i and therefore Z_i is. So Z certainly requires Z_i when G is strict in its ith argument. Likewise, if G ignores its ith argument, then given any test program for G there is an execution in which the answer is reduced to a value before Y_i is. So in the state above, if we have some execution in which Z and therefore Y' is reduced to a value, we can find an execution in which Z is reduced to a value but Y'_i is not, and from there we can find one in which Z_i is not. So Z definitely does not require Z_i when G ignores its ith argument. Hence, the rules for dependence given above are safe, assuming the strictness analysis method we use is also safe.

At this point, it is reasonable to ask why we have bothered to bring strictness analysis into the picture at all, since it seems to tell us nothing that we did not already know through dependence analysis. Putting it another way, since we are going to perform dependence analysis on all functions of a program anyway, why not use the paths found in analyzing G to supply dependence arcs for a call to G contained in F? In fact, this approach is technically feasible, but has pragmatic difficulties if functions are recursive or mutually recursive. Suppose F calls itself. In analyzing the dependences for F, we need to add arcs to the vertex that represents the recursive call, which in turn requires that we have already analyzed F so that we know which arguments have certain paths to the output, and which have no path at all. This sort of chicken-and-egg problem occurs whenever any kind of analysis is performed on recursive programs, and the usual solution is to set up a recursive equation describing the analysis, and solve by finding least fixpoints. There is no theoretical problem in applying this technique here; we could set up equations for D_C^F and D_P^F which were expressed in terms of paths through D_C^F and D_P^F, and solve to obtain the dependence graphs we desire (see [31] for an example of this kind of graphical fixpoint equation). In fact, the equations that would be obtained are isomorphic to the recursive equations that might be solved by a conventional strictness analyzer.[1] So in some sense the choice between setting up recursive graph equations or appealing to an outside analysis technique is simply a matter of taste. We felt that bringing strictness analysis into the picture is instructive for two reasons: one, it eliminates the need for us to describe the mechanics of recursive graph equations and proving that the necessary fixpoints exist; and two, it illustrates how the information obtained from analysis techniques other than the ones we propose can be incorporated directly into our framework.

Our earlier discussion of how strictness summarizes the input/output dependences of a function validates our rules for obtaining a dependence graph for a top-level function F, where the dependence graph approximates the requirement relations we obtain from test executions of F. On the other hand, there is no guarantee that the requirement relations for test executions of G will hold among the bindings for G added to the state when G is called from F. Specifically, G may be used in a context where its output is eventually fed back into one of its arguments, which would impose additional requirement relations between subexpressions of G. These must be taken into account when compiling code for G. We will defer this point until Section 6.9, however, as we cannot illustrate it with reasonable examples until after we have added data structures to the language.

[1] Specifically, we conjecture that they would correspond to the equations solved in backward analysis analyzers [24, 77], since the dependence graphs do not encode correlation between potential requirements.

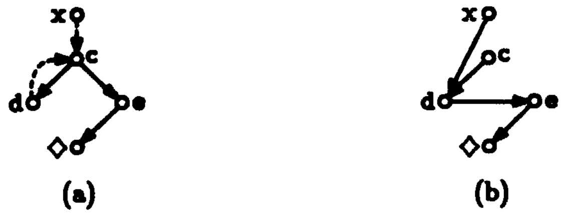


Figure 6.1: (a) Unsafe Dependence Graph; (b) Requirement Graph

6.7 Non-Flat Domains

We now attack dependence analysis for programs that use data structures. In addition to all the constructs allowed so far, we add *Struct* to the allowable values and the expressions **sel_*t*_*i*** *Identifier* and **is_*t*?** *Identifier* to the allowable *Simples*. Now consider the following small program, wherein **duple** is a 2-ary data type:

```
f x =
  {c = <duple,x,d>;
   d = duple_sel_1 c;
   e = duple_sel_2 c;
   in
     e}
```

Our first thought is to consider just the data constructors and selectors as ordinary functions, and use the rules from the last section to analyze this program. A data constructor is not strict in either argument (since **<duple,⊥,⊥>** $= \langle\perp,\perp\rangle \neq \perp$), nor does it ignore either argument (since **<duple,⊥,4>** $\neq$ **<duple,3,4>** $\neq$ **<duple,3,⊥>**). A selector, however, is strict in its argument: **duple_sel_1 ⊥** $= \perp$. Applying the dependence rules developed so far, then, gives the dependence graph in Figure 6.1a. Unfortunately, this graph is *not* a safe approximation of the actual requirement graphs for **f**. We can return to first principles and derive the requirement graphs by considering all executions of **f**, arriving at the graph in Figure 6.1b. As we would expect, **e** certainly requires **d**, but this relationship is absent from the dependence graph obtained from the strictness rules. If we used the dependence graph in Figure 6.1a to obtain an ordering constraint, it would lack an edge from **d** to **e**, and so we might compile code in which **e** appears before **d** in a sequential thread. Such code would be incorrect.

To understand why dependences involving data structure primitives cannot be inferred based on their strictness, we must remember why strictness worked for finding dependences through ordinary function calls. We observed the state resulting from reducing a procedure call, and concluded that the requirement relation for the caller should be equivalent to splicing in the requirement relation of the callee into the caller's graph. We could therefore summarize the effect on the caller by considering the paths from arguments to result in the requirement graphs of the callee, and we had showed already that this summary information was obtainable from strictness analysis. So strictness analysis was just a shortcut that simulated the reduction of the function call, transforming the program into a form without function calls, for which we already knew how to find dependence graphs. Now if there were some way of expressing data constructors and selectors in terms of the arithmetic expressions and conditionals that we already know how to handle, we could expect our current methods to apply to data structures as well. Unfortunately, there is no way to express a non-strict data constructor in terms of arithmetic and conditional expressions.[2] There is no logical reason, therefore, for the strictness of constructors and selectors to summarize correctly the dependence relationships that hold through them, and, as our example showed, it does not.

The failure of our previous methods arises because data structures carry more than one atomic piece of information. If a variable is bound to a scalar expression, then before it becomes a value the variable carries no information to the places which use it, and after it becomes a value it carries all the information it will ever carry. For a scalar variable, being reduced to a value is the only event which can make any difference to its consumers. A data structure, on the other hand, carries several pieces of information, which may become available to consumers at different times, owing to non-strictness. That is, if a variable is reduced to a value that is a data structure, some consumers may have all the information they need, but others may not because not all components have become values. If those components become values at some later time, the other consumers can proceed.

The fallacy in Figure 6.1a is that d and e are treated as if c's being reduced to a value were the only event of interest to them. E needs the value of c, but since d does also, the production of c cannot depend on d, and so a dependence from d to e is ruled out. In reality, of course, e depends not only on c's becoming a value, but also on the second component of the data structure becoming a value. It is true that c's becoming a value cannot depend on d, but having its second component become a value does.

[2] Non-strict data constructors *can* be expressed in terms of conditionals and higher-order functions, but we have not yet shown how to handle higher-order functions. In the next section we invert this observation and use the technology we develop for data structures to handle higher-order functions.

```
{...
  a = <duple,x,y>;
  b = a;
  i = is_duple? b;
  j = sel_duple_1 b;
  k = sel_duple_2 b;
  ...}
```

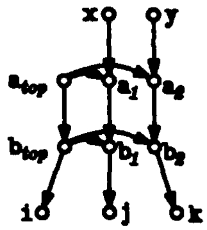
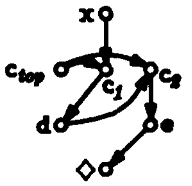


Figure 6.2: Program Illustrating Artificial Dependence Vertices

We can analyze this behavior with an artificial device that introduces a distinct vertex into the graph for each distinct piece of information carried around in the program. Suppose for a moment that the only data structure we allow is a two-tuple of integers, and consider the program fragment shown in Figure 6.2. b carries not one but three distinct pieces of data: the two components of the structure plus the information that b is of type duple. The variables i, j, and k are sensitive to different combinations of these pieces: i only requires the type of b, j requires the first component but not the second, and vice versa for k.

To model this situation in dependence graphs, we can use three vertices for each non-scalar variable, each vertex corresponding to one of the three pieces of information carried by that variable, as shown in the graph at the right of the figure. The two vertices with the *top* subscript represent "top-level" values; the shell of the data structure that just carries the information about its type. The other four subscripted vertices represent components of the data structures, with the lateral arcs reflecting the fact that a component cannot be accessed without also having the shell. In other words, for a component of b to have a value, b itself must have a value.

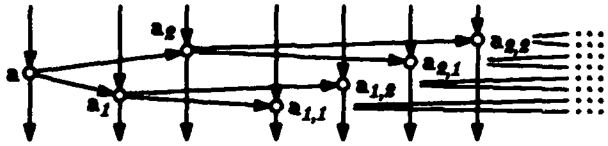
This trick of adding vertices to represent data structure components allows the dependences for each atomic piece of information to be traced separately. In the program above, for example, we see that j depends on x but not y, and that i depends on neither x nor y. If we return to the program we tried to analyze at the beginning of this section, where we arrived at the unsafe dependence graph in Figure 6.1a, the new method arrives at this dependence graph:



This graph properly reflects the dependence between d and e.

We should point out that the extra vertices added to the dependence graph are truly artificial, in the sense that they do not correspond to any expression in the original functional quads program. Consider again the program and graph in Figure 6.2. In compiled code for this program, executing the binding b = a simply fetches a pointer to a data structure from a and stores it in b. The dependence graph vertex corresponding to this computation is b_{top}, since the copying of the pointer makes the top-level shell of the structure available to consumers of b. On the other hand, b_1 and b_2 correspond to no instructions in the compiled code; the components of b become values implicitly when the arguments to the data constructor become values. In general, only the *top*-subscripted vertices will be of interest when generating code, and so once we have computed the separation and ordering constraints from the dependence graph, the artificial vertices will be discarded (this is the main reason why the definition of the approximation relation $\sqsupseteq$ allows extraneous vertices). We will return to this in greater detail when we take up code generation in Chapter 8.

Adding a vertex for each distinct value within a data structure was feasible when we restricted ourselves to two-tuples of integers, but runs into difficulties when we allow general data structures, whose components may themselves be any data structure. The problem of course, is that each variable can carry an unbounded number of atomic values, so we would need an unbounded number of artificial vertices:



Of course, this figure only illustrates the case where the only kind of data structure is a two-tuple; the situation is correspondingly more complex as the number of data types increases.

The way around this problem is to collapse the infinite set of vertices for each variable into a finite set. To do this, we need two lemmas which we illustrate pictorially in Figure 6.3. The idea is that we can combine two vertices of the dependence graph in a safe way, that is, so that the constraints obtained for the other vertices of the graph are a safe approximation to the constraints of the uncollapsed graph. A formal statement and proof of the collapsing lemmas can be found in the appendix to this chapter, Section 6.10.

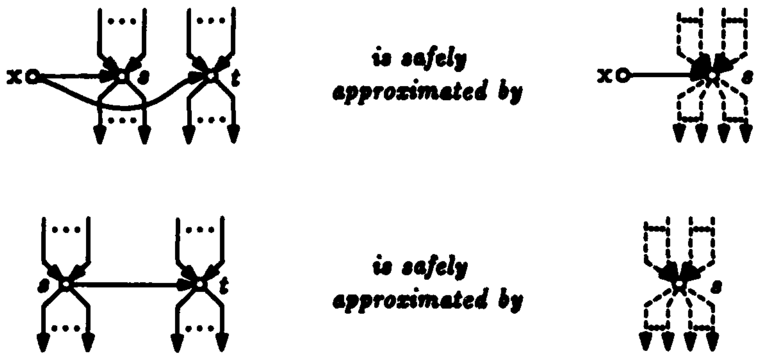


Figure 6.3: Collapsing Lemmas

One thing to note is that the edges leading to and away from collapsed vertices are always *potential* dependences, even if in the original graph some of the edges were certain dependences. This is not surprising; because we make one vertex track several independent values, there are no dependences which apply to all the values the vertex represents (an exception is illustrated in the first lemma; both collapsed vertices have a certain edge from the vertex x, and so this edge can remain as certain). It is safe for all of the edges to be potential, however, because there are always subsets which correspond to the dependences through each of the original vertices.

Of course, when we collapse vertices of the dependence graph, we inevitably lose information; in this case, we lose precision in tracking the various values contained in data structures. We are therefore faced with a design decision as to what set of artificial vertices to collapse for each variable; Figure 6.4 shows some possibilities. Type information is valuable here, for we may wish to tune the choice of artificial vertices for a variable to match the type of that variable. If a variable is of type 2-tuple, for example, then the vertex set shown in Figure 6.4c seems logical: we track each component separately, collapsing subcomponents if the components are themselves data structures. On the other hand, for an array type it is infeasible to track each component separately since the number of components is not known until run time, and in fact may differ from one invocation of the function to the next; Figure 6.4b seems the appropriate choice.[3] Recursive types such as lists might benefit from yet a different approach, such as one that collapses all "heads" into one point and all "tails" into another.[4] If no type

[3] Arrays admit a whole other class of analysis techniques generally referred to as *subscript analysis* [15]. The idea is to show that for no input can two array index expressions overlap; this information could be used to eliminate potential dependence arcs in our framework. This is a good topic for future research.

[4] This is roughly what Wadler and Hughes are doing when they analyze "head strictness" and "tail strictness" [77].

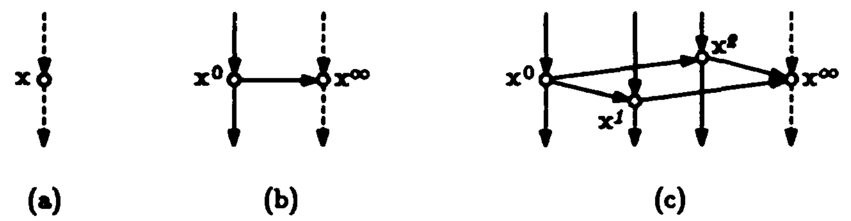


Figure 6.4: Some Possible Collapsings for Data Structure Variables

information is available, or if the language is not strongly typed, then the natural choice is two-point approximation (Figure 6.4b), which just distinguishes between a top-level value and the components. Using only a one-point approximation does not yield good results, because the collapsing lemmas end up turning all dependence arcs into potential dependences. Investigating the properties of various sets of points is a topic for future research; for the remainder of this thesis we consider only scalar/non-scalar type information and use the two-point approach.

We now present the complete set of rules for doing dependence analysis under a two-point approximation for non-scalar variables. Given a function,

$$F^{(n)}\ X_1\ \ldots\ X_n = \{B_1;\ \ldots;\ B_m;\ \mathtt{in}\ X\};$$

The two-point dependence graphs are given by:

$$\begin{aligned}
\mathcal{V} &= \{X_0,\ldots,X_n\} \cup \mathcal{LV}[B_1;\ \ldots;\ B_m] \cup \{\Diamond\} \\
\mathcal{V}_2 &= \{\, Z^0, Z^\infty \mid Z \in \mathcal{V}\,\}
\end{aligned}$$

$$\begin{aligned}
D_C &= \mathcal{DC}_2[B_1;\ \ldots;\ B_m] \cup \{(X^0,\Diamond^0)\} \cup \bigcup_{Z\in\mathcal{V}} \{(Z^0,Z^\infty)\} \\
D_F &= D_C \cup D_P \\
D_P &= \mathcal{DP}_2[B_1;\ \ldots;\ B_m] \cup \{(X^\infty,\Diamond^\infty)\}
\end{aligned}$$

$$\begin{aligned}
\vec{D}_C &= (\mathcal{V}_2, D_C) \\
\vec{D}_F &= (\mathcal{V}_2, D_F) \\
\vec{D}_P &= (\mathcal{V}_2, D_P)
\end{aligned}$$

The vertex set has two vertices for each variable: the 0-superscripted vertex represents the top-level value, and the ∞-superscripted vertex the other values. There is a certain dependence between each "top-level" vertex and the corresponding "other" vertex. If we have scalar/non-scalar type information, then we can eliminate the ∞-vertex for any variable that is known to be a scalar at compile time.

$$\begin{aligned}
\mathcal{DC}_2\llbracket B_1;\ \ldots;\ B_m\rrbracket &= \mathcal{DC}_2\llbracket B_1\rrbracket \cup\cdots\cup \mathcal{DC}_2\llbracket B_1\rrbracket\\
\mathcal{DP}_2\llbracket B_1;\ \ldots;\ B_m\rrbracket &= \mathcal{DP}_2\llbracket B_1\rrbracket \cup\cdots\cup \mathcal{DP}_2\llbracket B_1\rrbracket
\end{aligned}$$

$$\begin{aligned}
\mathcal{DC}_2\llbracket X \texttt{ = const } C\rrbracket &= \emptyset\\
\mathcal{DC}_2\llbracket X \texttt{ = } Y\rrbracket &= \{(Y^0, X^0)\}\\
\mathcal{DC}_2\llbracket X \texttt{ = } Y_1\ Op\ Y_2\rrbracket &= \{(Y_1^0, X^0), (Y_2^0, X^0)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{DP}_2\llbracket X \texttt{ = const } C\rrbracket &= \emptyset\\
\mathcal{DP}_2\llbracket X \texttt{ = } Y\rrbracket &= \{(Y^\infty, X^\infty)\}\\
\mathcal{DP}_2\llbracket X \texttt{ = } Y_1\ Op\ Y_2\rrbracket &= \emptyset
\end{aligned}$$

C denotes any scalar value. A simple binding has parallel edges between the 0 vertices and between the ∞ vertices, but the ∞ vertices must be potential because of the collapsing lemmas. The arithmetic expressions are treated in the same way as before. Conditionals are also treated as before:

$$\begin{aligned}
\mathcal{DC}_2\left\llbracket\begin{array}{l} X \texttt{ = if } Y \texttt{ then \{}B_{t,1}\texttt{;}\ldots\texttt{;}B_{t,n}\texttt{; in } Y_t\texttt{\}}\\ \qquad\qquad\quad \texttt{else \{}B_{e,1}\texttt{;}\ldots\texttt{;}B_{e,m}\texttt{; in } Y_e\texttt{\}}\end{array}\right\rrbracket &= \{(Y^0, Z^0) \mid Z \in \mathcal{LV}\llbracket B_{t,1};\ldots;B_{t,n};\rrbracket\}\cup\\
&\quad\ \{(Y^0, Z^0) \mid Z \in \mathcal{LV}\llbracket B_{e,1};\ldots;B_{e,n};\rrbracket\}\cup\\
&\quad\ \mathcal{DC}_2\llbracket B_{t,1};\ldots;B_{t,n};\rrbracket\cup\\
&\quad\ \mathcal{DC}_2\llbracket B_{e,1};\ldots;B_{e,m};\rrbracket\cup\\
&\quad\ \{(Y^0, X^0)\}\\
\mathcal{DP}_2\left\llbracket\begin{array}{l} X \texttt{ = if } Y \texttt{ then \{}B_{t,1}\texttt{;}\ldots\texttt{;}B_{t,n}\texttt{; in } Y_t\texttt{\}}\\ \qquad\qquad\quad \texttt{else \{}B_{e,1}\texttt{;}\ldots\texttt{;}B_{e,m}\texttt{; in } Y_e\texttt{\}}\end{array}\right\rrbracket &= \{(Y_t^0, X^0), (Y_e^0, X^0)\}\cup\\
&\quad\ \{(Y_t^\infty, X^\infty), (Y_e^\infty, X^\infty)\}\cup\\
&\quad\ \mathcal{DP}_2\llbracket B_{t,1};\ldots;B_{t,n};\rrbracket\cup\\
&\quad\ \mathcal{DP}_2\llbracket B_{e,1};\ldots;B_{e,m};\rrbracket
\end{aligned}$$

The interesting rules are the new ones for data structures:

$$\begin{aligned}
\mathcal{DC}_2\llbracket X \texttt{ = <}t^{(n)}\texttt{,}Y_1\texttt{,}\ldots\texttt{,}Y_n\texttt{>}\rrbracket &= \emptyset\\
\mathcal{DP}_2\llbracket X \texttt{ = <}t^{(n)}\texttt{,}Y_1\texttt{,}\ldots\texttt{,}Y_n\texttt{>}\rrbracket &= \{(Y_1^0, X^\infty), (Y_1^\infty, X^\infty), \ldots, (Y_n^0, X^\infty), (Y_n^\infty, X^\infty)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{DC}_2\llbracket X \texttt{ = is\_}t\texttt{? } Y\rrbracket &= \{(Y^0, X^0)\}\\
\mathcal{DP}_2\llbracket X \texttt{ = is\_}t\texttt{? } Y\rrbracket &= \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathcal{DC}_2\llbracket X \texttt{ = sel\_}t\texttt{\_}i\ Y\rrbracket &= \{(Y^0, X^0)\}\\
\mathcal{DP}_2\llbracket X \texttt{ = sel\_}t\texttt{\_}i\ Y\rrbracket &= \{(Y^\infty, X^0), (Y^\infty, X^\infty)\}
\end{aligned}$$

To complete the definition of $\mathcal{DC}_2$ and $\mathcal{DP}_2$, we must give the rules for first-order procedure calls. As we discussed in Section 6.6, we want to achieve the effect of splicing in the graph of the callee into the graph of the caller, by somehow determining the input/output connectivity of the callee. Again, we would like to use some sort of strictness analysis technique so that recursion is handled outside the realm of graphs. But ordinary strictness analysis is no longer sufficient, for we need to distinguish between strictness in the top-level value and strictness in subcomponents.

To define the kind of strictness information we want, we need to define two auxiliary functions. The first is a predicate which identifies "top-level" values that are free of additional information (the reader may wish to review the semantic domains given in Section 4.7).

$$\mathit{toponly}(x) = \begin{cases} \mathit{true} & \text{if } x = \bot,\ x \in C,\ x = \bot_S, \text{ or } x = \langle t, \bot, \ldots, \bot \rangle, \\ \mathit{false} & \text{otherwise.} \end{cases}$$

The second removes all information save the top-level value:

$$\mathit{filter}(x) = \begin{cases} \langle t, \bot, \ldots, \bot \rangle & \text{if } x = \langle t, x_1, \ldots, x_n \rangle, \\ x & \text{otherwise.} \end{cases}$$

Thus we have $\mathit{toponly}(\mathit{filter}(x)) = \mathit{true}$ for all x. We then define five properties, P1 through P5. A function f has property P for its ith argument if the corresponding equation below is true for all $x_1, \ldots, x_n$.

$$\begin{array}{ll}
\text{P1}: & f(\ldots, \bot, \ldots) = \bot \\
\text{P2}: & \mathit{filter}(f(\ldots, \bot, \ldots)) = \mathit{filter}(f(\ldots, \mathit{filter}(x_i), \ldots)) \\
\text{P3}: & \mathit{filter}(f(\ldots, \mathit{filter}(x_i), \ldots)) = \mathit{filter}(f(\ldots, x_i, \ldots)) \\
\text{P4}: & \mathit{toponly}(f(\ldots, x_i, \ldots)) \vee (f(\ldots, \bot, \ldots) = f(\ldots, \mathit{filter}(x_i), \ldots)) \\
\text{P5}: & \mathit{toponly}(f(\ldots, x_i, \ldots)) \vee (f(\ldots, \mathit{filter}(x_i), \ldots) = f(\ldots, x_i, \ldots))
\end{array}$$

These properties have the following interpretation:

P1 The top level of the ith argument is required to produce the top level of the result.

P2 The top level of the result completely ignores the top level of the ith argument.

P3 The top level of the result does not require any subcomponent of the ith argument.

P4 No subcomponent of the result requires the top level of the ith argument.

P5 No subcomponent of the result requires any subcomponent of the ith argument.

(Properties P1 and P2 reduce to ordinary strictness and ignoring, respectively, on flat domains.)

Now assume that we have a strictness analyzer which for every function of the program computes five sets corresponding to the five properties:

$$p_k(f) = \{\, i \mid \text{Property P}k \text{ is inferred for } f \text{ on its } i\text{th argument} \,\}$$

Then the rules for adding dependence edges for a first order call are:

$$\begin{aligned}
\mathcal{DC}_2[\![X \;\texttt{=}\; (F^{(n)}\ Y_1\ \ldots Y_{n-1})\ Y_n\texttt{;}]\!] &= \{\, (Y_i^0, X^0) \mid i \in p_1(f) \,\} \\
\mathcal{DP}[\![X \;\texttt{=}\; (F^{(n)}\ Y_1\ \ldots Y_{n-1})\ Y_n\texttt{;}]\!] &= \{\, (Y_i^0, X^0) \mid i \notin p_1(f) \wedge i \notin p_2(f) \,\} \cup \\
&\quad\ \{\, (Y_i^\infty, X^0) \mid i \notin p_3(f) \,\} \cup \\
&\quad\ \{\, (Y_i^0, X^\infty) \mid i \notin p_4(f) \,\} \cup \\
&\quad\ \{\, (Y_i^\infty, X^\infty) \mid i \notin p_5(f) \,\}
\end{aligned}$$

The author has successfully used the Wadler and Hughes context analysis technique [77] to infer these five properties from functional quads programs.

6.8 Higher-Order Functions

The only remaining functional quads construct is general application, that is, bindings of the form X = Y Z. The easiest way to reason about this construct is as a call to a primitive binary function *apply*, which takes a function and a value and returns the result of applying that function to that value. Without any further information, then, *apply* is a function that is strict in its first argument and may or may not use its second argument, depending on what function is being applied. Higher-order strictness analysis techniques are still in their infancy [16, 36]; they attempt to determine for each instance of *apply* whether that application will always use or always ignore its second argument.

We shall not delve further into higher-order strictness analysis, but simply present the equations for the conservative case where nothing is inferred about a general application. Operationally, *apply* examines the function value Y: if it is a partial application that needs more than one argument before the arity is satisfied, it just builds another partial application containing the new argument Z; otherwise, it invokes the function over which Y is closed, passing the arguments recorded in Y along with the new argument Z. These two cases correspond to Rules R7 and R8 in the functional quads reduction system, respectively.

The important feature to note is that the function value Y may be a partial application, and because it may be closed over some identifiers it behaves like a data structure. We therefore need to use non-flat domain techniques for general applications. In two-point analysis, a conservative treatment of *apply* behaves like a function with Property P1 for its first argument and none of properties P1 through P5 for its second. The dependence equations, therefore, are:

$$\begin{aligned}\mathcal{DC}_2[\![X = Y\ Z]\!] &= \{(Y^0, X^0)\}\\ \mathcal{DP}[\![X = Y\ Z]\!] &= \{(Y^\infty, X^0), (Y^0, X^\infty), (Y^\infty, X^\infty),\\ &\quad (Z^0, X^0), (Z^\infty, X^0), (Z^0, X^\infty), (Z^\infty, X^\infty)\}\end{aligned}$$

Even without higher-order strictness analysis, though, a compiler may be able to detect that certain applications will always have unsatisfied arity (*i.e.*, Rule R7 rewrites). In that case, the top level of X completely ignores Z, and we have the following equations:

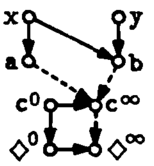
$$\begin{aligned}\mathcal{DC}_2[\![X = Y\ Z]\!] &= \{(Y^0, X^0)\}\\ \mathcal{DP}[\![X = Y\ Z]\!] &= \{(Y^\infty, X^0), (Y^0, X^\infty), (Y^\infty, X^\infty),\\ &\quad (Z^\infty, X^0), (Z^\infty, X^\infty)\}\end{aligned}$$

Choosing vertex sets other than the two-point set and incorporating higher-order strictness analysis are topics for future research.

6.9 Feedback Dependences

In Section 6.6, we remarked that in compiling code for a function G which is called from other functions, we must take into account dependences imposed on G by the context in which calls to it appear. We illustrate with the following example:

```
g x y =
  {a = x + 7;
   b = x + y;
   c = <duple,a,b>;
   in
     c}
```



We show the dependence graph as produced by the two-point rules developed above, where we have omitted the ∞-vertices for scalar variables and all the vertices for the constant 7. There is no dependence in either direction between a and b, so according to this graph, the compiler is free to place the two additions in either order within the same thread.

Now consider what happens if g is called from f as follows:

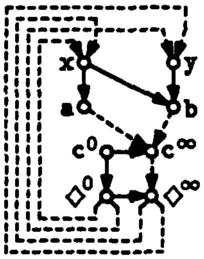
```
f x =
  {p = g x q;
   q = sel_duple_1 p;
   in
     q};
```

The first component of the tuple returned from g is fed back into g as the second argument. If we had compiled code for g which tried to compute b before a in the same thread, the code would not work properly when called from f, because there is actually a dependence from a to b when called from this context.

The most conservative way to handle this problem is to include potential dependences from the output of a function to all of its inputs; for the two-point dependence analysis, this is expressed formally as:

$$\begin{aligned} D_P \;=\; & \mathcal{DP}_2[\![B_1;\ \ldots;\ B_m]\!] \cup \{(X^\infty, \Diamond^\infty)\} \cup \\ & \{(\Diamond^0, X_i^0), (\Diamond^\infty, X_i^0), (\Diamond^0, X_i^\infty), (\Diamond^\infty, X_i^\infty) \mid 1 \le i \le n\} \end{aligned}$$

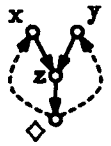
which replaces the definition of D_P given on page 116. Applying this to the example above gives the following graph:



If this graph is used to find constraints, there will be an ordering constraint from a to b, as we expect.

The reader may be distressed to find that we must add potential dependence edges from the output to every input of a function, since doing so is likely to introduce many cycles into the graph, which will lead to many separation constraints and therefore tiny sequential threads. There are one or two factors which mitigate this problem. One is that in many cases the feedback edges will be inadmissible and therefore not affect code generation. Specifically, if a function is strict in an argument, then the feedback edges to that argument will all be inadmissible:

```
f x y =
  {z = x + y;
   in
     z}
```



The only subset of the potential dependences here for which the full graph is acyclic is the empty set—both of these feedback edges are effectively removed by the definition of the constraint graphs. Another mitigating factor is that the feedback edges are not necessary if the output of a function is always a scalar. Suppose there were some argument of a function which depended on the output by virtue of dependences introduced by the caller. That argument will not be available until the output of the function has been produced. But since the output is a scalar, once it is produced there can be no new information added to it, and therefore the computation fed by the argument in question cannot contribute to the answer.

Even with these mitigating factors, that still leaves the vast majority of cases where the feedback edges will remain present to introduce cycles and induce small threads. This is not a defect in our analysis technique, but simply the price of non-strictness. We recall that in all the examples of Section 3.1, the expressive power of non-strictness was gained through the use of some sort of cyclic dependence, either around a function call, a data constructor, or a conditional. This is no coincidence; non-strictness is useful *precisely* because arguments need not be completely evaluated before a call is made, but there is no reason to delay an argument unless that argument derives some information from a portion of the value produced by the call. Hence, the feedback edges are not an unwanted nuisance, but are responsible for all the benefit the programmer gains from using a non-strict language as opposed to a strict one.

Even though feedback edges are the price of non-strictness, we certainly do not want to charge the programmer any more than necessary. That is, we should eliminate a feedback edge whenever we can show that there is no dependence from the output to that argument imposed by the caller, so that we introduce as few cycles into the dependence graph as possible. We have no results to present in this area, and simply point out that it is a research problem, perhaps the most important line of research to improve the quality of lenient compilation. There are two avenues that could be pursued. One is to analyze all callers of a function G and try to show that there are no contexts which introduce feedback dependences for some subset of G's arguments. This requires extensive interprocedural analysis, and is complicated considerably by the use of higher-order functions. On the other hand, if there are some situations which are provably feedback-free, two versions of G could be compiled, one which supports feedback and another which does not. The other avenue is to allow the programmer to tell the compiler that there are no feedback dependences for specific arguments, through the use of annotations. Both of these avenues should be explored to determine their feasibility and effectiveness.

6.10 Appendix: Proof of the "Collapsing" Lemma

In this appendix we state formally and prove the "collapsing" lemmas shown in Figure 6.3. Actually, both are just special cases of a single theorem. In that theorem, we identify two vertices s and t, and show that if t is an artificial vertex then t can be collapsed onto s as long as all edges to and from s and t are turned into potential edges, although certain edges formerly incident upon both s and t from the same origin can remain certain. We have to state this carefully: if $\langle \mathcal{V}, R_C, R_F \rangle$ is the original requirement set and $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle$ the collapsed set, we clearly cannot have $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$ because $\tilde{\mathcal{V}} \not\supseteq \mathcal{V}$. What we really want to show is that the collapsed set approximates all the constraints of the original set other than those involving the removed vertex t.

Theorem 6.5 *Let $\langle \mathcal{V}, R_C, R_F \rangle$ be a requirement set with distinguished vertices s and t, $\mathcal{V} = V \cup \{s, t\}$, and let R_C be partitioned into seven subsets according to the endpoints of each edge: $R_C = VV_C \cup VS_C \cup VT_C \cup SV_C \cup TV_C \cup ST_C \cup TS_C$, where TV_C, for example, contains the edges $(t, v) \in R_C$ for $v \in V$. Let R_F be similarly partitioned into seven subsets. Now define the requirement set where t is collapsed into s as follows:*

$$
\begin{aligned}
\tilde{\mathcal{V}} &= V \cup \{s\} \\
\tilde{R}_C &= VV_C \cup \{\, (v,s) \mid (v,s) \in VS_C \wedge (v,t) \in VT_C \,\} \\
\tilde{R}_F &= VV_F \cup VS_F \cup VS_C \cup SV_F \cup SV_C \cup \\
&\quad \{\, (v,s) \mid (v,t) \in (VT_C \cup VT_F) \,\} \cup \\
&\quad \{\, (s,v) \mid (t,v) \in (TV_C \cup TV_F) \,\}
\end{aligned}
$$

Then for the vertices in $\tilde{\mathcal{V}}$, $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$. That is, letting $CC_0(\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle) = \langle \tilde{\mathcal{V}}, \tilde{S}, \tilde{A} \rangle$ and $CC_0(\langle \mathcal{V}, R_C, R_F \rangle) = \langle \mathcal{V}, S, A \rangle$, then $\langle \tilde{\mathcal{V}}, \tilde{S}, \tilde{A} \rangle \sqsupseteq \langle \tilde{\mathcal{V}}, (S \mid \tilde{\mathcal{V}}), (A \mid \tilde{\mathcal{V}}) \rangle$.

Proof. We show that for every pair of endpoints $u, v \in \tilde{\mathcal{V}}$, the conditions of Theorem 6.1 are satisfied by $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle$. The first condition is trivially satisfied. The second is clearly satisfied since $\tilde{R}_C \subseteq R_C$. To show the third, given any path $u \xrightarrow{R}{}^{+} v$ in an acyclic subset $R \subseteq R_F$ that includes all the certain paths in R_C, we must show that there is an acyclic subset $\tilde{R} \subseteq \tilde{R}_F$ that also connects u and v and includes all certain paths in $\tilde{R}_C$. We proceed by cases, based on the vertices included along the path $u \xrightarrow{R}{}^{+} v$. Remember, this path can pass *through* the vertex t, but cannot start or end at t.

The path does not go through s or t.

The path is contained wholly in VV_F, and so is unaffected by the construction. $\tilde{R} = R$, and $\tilde{R}$ is therefore acyclic.

The path goes through s: $u \xrightarrow{R}{}^{+} s \xrightarrow{R}{}^{+} v$.

The same path still exists, and $\tilde{R} = R$. If for some reason R included edges to or from t that do not have analogs to or from s, the corresponding edges to or from s can be eliminated from $\tilde{R}$, since they cannot contribute to the path $u \xrightarrow{R}{}^{+} s \xrightarrow{R}{}^{+} v$. Hence, $\tilde{R}$ is acyclic.

The path goes through t: $u \xrightarrow{R}{}^{+} t \xrightarrow{R}{}^{+} v$.

The path will now go through s instead. A similar argument as for the last case prevents the introduction of cycles by eliminating any extraneous edges involving s.

The path goes through s then t: $u \xrightarrow{R}{}^{+} s \xrightarrow{R}{}^{+} t \xrightarrow{R}{}^{+} v$.

The corresponding path in the collapsed graph has a cycle from s to s, but that can simply be short-circuited, and the two s-edges involved in the cycle deleted from $\tilde{R}$ to make $\tilde{R}$ be acyclic.

The path goes through t then s: $u \xrightarrow{R}{}^{+} t \xrightarrow{R}{}^{+} s \xrightarrow{R}{}^{+} v$.

Analogous to the last case.

The path starts or ends with s.

Either is simply a degenerate case of the case that goes through s. ■

The proof reveals the mathematical reason why the edges to and from s and t are turned into potential edges: in various cases we needed the freedom to remove them to preserve the acyclicness of the subgraph containing a given path.

Corollary 6.6 *If $\langle \tilde{\tilde{\mathcal{V}}}, \tilde{\tilde{R}}_C, \tilde{\tilde{R}}_F \rangle$ is the requirement set obtained from performing the collapsing transformation on another requirement set $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle$, and $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$, and furthermore $\tilde{\tilde{\mathcal{V}}} \supseteq \mathcal{V}$, then $\langle \tilde{\tilde{\mathcal{V}}}, \tilde{\tilde{R}}_C, \tilde{\tilde{R}}_F \rangle \sqsupseteq \langle \mathcal{V}, R_C, R_F \rangle$, even though $\tilde{\tilde{\mathcal{V}}} \not\supseteq \tilde{\mathcal{V}}$.*

Proof. $\langle \tilde{\tilde{\mathcal{V}}}, \tilde{\tilde{R}}_C, \tilde{\tilde{R}}_F \rangle$ approximates all the constraints implied by $\langle \tilde{\mathcal{V}}, \tilde{R}_C, \tilde{R}_F \rangle$ except those involving the collapsed vertex t, but since $\tilde{\tilde{\mathcal{V}}} \supseteq \mathcal{V}$ these are irrelevant as far as $\langle \mathcal{V}, R_C, R_F \rangle$ is concerned anyway. ■

Thus, we can view the collapsed graph as an approximation, provided we have no ultimate interest in the collapsed vertex t; *i.e.*, as long as t is artificial.

Chapter 7

Constraint Computation and Partitioning

In Section 5.6 we remarked (and in Section 5.7 proved) that computing the constraint graphs S and A according to the standard constraint computation function is NP-complete. This means that, unless $P = NP$, a practical compiler must be satisfied with an algorithm which approximates S and A. Any approximate algorithm must meet the safety criteria defined in Section 5.6. In this chapter, we present such an approximate algorithm.

We first present the basic algorithm, which makes the approximation that *all* subsets of R_F are considered admissible. We then discuss situations in which this approximation does particularly badly (produces too many constraints), and offer ways of pre-processing the dependence graphs and post-processing the constraint graphs so that the approximate algorithm yields constraints closer to the theoretical ideal. An important side benefit of one of these steps is that our compilation methods will produce at least as good code as existing techniques for lazy compilation, given the same strictness analysis as input. Finally, we present an improved version of the approximate algorithm which has a faster asymptotic running time, and which also leads to a partitioning algorithm that avoids the pitfalls discussed in Section 5.5.

7.1 An Approximation to Constraint Graphs

Given that computing constraint graphs exactly is NP-complet͏e, we must search for an approximate method. The problem with the exact definition is that it must consider all subsets of R_F to determine which are admissible, introducing the exponentiality that is characteristic of NP-complete problems. So an obvious approximation is to consider *all* subsets of R_F admissible.

We call this approximation CC_A, and define it as follows.

$$CC_A(\langle \mathcal{V}, R_C, R_F\rangle) = \langle \mathcal{V}, \tilde{S}, \tilde{A}\rangle$$

$$\begin{array}{rrcl} \text{where} & \tilde{S} & = & \{\{u,v\} \mid (u,v),(v,u) \in \tilde{P}_P\} \\ & \tilde{A} & = & \{(u,v) \mid ((u,v) \in \tilde{P}_C \wedge (v,u) \notin \tilde{P}_C) \wedge \\ & & & \qquad ((u,v) \in \tilde{P}_F \wedge (v,u) \notin \tilde{P}_F)\} \\ & \tilde{P}_C & = & R_C^+ \\ & \tilde{P}_F & = & R_F^+ \\ & \tilde{P}_P & = & P_F - P_C \end{array}$$

The definition of P_C is unchanged from that in CC_0. Going back to the cases for edges in the precedence graph (Figure 5.5), there is no longer an admissibility criterion to rule out case 4, so we must interpret it. The interpretation is that if we were using the admissibility criterion, it would really be case 1, and so there should be a precedence constraint from u to v. This accounts for the modification in the definition of $\tilde{A}$.

Theorem 7.1 $CC_A \sqsupseteq CC_0$.

Proof. As in the proof of Theorem 6.1, it is helpful here to work with the precedence graphs contained in the definitions of CC_0 and CC_A. We see immediately that:

$$\begin{array}{rcll} \tilde{P}_C & = & P_C & \\ \tilde{P}_F & \supseteq & P_F, & \qquad \text{and therefore} \\ \tilde{P}_P & \supseteq & P_P & \end{array}$$

The bulk of the proof of Theorem 6.1 applies here without modification, the exception being the very last subcase, where we have $(v,u) \in \tilde{P}_F$. Given the modified definition of A in CC_A, the last subcase now follows immediately. ∎

One algorithm for computing this approximation is simply to take transitive closures of R_C and R_F. The time complexity of transitive closure is no worse than matrix multiply, or $O(n^3)$ where n is the number of vertices (this can be improved somewhat by using tricks like Strassen's, but in any event worse than $O(n^2)$) [2]. We will present a faster algorithm in Section 7.3.

7.2 Improving the Approximation

The approximation to S and A given in the last section assumes that every subset of R_F is admissible, and will infer many separation and ordering constraints that would not be present if

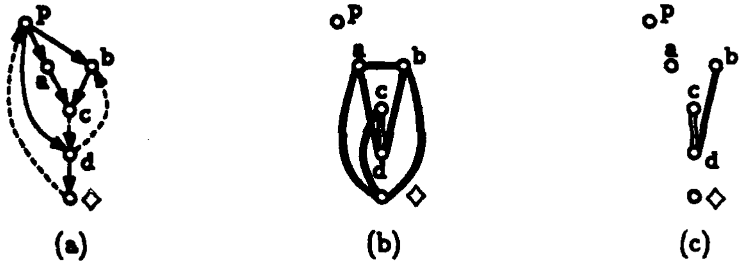


Figure 7.1: Example of Eliminating Inadmissible Edges

the inadmissible subsets were eliminated. We present two techniques which improve the quality of the approximation.

7.2.1 Eliminating Inadmissible Edges

The most obvious improvement is to recognize that while determining all of the subsets of requirements which are inadmissible is hard, there are certain potential *edges* which are themselves inadmissible. Specifically, if there is a certain path $u \xrightarrow{R_C}{}^{+} v$, then there is no admissible subset of R_F which will include (v, u). We can therefore eliminate all such edges from R_F before computing constraints. While this would have no effect if we used the exact definition of constraint graphs, it makes a significant difference when using the approximation given above. The most important way it pays off is in eliminating feedback edges to strict arguments of a function. An example is shown in Figure 7.1; (a) is a dependence graph obtained from the methods of Chapter 6, (b) is the separation graph computed from (a) by the approximation CC_A, and (c) is the separation graph computed by CC_A if the inadmissible edge $(\Diamond, p)$ is first eliminated from (a).

Theorem 7.2 *Given a requirement set $\langle \mathcal{V}, R_C, R_F\rangle$ and another in which inadmissible edges have been removed: $\langle \mathcal{V}, R_C, \tilde{R}_F\rangle$, where $\tilde{R}_F = R_F - \{(v,u) \mid u \xrightarrow{R_C}{}^{+} v\}$. Then*
(a) $CC_0(\langle \mathcal{V}, R_C, \tilde{R}_F\rangle) = CC_0(\langle \mathcal{V}, R_C, R_F\rangle)$; and
(b) $CC_A(\langle \mathcal{V}, R_C, \tilde{R}_F\rangle)$ has no more constraints than $CC_A(\langle \mathcal{V}, R_C, R_F\rangle)$.

Proof. *(a)* follows immediately from the construction of $\mathcal{R}_{ADM}$ in the definition of CC_0. *(b)* is true since eliminating edges from R_F can only reduce the size of $\tilde{P}_F$ and $\tilde{P}_P$ in the definition of CC_A, and therefore only reduce the size of $\tilde{S}$ and $\tilde{A}$. ∎

Here is an algorithm for removing all inadmissible edges with $\Theta(|\mathcal{V}|^2 + |\mathcal{V}||R_C|)$ running time.

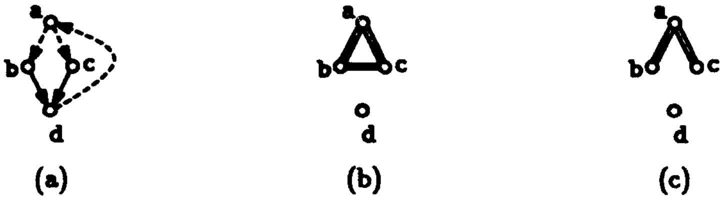


Figure 7.2: Example of Collapsing Strict Regions

```
for $u \in \mathcal{V}$ do
  $Reachable(u) \leftarrow \emptyset$
  $Mark(u) \leftarrow false$
for $u \in \mathcal{V}$ do
  if $\neg Mark(u)$ then
    DFS($u$)
for $u \in \mathcal{V}$ do
  for $v \in Reachable(u)$ do
    Remove $(v, u)$ from $R_F$

procedure DFS($u$) :
  $Mark(u) \leftarrow true$
  for each $v$ such that $(u, v) \in R_C$ do
    if $\neg Mark(v)$ do
      DFS($v$)
    $Reachable(u) \leftarrow Reachable(u) \cup Reachable(v)$
```

This is just a depth-first search of R_C; we can use depth-first search in place of the usual transitive closure algorithm because R_C is acyclic.

7.2.2 Collapsing Strict Regions

Figure 7.2 shows an example of another improvement that can be made to the approximate algorithm. Part (a) shows a requirement graph, and (b) its approximate separation graph under $\mathcal{CC}_A$. The edge between b and c is extraneous, since there are no admissible subsets of potential edges for which there is any dependence between b and c in either direction. We now describe the method of collapsing strict regions, which allows the approximate algorithm to produce the separation graph shown in (c).

Definition 7.3 *A* strict region *in a requirement set* $\langle \mathcal{V}, R_C, R_F \rangle$ *is a subset* $V \subseteq \mathcal{V}$ *with a distinguished exit vertex* $v_0 \in V$ *such that*

1. $(R_F \mid V) = (R_C \mid V)$.

2. For all $u \in V - \{v_0\}$, $u \xrightarrow{R_C}{}^{+} v_0$.

3. For any $u \in V - \{v_0\}$ there does not exist $v \in \mathcal{V} - V$ such that $(u,v) \in R_F$ or $(u,v) \in R_C$.

The first condition says that all arcs within the strict region are certain arcs (and also therefore that the strict region is acyclic). The second says that every vertex in the region has a strict path to the exit vertex. The third says that the only arcs leaving the strict region have the exit vertex v_0 as their origin (which is why we call it the exit vertex).

Theorem 7.4 *Let $V \subseteq \mathcal{V}$ be a strict region in $\langle \mathcal{V}, R_C, R_F \rangle$, and let $CC_0(\langle \mathcal{V}, R_C, R_F \rangle) = \langle \mathcal{V}, S, A \rangle$. Then for all $u, v \in V$, $\{u,v\} \notin S$ and $(u,v) \in S \iff u \xrightarrow{R_F \mid V}{}^{+} v$.*

Proof. Considering the definition of CC_0, there is no $R \in \mathcal{R}_{ADM}$ with a path $v_0 \xrightarrow{R}{}^{+} u$ for any $u \in V$; because of conditions (1) and (2) above such an R would be cyclic. But condition (3) implies that any cycle that might involve vertices in V would involve v_0, so within the strict region, P_F is acyclic. Thus $(S \mid V) = \emptyset$ and $(A \mid V) = (R_C \mid V)^+$. ∎

No matter what approximate constraint computation algorithm we happen to be using, we can use the theorem above to generate exact constraints for strict regions: there are no separation constraints, and ordering constraints are in one-to-one correspondence with paths in the original requirement graph. Alternatively, we can collapse the strict regions to a single point before constraint computation, and then use the theorem to regenerate the constraints for the region.

The term "strict regions" arises because they correspond to the sequential threads produced by existing lazy compilers that use strictness analysis. Consider the following program fragment (from the source language, not functional quads):

```
{...
 a = f0 (f1 a b) (f2 (f3 c d) e) (f4 f g);
 ...}
```

Now suppose that **f1**, **f2**, **f3**, and **f4** are each strict in both arguments, and that **f0** is strict in its first two arguments but not its third. Current lazy compilers which can infer this strictness information will insert forces and delays as follows:

```
{...
 x = delay (f0 (f1 a b) (f2 (f3 c d) e) (delay (f4 f g)));
 ...}
```

The compiled code for this fragment would place the calls to **f1**, **f2**, **f3**, and **f0** all in the same thread, with the call to **f4** in a separate thread.

```
{...
 t1 = f1 a b;
 t2 = f2 t3 e;
 t3 = f3 c d;
 t4 = f4 f g;
 x  = f0 t1 t2 t4;
 ...}
```

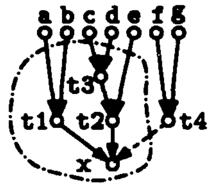


Figure 7.3: Program with a Strict Region

Now consider what a compiler using our methods would do with this program, given that it has access to the same strictness analyzer as the lazy compiler; Figure 7.3 shows the program after conversion to functional quads, and the dependence graph produced for it (to simplify the figure, we have assumed that all of the variables involved are scalar variables; the discussion remains the same even if the variables are non-scalar). Because `f0` is not strict in its third argument, the dependence from `t4` to `x` is potential, and the strictness of all other arguments involved is responsible for the other edges being certain. Now the vertices enclosed in the dot-and-dashed line form a strict region, and so with the strict region theorem we are guaranteed not to generate separation constraints between vertices in the region. We therefore can assign the calls to `f1`, `f2`, `f3`, and `f0` to the same thread, as the lazy compiler would.

In general, detecting strict regions guarantees that we will never introduce a separation constraint between two subexpressions that would be assigned to the same thread by current lazy compilers, no matter what constraint computation algorithm we use. Hence we are guaranteed of always generating as large threads as are possible using existing techniques.

There are two more strict region heuristics that warrant some discussion. The first is that an inverted version of the strict region theorem can be proved, where instead of a single exit vertex the region has a single *entry* vertex, with certain paths to every other vertex in the region. Through a similar proof, it can be shown that in these regions, too, there are no separation constraints and the only ordering constraints correspond to certain paths. The details are an exercise for the reader.[1]

[1] The earlier argument about strict regions can be viewed as an alternative justification for using strictness analysis to eliminate delays in a functional program. Mycroft [52] discusses two abstract interpretations of programs useful for eliminating delays, which he calls $E^{\#}$ and $E^{\flat}$. The first of these is what is generally termed strictness analysis, and so its use corresponds to strict regions as defined earlier. It appears that the use of $E^{\flat}$ as described by Mycroft has an analogous correspondence to inverted strict regions.

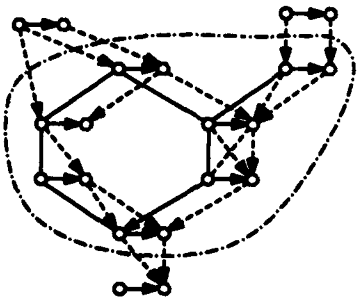


Figure 7.4: Example of a Strict Region in a Two-Point Dependence Graph

The other point is that the strict region theorems can be extended to handle two-point and other multi-point dependence analysis. In the two point scheme, the definition of a strict region would be modified to allow for other vertices in the strict region which are not necessarily connected by certain paths to the exit vertex (see Figure 7.4). The definition of a two-point strict region is slightly more complex:

Definition 7.5 *A* two-point strict region *in a requirement set* $\langle \mathcal{V}, R_C, R_F \rangle$ *is a subset* $V \subseteq \mathcal{V}$ *partitioned into two subsubsets* $V = V^0 \cup V^\infty$, *where each subsubset has a distinguished exit vertex* $v^0 \in V^0$ *and* $v^\infty \in V^\infty$, *and which satisfies:*

1. $(R_F \mid V^0) = (R_C \mid V^0)$.
2. *For all* $u \in V^0 - \{v^0\}$, $u \overset{R_C}{\longrightarrow}{}^{+} v^0$.
3. $(v^0, v^\infty) \in R_C$.
4. *For any* $u \in V - \{v^0, v^\infty\}$ *there does not exist* $v \in \mathcal{V} - V$ *such that* $(u, v) \in R_F$ *or* $(u, v) \in R_C$.

It can be shown that among the vertices in V^0, there are no separation constraints and the only ordering constraints correspond to paths in R_C. The proof is an exercise for the reader.

7.3 Improving the Running Time

The approximation CC_A we have been discussing in this chapter is defined in terms of transitive closures of the graphs obtained through dependence analysis. As we remarked earlier, we can use this definition directly and use transitive closure algorithms to compute the constraint graphs. Unfortunately, the best transitive closure algorithms are only as fast as the best matrix multiply algorithms, so the best constraint computation algorithm will have running time $O(|\mathcal{V}|^3)$ (using

Strassen's trick, this can be improved to $O(|\mathcal{V}|^{log_2 7})$, and similar tricks may reduce it further, but in any case never better than $O(|\mathcal{V}|^2))$ [2].

We now describe a faster algorithm, based on the observation that the edges in $\tilde{S}$ are contained in strongly connected components of $\vec{R}_F$.

Theorem 7.6 *Let $CC_A(\langle \mathcal{V}, R_C, R_F \rangle) = \langle \mathcal{V}, \tilde{S}; \tilde{A} \rangle$. Then $\{u,v\} \in \tilde{S}$ if and only if u and v are in some strongly connected component of R_F, $(u,v) \notin R_C^+$, and $(v,u) \notin R_C^+$.*

Proof. (If) Since u and v are in some strongly connected component of R_F, $(u,v),(v,u) \in R_F^+ = \tilde{P}_F$. $(u,v) \notin R_C^+$ and $(v,u) \notin R_C^+$ further implies that $(u,v),(v,u) \in R_F^+ - R_C^+ = P_P$. Therefore $\{u,v\} \in \tilde{S}$.

(Only if) $\{u,v\} \in \tilde{S}$ implies $(u,v),(v,u) \in R_F^+ - R_C^+$, and so in particular $(u,v),(v,u) \notin R_C^+$ and furthermore $(u,v),(v,u) \in R_F^+$ which means that u and v are in a strongly connected component. ∎

This suggests the following algorithm for computing $\tilde{S}$ and $\tilde{A}$:

1. Find the strongly connected components of $\vec{R}_F$.
2. For each pair of vertices u,v in a component, there is an edge $(u,v) \in \tilde{A}$ if $(u,v) \in R_C^+$, and $\{u,v\} \in \tilde{S}$ otherwise.
3. Form the *reduced graph* $\vec{R}'_F$ of $\vec{R}_F$ (the graph that has a vertex for each strongly connected component of R_F and an edge whenever R_F has an edge between vertices in two components). This graph is acyclic.
4. For each pair of vertex sets U, V in $\vec{R}'_F$ such that $U \xrightarrow{R'_F}{}^{+} V$, there is an edge in $\tilde{A}$ from every vertex in U to every vertex in V.

Finding the strongly connected components of a graph has running time $O(V + E)$, where V is the number of vertices and E the number of edges, and finding transitive closures of acyclic graphs has running time $O(V \cdot E)$ (using depth-first search). So the entire algorithm above has worst case running time $O(|\mathcal{V}||R_F|)$.

7.4 Avoiding New Cycles

In Section 5.4, we pointed out that the lack of an edge in both S and A between two vertices meant that there was no *a priori* constraint between them, but when placed in the same thread care must be taken to prevent the introduction of new cycles. The efficient algorithm for computing approximate constraint graphs suggests a procedure for ordering vertices so that this problem is avoided. Essentially, we will not compute $\tilde{A}$ at all, but instead we will assign numbers to every vertex such that when two vertices are assigned to the same thread they should be ordered according to the numbers we assign. We compute these numbers at the same time the separation constraint $\tilde{S}$ is constructed; afterwards, the code generator may partition the program as it chooses (consistent with $\tilde{S}$), but must observe the assigned numbers when ordering each thread.

The algorithm is follows. We assign two numbers to each vertex, called the *major* number and the *minor* number.

1. Find the strongly connected components of $\vec{R}_F$.
2. For each pair of vertices u, v in a component, there is an edge $\{u, v\} \in \tilde{S}$ unless $(u, v) \in R_C^+$.
3. For each strongly connected component, perform a topological sort using the edges of R_C, assigning minor numbers to each vertex in the component.
4. Form the reduced graph $\vec{R}'_F$ of $\vec{R}_F$ (the graph that has a vertex for each strongly connected component of R_F and an edge whenever R_F has an edge between vertices in two components). This graph is acyclic.
5. Perform a topological sort on the reduced graph, assigning a major number to all the vertices of each component.

When the program is partitioned into threads, each thread is ordered so that u precedes v only if u's major number is lower than v's, or if the major numbers are the same and u's minor number is lower than v's. This ordering is guaranteed not to introduce any additional cycles into R_F, no matter what partitioning is chosen.

Chapter 8

Code Generation

Having seen how to partition a function definition into a set of sequential threads, we turn to the task of generating actual object code based on that partition. Recall that the output of the partitioning phase is a set of "threads", where each thread is just an ordered sequence of identifiers such that each identifier corresponds to a single functional quads statement in the original function definition. At a very high level, then, object code for a function is generated by first translating each functional quads statement into a corresponding piece of sequential code and then arranging those pieces according to the sequences given in the partitioning; the resulting code sequences collectively implement the function.

There are several factors which distinguish this sort of multi-threaded code generation from the conventional code generation problem. The most obvious of these is that each function is implemented by a collection of independent code segments, which therefore must communicate and synchronize with one another. For example, if thread 1 uses the value of a which is computed by thread 2, then thread 1 must include code to wait for thread 2 to finish computing a. Moreover, the location which holds the value of a must indicate to thread 1 whether a has been computed; it must be equipped with a presence bit, and thread 2 must include code to set the presence bit when it stores a.

The second factor stems from non-strictness: because function calls and data constructors are non-strict, it may be necessary to pass values to procedures or data constructors before they have been computed. For example, to execute the constructor (cons a b), two words of storage are allocated, and a and b are copied into them. This is easy if the values of a and b have already been computed, but tricky if they have not. We must somehow provide for copying values which do not yet exist.

The third factor relates to the compilation of conditional expressions. The statements comprising the "then" or "else" arm of a conditional may not necessarily be assigned to the same thread, and even within one thread they may be interspersed with statements appearing outside the conditional. This requires a special technique for the introduction of conditional branch instructions into the sequential threads.

To accommodate these factors, we will augment the usual sequential quads notation with a few primitives that deal with presence bits, inter-thread synchronization, and the copying of values not yet computed. These aspects are explicit in the notation, to allow reasoning about optimizations such as eliminating unnecessary presence bits and synchronization; nevertheless, the notation hides the details of how these mechanisms are actually implemented. There is considerable freedom here, and the appropriate choice will depend on the target architecture as well as design decisions relating to scheduling policy and parallelism: target architectures may or may not have hardware support for the presence bits, scheduling of threads may be on a demand driven basis (execution of a thread is not begun until at least one other thread is waiting for a value computed by that thread) or on an eager basis (all threads for a function are begun when the function is invoked), and threads may execute concurrently (as on a multiprocessor) or one at a time. With our notation, all of these choices are reflected simply by choosing an appropriate implementation of the primitives in terms of the instruction set provided by the target.

The basic plan of this chapter, then, is to describe how to generate object code by first showing how to produce sequential quads augmented with some special primitives, and then illustrating the way these primitives can be implemented for a variety of combinations of hardware and scheduling policy. Some of the implementations have additional partitioning restrictions and opportunities for optimizations, and we will explore these. We will also consider lazy evaluation, and show how code which mimics lazy evaluation can be obtained from our framework by introducing still further restrictions on partitioning—a side benefit of which will be an understanding of why lenient evaluation will always yield as large or larger threads compared to lazy evaluation. Throughout, it will be clear that the quality of object code depends strongly on which partitioning is chosen from the many partitionings which satisfy the constraint graphs. One topic which we will *not* discuss is algorithms for choosing one partitioning over another; this is an important topic for future research. Nevertheless, we will draw some conclusions about what characterizes a "good" partitioning.

8.1 Code Generation Concepts

The behavior of a function is defined in terms of the reductions that take place when it is invoked in the functional quads reduction system, and so the goal of code generation is to produce target code which performs the same computations that take place during reduction. When a function is invoked in functional quads (via the rewrite rule R8), new bindings for each local variable are added to the state, and succeeding reductions eventually reduce the right hand sides of these bindings to values. At a high level, then, when the target code for a function is invoked it must allocate storage for the local variables and then compute their values. Of course, the target code for a function is actually several pieces of code, each of which is responsible for the computation of a subset of the local variables, according to the partitioning computed from the constraint graphs.

The first order of business is to use the constraint graphs to partition function being compiled. Not all of the vertices in the constraint graphs are relevant to code generation, for as we saw in Chapter 6 it is often useful to include artificial vertices to aid in tracing the effects of dependence through data structures. Once we have the constraint graphs for the function, therefore, all artificial vertices are discarded from the graphs, preserving the constraints between the vertices that remain. Specifically, we discard all but the vertices that represent "top-level" values of the local variables of the function; there is a one-to-one correspondence between these vertices and statements in the original functional quads program. The top-level vertex for $\Diamond$ is also retained, as it represents the action of returning a result to the caller.

Once irrelevant vertices have been discarded, we create a partition

$$\{(v_{1,1}, v_{1,2}, \ldots), (v_{2,1}, v_{2,2}, \ldots), \ldots\}$$

satisfying the remaining separation and ordering constraints in S and A. We will hereafter assume that such a partitioning has already been performed; as we have already discussed, this amounts to k-coloring S and topological sorting according to A, and we have outlined one such algorithm for this in Section 7.4.[1] In practice, there will be many possible partitionings of a function, and the partitioning chosen can have a large effect on the quality of object code produced. This implies that we may want to combine partitioning and code generation, or at least incorporate heuristics into partitioning that anticipate the needs of code generation. We

[1] The algorithm given in that section bypasses the construction of A, and so to use that algorithm the artificial vertices must be retained until after major and minor numbers have been assigned.

will defer this point to the end of the chapter.

Each vertex in the requirement and constraint graphs represents the action of reducing an identifier to a value, so each identifier in the partitioning will correspond to a segment of code which computes the value of that identifier. For example, suppose we were compiling this function:

```
def example x y =
  {a = ...;
   b = f + d;
   c = ...;
   d = ...;
   e = x * y;
   f = ...;
   g = const <cons,c,e>;          %% i.e., (cons c e)
   in
     g};
```

and the partitioning phase chose the following three threads: (b, g, ◇), (a, f, e), and (c, d). The object code would be the following:

```
function example (x, y)    thread 2 of example      thread 3 of example
⟨Initialization⟩           ⟨Compute value of a⟩     ⟨Compute value of c⟩
⟨Compute value of b⟩       ⟨Compute value of f⟩     ⟨Compute value of d⟩
⟨Compute value of g⟩       ⟨Compute value of e⟩     stop
⟨Return g to caller⟩       stop
stop
```

We assume that the caller of a function initiates and passes parameters to the thread which computes ◇, which we will always call thread 1. The initialization code in that thread allocates storage for the local variables a through g, and then starts up the second and third threads—exactly what this entails will be discussed later. Each of the "compute value of x" segments is the sequential quads equivalent of the corresponding functional quads statement x = *Exp.*

Some of the code segments which compute the value of a variable will contain special instructions needed to deal with multiple threads. Consider the statement b = f + d. It is not sufficient to merely include the statement b := f + d in thread 1, for when we reach it there is no guarantee that threads 2 and 3 will have progressed to the points where the values of f and d are computed. To synchronize properly, we must include presence bits in the locations allocated for the local variables, and generate the following code:

```
...
force f
force d
b $:=_v$ val(f) + val(d)
...
```

The statement *force* `f` tests the presence bit for `f` and halts execution of the thread until that presence bit turns on.[2] The *val* notation indicates that presence bits of `f` and `d` need to be stripped off before their values are added, while the v subscript in the assignment statement indicates that the presence bit for `b` should be turned on when the assignment is made (v stands for *value*). The need to force the operands of the addition is a direct consequence of the functional quads reduction system: an addition in functional quads (rule R4) must be preceded by R1b and R1c rewrites, which in turn can only take place when the operands of the addition have been reduced to values.

Another wrinkle in generating code for a functional quads statement arises because of non-strictness. Again referring to **`example`**, consider the statement **`g = const <cons,c,e>`**. To reduce **`g`** to a value we must create a two-element data structure with structure tag **`cons`** containing `c` and `e`, and as before the values of `c` and `e` may or may not have been computed yet. Unlike the earlier addition, though, it is *not* safe to force `c` and `e`; data constructors are non-strict, and must execute even if the component values are not yet evaluated. Specifically, it may not be possible to compute the value of `c` or `e` until after **`g`** has a value, so waiting for either to become a value before computing **`g`** could lead to deadlock (consider the case where the argument **`x`**, used to compute `e`, depends on the result of the call). We instead generate the following code:

```
...
temp      :=     allocate 2
temp[1]   $:=_c$ c
temp[2]   $:=_c$ e
g         $:=_v$ temp
...
```

The c-subscripted assignment is a special kind of assignment which makes the left hand side location be a *copy* of the right hand side location: executing *force* `temp[1]` will have the same effect as executing *force* `c`, and a subsequent fetch *val*(`temp[1]`) will retrieve the value of `c`.

[2] The *force* statement used here should not be confused with the `force` operator of Henderson's transformation (Section 2.2), although there is a connection between them which will become apparent later in this chapter.

Notice that this is more than merely copying the *contents* of c into temp[1]; we really mean that the two locations become equivalent for all time, so that when a value is stored into c it will effectively be stored in temp[1] as well.

The c-subscripted assignment is also used for procedure calls, for like data constructors, procedure calls are also non-strict. Thus, if the statement for g were a procedure call such as g = (foo$^{(2)}$ c) e, the code generated would be:

```
...
begincall foo
Arg$_1$  $:=_c$ c
Arg$_2$  $:=_c$ e
invoke foo
g      $:=_v$ Res
endcall foo
...
```

The notation here hides the details of the calling convention: the *begincall* primitive does whatever is needed to begin a call for foo (*e.g.*, allocating space for the arguments on the stack or in registers), *invoke* actually transfers control to foo, and *endcall* performs any necessary cleanup. Between the *begincall* and *endcall* statements are allowed the special identifiers Arg$_i$ and Res which stand for whatever locations are to be used to pass arguments and receive the result from the function being called. The choice of a particular calling convention for each function is a topic adequately covered elsewhere [66, 50]; the important feature for our purposes is that the non-evaluating $:=_c$ operator is used to pass the arguments.

The upshot of all this is that memory locations in the target implementation can contain three classes of data:

1. A *value*, such as an integer, boolean, (pointer to a) data structure, *etc.*
2. A *promise* to receive the value computed by some thread. In the example above, f initially contains a promise to receive the value computed by thread 2, and after thread 2 executes it contains a value.
3. A *copy* of some other location, such that forces and references to this location behave as if they were forces and references to the other location.

Because the target implementation will need tag bits or an equivalent mechanism to indicate which class of data a location holds, these locations will be called *tagged* locations. When a tagged location is created it may be initialized to any of the three classes, but the only

Statement	::=	*UntaggedLoc* := *Expression* |
		TaggedLoc $:=_v$ *Expression* |
		TaggedLoc $:=_p$ *Closure* |
		TaggedLoc $:=_c$ *TaggedLoc* |
		force *TaggedLoc* |
		begincall *ProcedureName* |
		invoke *ProcedureName* |
		endcall *ProcedureName* |
		goto *Label* |
		if *Operand* **goto** *Label* |
		if ¬ *Operand* **goto** *Label* |
		stop
Expression	::=	*Operand* | **allocate** *Constant* |
		Operand + *Operand* | *Operand* > *Operand* | ...
Operand	::=	*Constant* | *UntaggedLoc* | **val**(*TaggedLoc*)
UntaggedLoc	::=	*Identifier*
TaggedLoc	::=	*Identifier* | $\mathbf{arg}_i$ | **Res** | ◇ |
		UntaggedLoc[*Constant*]

Figure 8.1: Grammar of Sequential Quads

subsequent write that is allowed is when a thread stores a value in a location that previously contained a promise (a location that contains a copy, however, may be implicitly overwritten with a value when the copied location receives a value). In contrast, *untagged* locations which can only contain values may be freely used within a thread; these are just ordinary memory locations that may be read and written at will, and are typically used as temporaries (*e.g.*, the location `temp` in the earlier example). The use of tagged locations typically requires more overhead than untagged locations.

With all this in mind, the complete grammar of the sequential quads notation we will use to describe object code is given in Figure 8.1, which apart from five constructs dealing with tagged locations is pretty much standard [3]. Three of the five tagged location constructs are assignments, $:=_v$, $:=_p$, and $:=_c$, corresponding to the three classes of data that can be stored in a tagged location. The fourth is *force*, which suspends execution until the indicated location holds a value, and the fifth is *val*, which extracts the data from a location known to hold a value. To clarify the remaining notation: `a[`*c*`]` assumes that the untagged location `a` contains the address of the first word of a contiguous block of memory, and refers to the *c*th location beyond that word.[3] The expression *allocate c* allocates c consecutive words of tagged locations and returns the address of the first word. $\mathbf{Arg}_i$ and **Res** are only legal between a *begincall*/*endcall* pair, and correspond to the locations named by the formal parameters and $\Diamond$ in the code for the function being called. The remainder of the notation should be evident from the earlier discussion.

To generate code for a particular target machine we must choose a representation for the three classes of tagged data (values, promises, and unevaluated copies) and convert the operators $:=_v$, $:=_p$, $:=_c$, *force*, and *val* into appropriate machine instructions. Many variations are possible, a few of which will be described in Section 8.4, but they all share the property that tagged locations are more expensive to use than untagged locations, and that storing a value in a tagged location ($:=_v$) is better than storing an unevaluated copy ($:=_c$).

8.2 Basic Code Generation

As outlined in the last section, the first cut at object code for a function is just the concatenation of code for each functional quads statement, in the order given by the partitioning. In this section we give the translations of each kind of functional quads statement into sequential quads, with examples. We will not be concerned with producing highly optimized code, as optimizations will be taken up in later sections.

Throughout this section, we will assume that Johnsson's lambda lifting transformation [42] has been performed to lift internal definitions to top level, converting their free variables to formal parameters. Our compilation method does not *require* that lambda lifting be used to handle internal definitions; for illustrative purposes it simply has the advantage of not introducing any new mechanisms for accessing variables in outer lexical scopes. A real compiler may choose to use Algol-style displays or some other environment structure, with the appropriate modifications to the schemata below.

8.2.1 First Order Constructs

The schemata for translating the majority of functional quads constructs into sequential quads are given in Figure 8.2. Most of these were discussed in the last section, and all are very straightforward. In these schemata it is assumed that n-tuple data structures are represented as a contiguous block of $(n+1)$ words, where the first word contains the structure tag and the

[3] This is slightly non-standard: in [3] it would refer to the cth consecutive location that follows a itself, as opposed to the address contained in a.

X = Y	*force* Y X :=$_v$ *val*(Y)
X = Y_1 + Y_2	*force* Y_1 *force* Y_2 X :=$_v$ *val*(Y_1) + *val*(Y_2)
X = const C	X :=$_v$ C

X = const <t,Y_1,...,Y_n>	temp := *allocate* $n+1$
	temp[0] :=$_v$ t
	temp[1] :=$_c$ Y_1
	...
	temp[n] :=$_c$ Y_n
	X :=$_v$ temp

X = sel_t_i Y	*force* Y temp := *val*(Y) *force* temp[i] X :=$_v$ *val*(temp[i])

X = is_t? Y	*force* Y
	temp1 := *val*(Y)
	temp2 := *val*(temp1[0])
	X :=$_v$ temp2 == t

X = ($F^{(n)}$ Y_1 ...Y_{n-1}) Y_n	*begincall* F Arg$_1$:=$_c$ Y_1 ... Arg$_n$:=$_c$ Y_n *invoke* f X :=$_v$ *val*(Res) *endcall* f

Figure 8.2: Basic Code Generation Schemata

remaining words the components. Depending on the type system of the source language, it may be possible in some instances to omit the structure tag slot, with the obvious modifications to the translation. Notice that the rule for X = Y forces Y; this is because the object code must reduce X to a value. On the other hand, it is probably better to simply eliminate the statement altogether by replacing all references to X by Y in the original program.

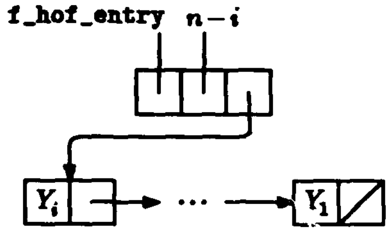
8.2.2 Higher-Order Functions

Missing from Figure 8.2 are the schemata for handling higher-order functions. There are many ways of compiling such code; we will describe a method which employs a direct representation

```
$X$ = $Y_1$ $Y_2$          force $Y_1$
                           ap        :=   val($Y_1$)
                           entry     :=   val(ap[0])
                           rem       :=   val(ap[1])
                           chn       :=   val(ap[2])
                           rdy       :=   rem == 1
                           if rdy goto L1
                           n_ap      :=   allocate 3
                           n_ap[0]   :=$_v$ entry
                           n_rem     :=   rem - 1
                           n_ap[1]   :=$_v$ n_rem
                           n_chn     :=   allocate 2
                           n_chn[0]  :=$_c$ $Y_2$
                           n_chn[1]  :=$_v$ chn
                           n_ap[2]   :=$_v$ n_chn
                           $X$         :=$_v$ n_ap
                           goto L2
                     L1:   begincall (entry)
                           Arg$_1$      :=$_v$ chn
                           Arg$_2$      :=$_c$ $Y_2$
                           invoke (entry)
                           $X$         :=$_v$ val(Res)
                           endcall (entry)
                     L2:   ...

$X$ = const ($F^{(n)}$)      temp      :=   allocate 3
                           temp[0]   :=$_v$ $F$_hof_entry
                           temp[1]   :=$_v$ $n$
                           temp[2]   :=$_v$ nil
                           $X$         :=$_v$ temp
```

Representation of
($F^{(n)}$ Y_1 ...Y_i)



H.O.F. entry code for $F^{(n)}$

```
function f_hof_entry (chn, last)
beginall f
Arg$_n$      :=$_c$ last
temp      :=   val(chn)
Arg$_{n-1}$   :=$_c$ temp[0]
temp      :=   val(temp[1])
Arg$_{n-2}$   :=$_c$ temp[0]
temp      :=   val(temp[1])
...
Arg$_1$      :=$_c$ temp[0]
invoke f
temp      :=   val(Res)
endcall f
◇         :=$_v$ temp
```

Figure 8.3: Basic Code Generation Schemata for Higher Order Functions

of partial application values, patterned after [71]. The form of a partial application object is depicted in the upper right corner of Figure 8.3: it is a 3-tuple containing the name of the function, the number of further applications needed to satisfy the arity, and the arguments already applied in the form of a linked list. Applying this to an argument does one of two things depending on whether the arity is satisfied. If it is not, then a new partial application is constructed with a decremented arity count and a new entry added to the front of the argument chain (this is why the arguments in the chain appear in reversed order). If it is, the function is invoked by sending the chain and the final argument to a special entry point created for the function, which unpacks the chain and performs an ordinary call to the function. Having a separate piece of entry code for each function allows the first-order calling convention to be customized on a per-procedure basis, while still presenting a uniform interface to the general apply in which the identity of the function is not known at compile time.

The remainder of Figure 8.3 shows sequential quads code to accomplish all this (in the code for application, some temporaries have been given names other than `temp` to aid in readability). The noteworthy aspect of the code for application is that Y_1 is forced, while Y_2 is copied unevaluated; this is consistent with the fact that in the reduction system Y_1 must be a value before either rule R7 or rule R8 applies, while Y_2 need not. Also in the figure is code for creating an empty partial application, generated when a procedure is used as a value in the source program. This could be generalized to directly compile a statement like

$$X = \texttt{const}\ (F^{(n)}\ Y_1\ \ldots\ Y_i)$$

by building the appropriate partial application structure.

8.2.3 Conditionals

The translation for the conditional statement completes the description of basic code generation. In the functional quads reduction system, conditionals serve two roles: they select one of two values to which the conditional's left hand side is bound, and they prevent the execution of bindings appearing in the arm not selected by the predicate. Thus we would like to surround the code generated for the arms of a conditional with conditional branches, but in practice this is somewhat involved because after partitioning there is no guarantee that statements taken from a given "then" or "else" arm will be assigned to the same thread, or even in contiguous segments in different threads.

The program in Figure 8.4 illustrates the problem. The value of `r` computed in the "then" arm of the conditional is fed back into the conditional through the variable `c`, computed outside the conditional. Analysis of the dependence graph shows that this program can be compiled into a single sequential thread only if `c` occurs between `r` and `s`, for example, $(\mathtt{p}, \mathtt{r}, \mathtt{t}, \mathtt{q}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{s}, \mathtt{d}, \Diamond)$. We cannot, therefore, simply generate a single conditional branch which branches around the code for `r`, `s`, and `t`, because they cannot occur contiguously.

The solution to this problem annotates the partitioning with *control strings* that indicate the control conditions governing the execution of each statement. For each variable in the

```
def cond_example x y q =
  {p = x > y;
   a = if p then
     {r = x + 8;
      s = c + 7;
      t = const <cons,r,s>;
      in
        t}
   else
     {q = const <cons,x,y>;
      in
        q};
   b = sel_cons_1 a;
   c = b + 9;
   d = sel_cons_2 a;
   in
     d};
```

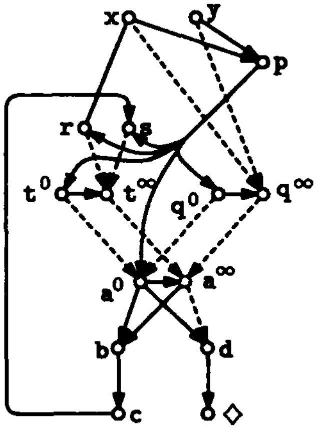


Figure 8.4: Program Illustrating Non-Contiguous Conditional

original functional quads program, we compute a control string by considering the innermost block which encloses the binding defining it:

1. If the block is not in the arm of a conditional, the control string is the empty string.
2. If the block is the "then" arm in the binding `x = if p then {...} else {...}`, the control string is Ap, where A is the control string of `x`.
3. If the block is the "else" arm in the binding `x = if p then {...} else {...}`, the control string is $A\overline{p}$, where A is the control string of `x`.

The control string of a variable `x` can be interpreted as a boolean formula which must be true in order for the binding defining `x` to be executable; the set of statements corresponding to each unique control string comprise a ***basic block*** as defined in [71].

Annotating the thread (p, r, t, q, a, b, c, s, d, ◇) with control strings gives:

```
    function cond_example (x, y)
    ⟨Initialization⟩
    ⟨Code for p = x > y⟩
p   ⟨Code for r = x + 8⟩
p   ⟨Code for t = const <cons,r,s>⟩
$\overline{p}$   ⟨Code for q = const <cons,x,y>⟩
    ⟨Code for a = if p then t else q⟩
    ⟨Code for b = sel_cons_1 a⟩
    ⟨Code for c = b + 9⟩
p   ⟨Code for s = c + 7⟩
    ⟨Code for d = sel_cons_2 a⟩
    ◇ $:=_v$ val(d)
    stop
```

Notice that the fragment corresponding to code for the conditional itself (the code for a) simply selects one variable or another as the value of a. This in turn can be expressed as a pair of assignments with control strings mutually exclusive in p:

```
    function cond_example (x, y)
    ⟨Initialization⟩
    ⟨Code for p = x > y⟩
p   ⟨Code for r = x + 8⟩
p   ⟨Code for t = const <cons,r,s>⟩
$\overline{p}$   ⟨Code for q = const <cons,x,y>⟩
p   ⟨Code for a = t⟩
$\overline{p}$   ⟨Code for a = q⟩
    ⟨Code for b = sel_cons_1 a⟩
    ⟨Code for c = b + 9⟩
p   ⟨Code for s = c + 7⟩
    ⟨Code for d = sel_cons_2 a⟩
    ◇ $:=_v$ val(d)
    stop
```

The following algorithm converts control strings to conditional branches:

1. Find a maximal group of adjacent statements whose control string begins with the same term (either x or $\overline{x}$, where x is an identifier).
2. Generate a conditional branch statement to bypass the group if the condition represented by the common term is false. A conditional branch must force the predicate variable.
3. Remove the common term from the control strings of the group's statements.
4. Repeat steps 1 through 3 until only empty control strings remain.

This algorithm ends up processing nested conditionals from the outside in. Applying it to the thread above yields:

```
     function cond_example (x, y)
     (Initialization)
     (Code for p = x > y)
     force p
     if ¬val(p) goto L1
     (Code for r = x + 8)
     (Code for t = const <cons,r,s>)
L1:  force p
     if val(p) goto L2
     (Code for q = const <cons,x,y>)
L2:  force p
     if ¬val(p) goto L3
     (Code for a = t)
L3:  force p
     if val(p) goto L4
     (Code for a = q)
L4:  (Code for b = sel_cons_1 a)
     (Code for c = b + 9)
     force p
     if ¬val(p) goto L5
     (Code for s = c + 7)
L5:  (Code for d = sel_cons_2 a)
     $\Diamond :=_v$ val(d)
     stop
```

Obviously, fewer branches are generated when the groups of adjacent statements with common control prefixes are as large as possible, and this is one criterion which should guide the partitioning phase. On the other hand, adjacent statements with mutually exclusive control strings can always be exchanged safely without referring to the constraint graphs, as there cannot possibly be any dependence between them. This is particularly helpful in placing the pair of assignment statements created for the conditional (a = t and a = q in the example above):

```
    function ...                      function ...                        function ...
    ⟨Init⟩                            ⟨Init⟩                              ⟨Init⟩
    ⟨Code for p⟩                      ⟨Code for p⟩                        ⟨Code for p⟩
p   ⟨Code for r⟩                  p   ⟨Code for r⟩                        force p
p   ⟨Code for t⟩                  p   ⟨Code for t⟩                        if ¬val(p) goto L1
p̄   ⟨Code for q⟩                  p   ⟨Code for a = t⟩                    ⟨Code for r⟩
p   ⟨Code for a = t⟩     ⟶        p̄   ⟨Code for q⟩          ⟶             ⟨Code for t⟩
p̄   ⟨Code for a = q⟩              p̄   ⟨Code for a = q⟩                    ⟨Code for a = t⟩
    ⟨Code for b⟩                      ⟨Code for b⟩                  L1:   force p
    ⟨Code for c⟩                      ⟨Code for c⟩                        if val(p) goto L2
p   ⟨Code for s⟩                  p   ⟨Code for s⟩                        ⟨Code for q⟩
    ⟨Code for d⟩                      ⟨Code for d⟩                        ⟨Code for a = q⟩
    ◊ :=_v val(d)                     ◊ :=_v val(d)                 L2:   ⟨Code for b⟩
    stop                              stop                                ⟨Code for c⟩
                                                                          force p
                                                                          if ¬val(p) goto L3
                                                                          ⟨Code for s⟩
                                                                    L3:   ⟨Code for d⟩
                                                                          ◊ :=_v val(d)
                                                                          stop
```

Flow analysis techniques [3] can be used to improve the code by removing branches or converting them to unconditional branches. Opportunities for this most often arise between adjacent groups of statements with mutually exclusive control prefixes. To illustrate:

```
       ...                                    ...
       if ¬val(p) goto L1                     if ¬val(p) goto L1
       ⟨Code for r⟩                           ⟨Code for r⟩
       ⟨Code for t⟩                           ⟨Code for t⟩
       ⟨Code for a = t⟩                       ⟨Code for a = t⟩
L1:    force p                                goto L2
       if val(p) goto L2        ⟶      L1:    ⟨Code for q⟩
       ⟨Code for q⟩                           ⟨Code for a = q⟩
       ⟨Code for a = q⟩                L2:    ⟨Code for b⟩
L2:    ⟨Code for b⟩                           ⟨Code for c⟩
       ⟨Code for c⟩                           ...
       ...
```

The control string technique works regardless of whether the arms of a conditional are contiguous or separated, in one thread or many.

8.2.4 Initialization

We assume that invoking a function initiates execution of its first thread, so it is that thread's responsibility to allocate space for and initialize the local variables and to initialize the other threads (if any). As we have discussed, the initial contents of each local variable is a promise to receive the value computed by a particular thread, the identity of the thread being determined by the partitioning. Initializing a thread entails creating an environment by which it can access local variables, formal parameters, and variables imported from enclosing lexical scopes.

To illustrate the form of initialization code, suppose we are compiling a function `f` which is partitioned into three threads: (a1, a2, ◇), (b1, b2), and (c1, c2, c3). Suppose further that it has two formal parameters p1 and p2, and imports the variable i from a surrounding scope (if lambda lifting is used, i will actually appear as a formal). The beginning of the code for thread 1 is as follows:

```
function f (p1, p2)
⟨Allocate location for a1⟩
...
⟨Allocate location for c3⟩
temp1 :=  ⟨Null Closure⟩
temp2 :=  ⟨Close thread 2 over p1, p2, i, a1, ..., c3⟩
temp3 :=  ⟨Close thread 3 over p1, p2, i, a1, ..., c3⟩
a1     $:=_p$ temp1
a2     $:=_p$ temp1
b1     $:=_p$ temp2
b2     $:=_p$ temp2
c1     $:=_p$ temp3
c2     $:=_p$ temp3
c3     $:=_p$ temp3
⟨Initiate thread 2⟩
⟨Initiate thread 3⟩
⟨Code for a1⟩
...
```

We will discuss each of the phases of initialization code in turn.

Allocating local variable locations. The first step allocates a tagged location for each local variable. We take no position on where each location is allocated—it may be in a register, on the stack, or in the heap—the only restriction is that the location must continue to exist as long as there are threads which may refer to it. A suitable lifetime analysis can be used to choose an appropriate storage class [66, 50]. We do not show allocation code for untagged (temporary) locations used within threads, as this is accommodated through standard register/stack allocation technology (see [19], for example).

Creating closures. Each of the other threads associated with a given function invocation will need access to some or all of the formal parameters, local variables, and other variables imported from enclosing lexical scopes, and so the addresses of these must be passed to the threads by creating a *closure*. In effect, a closure is a structure containing a pointer to the code for the thread along with enough pointers for that thread to gain access to all of the locations to which it refers; the latter group of pointers are collectively called the *environment*. Packaging the code pointer and environment into a closure allows the thread to be initiated at an arbitrary time in the future, as is required by demand-driven scheduling. Although the code above shows each thread closed over all formals, locals, and imports, of course it is only necessary to close a thread over the variables to which it actually refers. Again, we take no position on the layout of an environment, as these issues are adequately discussed elsewhere [50].

A point of notation: if `x` is a tagged location, then ⟨*Close thread 2 over* `x`⟩ means that the environment for thread 2 contains the *address* of `x`; thread 2 can store a value in `x`, or read it after some other thread stores a value. If `x` is an untagged location, or if we write ⟨*Close thread 2 over val*(`x`)⟩, then the environment contains only the *value* of `x`, thread 2 is limited to using the value of `x` as an operand, and within thread 2 `x` will be notated as an untagged location. Naturally, this is only possible if `x` is known to contain a value at the time the closure is built.

Storing promises and initiating threads. With the threads closed over appropriate environments, the local variables are initialized with promises (using the $:=_p$ operator) and the threads are initiated. Exactly what these two steps entail depends on the scheduling policy and on how tagged locations are implemented, as we will discuss in Section 8.4. In demand-driven implementations, for example, thread execution is initiated by the *force* operator, and the "initiate" code shown in the initialization will actually be omitted. In parallel eager implementations, on the other hand, the initiate code will begin the concurrent execution of the other threads, but the $:=_p$ operator will have a simpler implementation.

8.2.5 Examples

We now illustrate the basic code generation method with two of the programs from Section 3.1. The first of these is the **gen_fact_list** subroutine from the **make_fact_list** example in Section 3.1.2, whose functional quads equivalent and dependence graph are shown in Figure 8.5. (**Gen_fact_list** has been lambda lifted from the body of **make_fact_list**, making **fact_list** appear as a formal parameter.) Notice that feedback dependences have been introduced, but because **gen_fact_list** is strict in `i` and `n` feedback to them is inadmissible (see

```
gen_fact_list fact_list i n =
  {p = i > n;
   a =
     if p then
       {e = const <nil> in e}
     else
       {im1 = i - 1;
        prev = (nth im1) fact_list;
        this = i * prev;
        ip1 = i + 1;
        nfl = (gen_fact_list fact_list ip1) n;
        b = const <cons,this,nfl>;
        in
          b};
   in
     a}
```

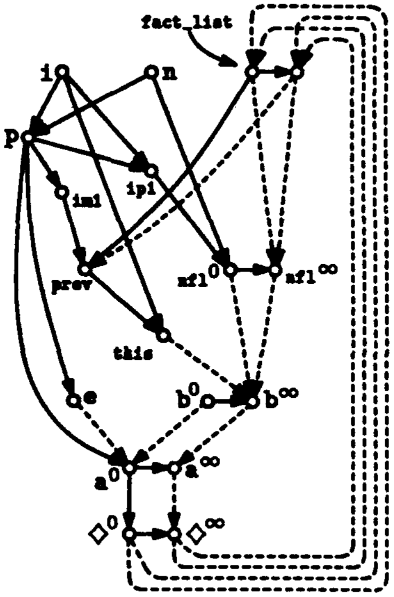


Figure 8.5: **gen_fact_list** Program and Dependence Graph

Section 6.9).

One possible partitioning of this program is into the two threads (**p**, **ip1**, **nfl**, **b**, **e**, **a**, ◇) and (**im1**, **prev**, **this**), which the reader may verify are consistent with the constraint graphs obtainable from the dependence graph (in fact, it is possible to produce a single thread, which will appear in a later section). This partitioning yields the following object code:

```
        function gen_fact_list (fact_list, i, n)
        ⟨Allocate locations for p, ip1, nfl, b, a,
         e, im1, prev, this⟩
        temp1   :=     ⟨Null Closure⟩
        temp2   :=     ⟨Close thread 2 over i, p, im1,
                        fact_list, prev, this⟩
        p       $:=_p$ temp1
        ...
        e       $:=_p$ temp1
        im1     $:=_p$ temp2
        prev    $:=_p$ temp2
        this    $:=_p$ temp2
        ⟨Initiate thread 2⟩
        force i
        force n
        p       $:=_v$ val(i) > val(n)
        force p
        if val(p) goto L1
        force i
        ip1     $:=_v$ val(i) + 1
        begincall gen_fact_list
        Arg$_1$    $:=_c$ fact_list
        Arg$_2$    $:=_c$ ip1
        Arg$_3$    $:=_c$ n
        invoke gen_fact_list
        nfl     $:=_v$ val(Res)
        endcall gen_fact_list
        temp    :=     allocate 3
        temp[0] $:=_v$ ⟨Tag for cons⟩
        temp[1] $:=_c$ this
        temp[2] $:=_c$ nfl
        b       $:=_v$ temp
        force b
        a       $:=_v$ val(b)
        goto L2
L1:     temp    :=     allocate 1
        temp[0] $:=_v$ ⟨Tag for nil⟩
        e       $:=_v$ temp
        force e
        a       $:=_v$ val(e)
L2:     force a
        ◊       $:=_v$ val(a)
        stop
```

```
        thread 2 of gen_fact_list
        force p
        if val(p) goto L1
        force i
        im1  $:=_v$ val(i) - 1
        begincall nth
        Arg$_1$ $:=_c$ im1
        Arg$_2$ $:=_c$ fact_list
        invoke nth
        prev $:=_v$ val(Res)
        endcall nth
        force i
        force prev
        this $:=_v$ val(i) * val(prev)
L1:     stop
```

In creating this code we have applied some of the branch elimination optimizations discussed earlier (in particular, the code computing e was permuted with one of the assignments to a), but no other optimizations have been applied. There are ample opportunities for eliminating redundant *force* statements and the like, as discussed in the next section.

The second program we shall consider is the conditional dependence program from Section 3.1.3, shown with its dependence graph in Figure 5.4, for which the constraint graphs appear on page 87. This program requires at least two threads, for example (p, a, aa, c, ◇) and (b, bb):

```
         function cond_examp (x)
         (Allocate locations for p, a, aa, b, bb, c)
         temp1 := (Null Closure)
         temp2 := (Close thread 2 over p, aa, b, bb)
         p      :=_p temp1
         a      :=_p temp1
         aa     :=_p temp1
         c      :=_p temp1
         b      :=_p temp2
         bb     :=_p temp2
         (Initiate thread 2)
         force x
         p      :=_v val(x) > 0
         force p
         if val(p) goto L1
         a      :=_v 3
         goto L2
L1:      force bb
         a      :=_v val(bb)
L2:      force a
         aa     :=_v val(a) + 5
         force aa
         force bb
         c      :=_v val(aa) + val(bb)
         force c
         ◇      :=_v val(c)
         stop
```

```
         thread 2 of cond_examp
         force p
         if val(p) goto L1
         force aa
         b  :=_v val(aa)
         goto L2
L1:      b  :=_v 4
L2:      force b
         bb :=_v val(b) + 6
         stop
```

8.3 Optimizations

Object code produced by the basic code generation schemata leaves much room for improvement. In addition to the usual sort of peephole optimization thąt can be applied to sequential quads, there are a number of optimizations which reduce the overhead of using tagged locations. We discuss these below.

8.3.1 Deferring Thread Initialization

The initialization code at the beginning of the first thread allocates storage for all local variables and initializes all other threads, according to Section 8.2.4. While workable, this may needlessly allocate storage for local variables defined in the arms of conditionals, some of which will never be used depending on the predicates. Similarly, threads which only compute the values of variables appearing within one arm of a conditional will do nothing if the conditional goes the other way. We can save some overhead, therefore, by conditionalizing the initialization of local variables and threads on the same predicates that control whether they will be needed at all.

Optimization 8.1 (Thread Initialization Deferment)
INSTANCE: A thread T such that the control strings of all its statements are prefixed by Ax, where A is a control string and x a control term; and the set of initialization statements S which closes thread T, stores the promises for the variables it computes, and initiates the thread.
ACTION:

1. Move the initialization statements S to a point immediately following the computation of the predicate variable **x** corresponding to the control term x (this may move S to a different thread).
2. Take Ax as the control string for each statement in S.
3. Strip the prefix Ax from each statement in T.

We described this optimization using the control strings discussed in Section 8.2.3; in terms of conditional branches the assertion that the control strings of all statements of T are prefixed by Ax says that there is a conditional branch which branches around the whole of T, Step 2 says that a branch around the statements of S is to be inserted, and Step 3 says that the branch around the whole of thread T is removed. By describing it in terms of control strings, we are allowing the conditional branch around S to be merged with similar conditional branches which are likely to follow the computation of **x**.

Applying this optimization to **gen_fact_list** yields the following:

```
function gen_fact_list (fact_list, i, n)
⟨Allocate locations for p, ip1, nfl, b, a,
 e, im1, prev, this⟩
temp1 $:=$ ⟨Null Closure⟩
p     $:=_p$ temp1
...
e     $:=_p$ temp1
force i
force n
p     $:=_v$ val(i) > val(n)
force p
if val(p) goto L1
temp2 $:=$ ⟨Close thread 2 over i, im1,
           fact_list, prev, this⟩
im1   $:=_p$ temp2
prev  $:=_p$ temp2
this  $:=_p$ temp2
⟨Initiate thread 2⟩
force i
ip1   $:=_v$ val(i) + 1
...
```

```
thread 2 of gen_fact_list
force i
im1  $:=_v$ val(i) - 1
begincall nth
Arg$_1$ $:=_c$ im1
Arg$_2$ $:=_c$ fact_list
invoke nth
prev $:=_v$ val(Res)
endcall nth
force i
force prev
this $:=_v$ val(i) * val(prev)
stop
```

Notice, too, that by eliminating the conditional branch from the second thread we eliminate all of that thread's references to p, so that p need no longer be included in thread 2's environment.

After applying Optimization 8.1, the following optimization may be used to conditionalize local variable allocation:

Optimization 8.2 (Local Variable Initialization Deferment)

INSTANCE: A local variable y whose control string is Ax, allocated by statement s, and such that the control strings of all thread initialization code which refers to y (*i.e.*, during closure creation) are prefixed by Ax.

ACTION: Move s to a point immediately following the computation of the predicate x, and take Ax as its control string.

Even though there can be no computation which refers to y before the predicate x is computed, there might be thread initialization code which does: if a thread refers to both y and some other variable z computed outside the conditional governed by x, the code which closes that thread over y and z may need to occur before x is computed. This accounts for the restriction given above.

In gen_fact_list, the variables ip1, nfl, b, e, im1, prev, and this are subject to this optimization, with e allocated if p is false, the rest if p is true:

```
      function gen_fact_list (fact_list, i, n)
      ⟨Allocate locations for p and a⟩
      temp1 :=  ⟨Null Closure⟩
      p     :=_p temp1
      a     :=_p temp1
      force i
      force n
      p     :=_v val(i) > val(n)
      force p
      if val(p) goto L1
      ⟨Allocate locations for ip1, nfl, b,
       im1, prev, this⟩
      ip1   :=_p temp1
      nfl   :=_p temp1
      b     :=_p temp1
      temp2 :=  ⟨Close thread 2 over i, im1,
                 fact_list, prev, this⟩
      im1   :=_p temp2
      prev  :=_p temp2
      this  :=_p temp2
      ⟨Initiate thread 2⟩
      force i
      ip1   :=_v val(i) + 1
      ...
      goto L2
L1:   ⟨Allocate location for e⟩
      e     :=_p temp1
      temp  :=  allocate 1
      ...
```

```
thread 2 of gen_fact_list
force i
im1  :=_v val(i) - 1
begincall nth
Arg_1 :=_c im1
Arg_2 :=_c fact_list
invoke nth
prev :=_v val(Res)
endcall nth
force i
force prev
this :=_v val(i) * val(prev)
stop
```

8.3.2 Eliminating Redundant Forces and Excess Copies

A *force* statement suspends execution of a thread until the indicated location contains a value, and for the remainder of the thread's execution that location will contain a value. Subsequent forces of that location are therefore unnecessary. Similarly, it is not necessary to force a location previously assigned with a $:=_v$ assignment. In addition to removing a *force* when a location is known to be a value, a $:=_c$ assignment can be converted to a cheaper $:=_v$ assignment if the right hand side location is known to be a value. Finally, if a thread is closed over a location known to contain a value, only the value need be included in the environment.

To describe these optimizations, we generalize slightly the notion of a *dominator* as used in conventional compilers [3]. The conventional definition says that in a sequential piece of code with conditional branches, a statement A dominates another statement B if all possible flow paths from the beginning of the code to B pass through A. Using this definition, a statement *force* x can be eliminated if dominated by another statement *force* x or by a statement x $:=_v$ *Exp*. This is a bit too restrictive, however, for it does not handle the case where all paths to a *force* x statement include another *force* x statement, but not necessarily the *same force* x statement. We therefore define *value-domination* as follows:

Definition 8.3 *In a thread, a statement A is* value-dominated for x, *where* x *is a variable, if all possible flow paths from the beginning of the thread to A pass through either a statement force* x *or* x $:=_v$ *Exp.*

If the code which initializes some thread is value-dominated for x, *then all statements in that thread are considered value-dominated for* x *as well.*

Given this definition, we give three optimizations:

Optimization 8.4 (Redundant Force Removal)

INSTANCE: A statement s of the form *force* x that is value-dominated for x.

ACTION: Remove statement s.

Optimization 8.5 (Copy Assignment Conversion)

INSTANCE: A statement s of the form *Loc* $:=_c$ x that is value-dominated for x.

ACTION: Replace s by the statement *Loc* $:=_v$ *val*(x).

Optimization 8.6 (Environment Assignment Conversion)

INSTANCE: A statement s of the form *ULoc* := ⟨*Close thread i over* ..., x, ...⟩ that is value dominated for x.

ACTION: Replace s by the statement *ULoc* := ⟨*Close thread i over* ..., *val*(x), ...⟩, and replace all occurrences of *val*(x) in thread i by just x.

Applying these optimizations to the code from the last section gives:

```
     function gen_fact_list (fact_list, i, n)
     ⟨Allocate locations for p and a⟩
     temp1     :=   ⟨Null Closure⟩
     p         $:=_p$ temp1
     a         $:=_p$ temp1
     force i
     force n
     p         $:=_v$ val(i) > val(n)
     if val(p) goto L1
     ⟨Allocate locations for ip1, nfl, b,
      im1, prev, this⟩
     ip1       $:=_p$ temp1
     nfl       $:=_p$ temp1
     b         $:=_p$ temp1
     temp2     :=   ⟨Close thread 2 over val(i),
                     im1, fact_list, prev, this⟩
     im1       $:=_p$ temp2
     prev      $:=_p$ temp2
     this      $:=_p$ temp2
     ⟨Initiate thread 2⟩
     ip1       $:=_v$ val(i) + 1
     begincall gen_fact_list
     $Arg_1$      $:=_c$ fact_list
     $Arg_2$      $:=_v$ val(ip1)
     $Arg_3$      $:=_v$ val(n)
     invoke gen_fact_list
     nfl       $:=_v$ val(Res)
     endcall gen_fact_list
     temp      :=   allocate 3
     temp[0]   $:=_v$ ⟨Tag for cons⟩
     temp[1]   $:=_c$ this
     temp[2]   $:=_v$ val(nfl)
     b         $:=_v$ temp
     a         $:=_v$ val(b)
     goto L2
'L1: ⟨Allocate location for e⟩
     e         $:=_p$ temp1
     temp      :=   allocate 1
     temp[0]   $:=_v$ ⟨Tag for nil⟩
     e         $:=_v$ temp
     a         $:=_v$ val(e)
L2:  ◇         $:=_v$ val(a)
     stop
```

```
thread 2 of gen_fact_list
im1   $:=_v$ i - 1
begincall nth
$Arg_1$  $:=_v$ val(im1)
$Arg_2$  $:=_c$ fact_list
invoke nth
prev $:=_v$ val(Res)
endcall nth
this $:=_v$ i * val(prev)
stop
```

and applying them to cond_examp gives:

```
      function cond_examp (x)
      ⟨Allocate locations for p, a, aa, b, bb, c⟩
      temp1 :=  ⟨Null Closure⟩
      temp2 :=  ⟨Close thread 2 over p, aa, b, bb⟩
      p     $:=_p$ temp1
      a     $:=_p$ temp1
      aa    $:=_p$ temp1
      c     $:=_p$ temp1
      b     $:=_p$ temp2
      bb    $:=_p$ temp2
      ⟨Initiate thread 2⟩
      force x
      p     $:=_v$ val(x) > 0
      if val(p) goto L1
      a     $:=_v$ 3
      goto L2
L1:   force bb
      a     $:=_v$ val(bb)
L2:   aa    $:=_v$ val(a) + 5
      force bb
      c     $:=_v$ val(aa) + val(bb)
      ◇     $:=_v$ val(c)
      stop
```

```
      thread 2 of cond_examp
      force p
      if val(p) goto L1
      force aa
      b  $:=_v$ val(aa)
      goto L2
L1:   b  $:=_v$ 4
L2:   bb $:=_v$ val(b) + 6
      stop
```

Notice that the second *force* bb in the first thread could not be eliminated, since it can be reached without passing through the first *force* bb.

8.3.3 Converting Tagged Locations to Untagged Locations

Tagged locations include presence bits which are needed to synchronize their use among multiple threads, and can be copied even before they have received a value. Many of a function's local variables, however, only appear in one thread, and furthermore are always assigned a value before use. These locations could just as well be untagged.

The following optimization should be applied only after applying optimizations 8.4 and 8.5.

Optimization 8.7 (Untagging)

INSTANCE: Local variable x allocated by statement *s* and which ignoring initialization code only appears in one thread, and furthermore does not appear on the right hand side of a $:=_c$ assignment.

ACTION:

1. Replace every statement of the form x $:=_v$ *Exp* by x := *Exp*.
2. Replace every occurrence of *val*(x) by just x.

3. Remove statement *s*.

4. Remove x from any statements which create thread closures (it should have only appeared in one thread's closure to begin with).

Applying this to gen_fact_list gives:

```
        function gen_fact_list (fact_list, i, n)
        temp1     :=    (Null Closure)
        force i
        force n
        p         :=    val(i) > val(n)
        if p goto L1
        (Allocate location for this)
        temp2     :=    (Close thread 2 over val(i),
                         fact_list, this)
        this      :=$_p$ temp2
        (Initiate thread 2)
        ip1       :=    val(i) + 1
        begincall gen_fact_list
        Arg$_1$      :=$_c$ fact_list
        Arg$_2$      :=$_v$ ip1
        Arg$_3$      :=$_v$ val(n)
        invoke gen_fact_list
        nfl       :=    val(Res)
        endcall gen_fact_list
        temp      :=    allocate 3
        temp[0]   :=$_v$ (Tag for cons)
        temp[1]   :=$_c$ this
        temp[2]   :=$_v$ nfl
        b         :=    temp
        a         :=    b
        goto L2
L1:     temp      :=    allocate 1
        temp[0]   :=$_v$ (Tag for nil)
        e         :=    temp
        a         :=    e
L2:     ◇         :=$_v$ a
        stop
```

```
thread 2 of gen_fact_list
im1  :=  i - 1
begincall nth
Arg$_1$ :=$_v$ im1
Arg$_2$ :=$_c$ fact_list
invoke nth
prev :=  val(Res)
endcall nth
this :=$_v$ i * prev
stop
```

and applying it to cond_example gives:

```
     function cond_examp (x)
     ⟨Allocate locations for p, aa, bb⟩
     temp1 :=  ⟨Null Closure⟩
     temp2 :=  ⟨Close thread 2 over p, aa, bb⟩
     p     $:=_p$ temp1
     aa    $:=_p$ temp1
     bb    $:=_p$ temp2
     ⟨Initiate thread 2⟩
     force x
     p     $:=_v$ val(x) > 0
     if val(p) goto L1
     a     :=  3
     goto L2
L1:  force bb
     a     :=  val(bb)
L2:  aa    $:=_v$ a + 5
     force bb
     c     :=  val(aa) + val(bb)
     ◇     $:=_v$ c
     stop
```

```
     thread 2 of cond_examp
     force p
     if val(p) goto L1
     force aa
     b  :=  val(aa)
     goto L2
L1:  b  :=  4
L2:  bb $:=_v$ b + 6
     stop
```

8.3.4 Using Strictness Analysis

If a function is strict in the top-level value of some argument, it is always safe to force that argument before making a call to that function. If we require that *every* call to a function force its strict arguments before making the call, then the function body never need force those arguments, and in fact they can be passed in untagged locations. In the worst case, this simply moves the *force* statements from the function body to the callers and allows untagged argument-passing locations. Often, however, the call will be value-dominated for some or all of the strict arguments, so that no extra forcing need be inserted in the caller, with a resulting net savings.

Optimization 8.8 (Argument Untagging)
INSTANCE: Function **f** strict in the top-level value of its *i*th argument (let **x** be the corresponding formal parameter), along with all code which makes first-order calls to **f**.
ACTION:

1. In each first-order call to **f**, replace a statement of the form $\mathtt{Arg}_i \ :=_v$ *Exp* by $\mathtt{Arg}_i$:= *Exp*.
2. In each first-order call to **f**, replace a statement of the form $\mathtt{Arg}_i \ :=_c$ *Loc* by the following two-statement sequence:

```
force Loc
Arg$_i$ := val(Loc)
```

3. In all threads comprising the body of f, remove any statement of the form *force* x.

4. In all threads comprising the body of f, replace all occurrences of *val*(x) by just x.

To illustrate, here is **gen_fact_list** again, where we note that **gen_fact_list** is strict in i and n, and **nth** is strict in all arguments.

```
      function gen_fact_list (fact_list, i, n)
      temp1    :=    ⟨Null Closure⟩
      p        :=    i > n
      if p goto L1
      ⟨Allocate location for this⟩
      temp2    :=    ⟨Close thread 2 over i, fact_list, this⟩
      this     :=$_p$   temp2
      ⟨Initiate thread 2⟩
      ip1      :=    i + 1
      begincall gen_fact_list
      Arg$_1$     :=$_c$   fact_list
      Arg$_2$     :=    ip1
      Arg$_3$     :=    n
      invoke gen_fact_list
      nfl      :=    val(Res)
      endcall gen_fact_list
      temp     :=    allocate 3
      temp[0]  :=$_v$   ⟨Tag for cons⟩
      temp[1]  :=$_c$   this
      temp[2]  :=$_v$   nfl
      b        :=    temp
      a        :=    b
      goto L2
L1:   temp     :=    allocate 1
      temp[0]  :=$_v$   ⟨Tag for nil⟩
      e        :=    temp
      a        :=    e
L2:   ◊        :=$_v$   a
      stop
```

```
thread 2 of gen_fact_list
im1  :=  i - 1
begincall nth
Arg$_1$  :=  im1
force fact_list
Arg$_2$  :=  val(fact_list)
invoke nth
prev :=  val(Res)
endcall nth
this :=$_v$ i * prev
stop
```

Notice that aside from a *force* fact_list statement in the second thread, no other additional *force* statements were needed.

We note that one of the first-order calls into which force statements must be inserted is the higher-order entry code created for the function (see Figure 8.3). Passing strict arguments as values and the entry code method for making it work in the presence of higher-order functions are due to [14].

8.3.5 Code Motion

One of the advantages of the sequential quads notation is that standard flow analysis and code motion techniques [3] can be applied to it. Here is an example of how code motion can be employed to reduce tagged location overhead.

In gen_fact_list from the last section, the tagged variable this is copied into the cons cell created in thread 1, but is not used elsewhere:

```
function gen_fact_list (fact_list, i, n)                 thread 2 of gen_fact_list
...                                                      ...
⟨Allocate location for this⟩                             this :=_v i * prev
temp2    :=   ⟨Close thread 2 over i, fact_list, this⟩   stop
this     :=_p temp2
⟨Initiate thread 2⟩
...
temp     :=   allocate 3
temp[0]  :=_v ⟨Tag for cons⟩
temp[1]  :=_c this
temp[2]  :=_v nfl
b        :=   temp
...
stop
```

This is a fairly common situation, as the conversion from the source language to functional quads introduces a variable when the argument to a data constructor is an expression, and non-strictness often requires it to be tagged. But a structure location is a perfectly good tagged location, and so we would like the second thread above to store directly into the cons cell, as follows:

```
function gen_fact_list (fact_list, i, n)              thread 2 of gen_fact_list
...                                                   ...
⟨Allocate location for b⟩                             force b
temp2    :=   ⟨Close thread 2 over i, fact_list, b⟩   temp3 :=   i * prev
⟨Initiate thread 2⟩                                   b[1]  :=_v temp3
...                                                   stop
temp     :=   allocate 3
temp[0]  :=_v ⟨Tag for cons⟩
temp[1]  :=_p temp2
temp[2]  :=_v nfl
b        :=_v temp
...
stop
```

This has eliminated the variable this, but b had to be made tagged, since thread 2 now refers to it, and is closed over it before it becomes a value. A better approach is to recognize that in the original program, the code which initializes this and thread 2 can be moved to a point just before the first reference to this:

```
function gen_fact_list (fact_list, i, n)            thread 2 of gen_fact_list
...                                                 ...
temp    $:=$ allocate 3                             this $:=_v$ i * prev
temp[0] $:=_v$ ⟨Tag for cons⟩                       stop
⟨Allocate location for this⟩
temp2   $:=$ ⟨Close thread 2 over i, fact_list, this⟩
this    $:=_p$ temp2
⟨Initiate thread 2⟩
temp[1] $:=_c$ this
temp[2] $:=_v$ nfl
b       $:=$ temp
...
stop
```

Now this can be removed by closing thread 2 over the value of temp, which holds the pointer to the structure:

```
function gen_fact_list (fact_list, i, n)            thread 2 of gen_fact_list
...                                                 ...
temp    $:=$ allocate 3                             temp3   $:=$ i * prev
temp[0] $:=_v$ ⟨Tag for cons⟩                       temp[1] $:=_v$ temp3
temp2   $:=$ ⟨Close thread 2 over i, fact_list, temp⟩   stop
temp[1] $:=_p$ temp2
⟨Initiate thread 2⟩
temp[2] $:=_v$ nfl
b       $:=$ temp
...
stop
```

The net result is the elimination of both tagged location this and the $:=_c$ assignment. We have also duplicated the behavior of Heller's L-structures [28], where a data structure slot initially points to a thread which ultimately stores a value directly into that slot.

8.4 Implementing Tagged Locations

An explanation of how to implement tagged locations completes the description of code generation. There are a limitless number of possible schemes, but we will try to illustrate a few of the more interesting ones. Each scheme is described by defining the representation of values, promises, and unevaluated copies, and giving five procedures corresponding to the five operations on tagged locations: $:=_v$, $:=_p$, $:=_c$, *force*, and *val*. To avoid being tied to any particular instruction set, we will use a "pseudo-algol" notation to describe these procedures. An example:

```
procedure $loc :=_v value$
  $[loc] \leftarrow V.value$
```

Brackets indicate indirection, so that [*loc*] means the location to which *loc* points. The contents of a tagged location are indicated as *tag.data*, so the above procedure stores the tag *V* in the tag part of the location to which *loc* points, and *value* in the data part. If the target architecture has hardware support for tagged locations this might be a single instruction, otherwise it might require some shifting and masking operations.

In taking a position on how tagged locations are to be implemented, we will also have to take a position on the thread scheduling policy, that is, when does the execution of a thread begin and end in relation to the execution of other threads. The object code itself sets boundaries on the scheduling policy: the earliest time at which thread 1 of a function can begin execution is when the function is invoked, and similarly the other threads of a function cannot begin until their initialization code has been executed. On the other hand, the latest time a thread can begin is when one of the locations it computes is forced, if the forcing thread is to make further progress. We will consider various points along this spectrum.

8.4.1 Demand-Driven Uniprocessor Implementations

By uniprocessor, we mean a conventional von Neumann machine, which can only execute one thread at a time and has no special hardware support for switching among threads. To accommodate multi-threaded code, then, we must either simulate multi-processor task switching in software, or have the threads explicitly transfer control between one another. The schemes we describe in this section all fall into the latter category, as this is likely to be the most efficient method (in any event, the multi-processor schemes described in Section 8.4.3 can all be adapted for the former approach, if desired).

The natural point at which to switch threads is at the *force* operator: forcing a location which does not yet contain a value transfers control to the thread which is to compute that value, with the forcing thread regaining control when the other thread reaches its *stop* statement. The

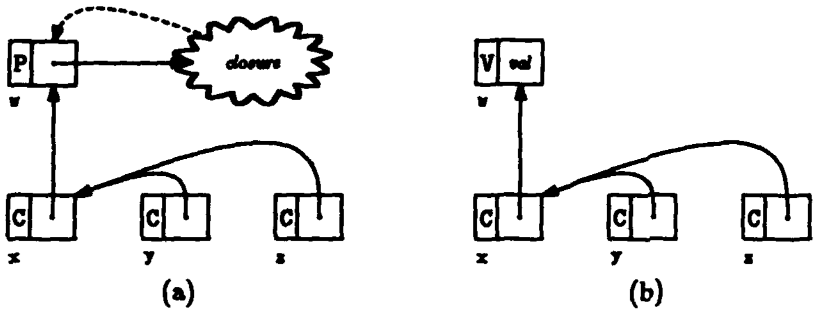


Figure 8.6: Tagged Location Scheme 1

force operator is thus a kind of procedure call, where the procedure to be called is indicated by the promise stored in the forced location.[4] The resulting scheduling is termed *demand-driven*, as a thread is not executed until another thread needs one of the values it computes. Demand-driven scheduling should not be confused with lazy evaluation; the connection between the two is the topic of Section 8.6.

In uniprocessor demand-driven schemes, a *force* statement invokes a thread through a form of procedure call, and so the *stop* statement is compiled as a return to the point at which the *force* occurred. Similarly, the *invoke* statement found in the function call schema is compiled as a call to thread 1 of the function being called, differing from *force* only in that arguments are passed and a result returned. Because threads are initiated by *force* statements, the "initiate" portion of thread initialization (Section 8.2.4) is omitted.

The procedure call behavior of *force* constrains the way threads will be interleaved at run time; in particular, the forcing thread does not regain control until the forced thread terminates. The impact of this on partitioning is discussed in Section 8.5.

Scheme 1

The simplest scheme uses three tags to indicate which of the three classes of data a tagged location contains:

V.value The value given by *value*.

P.closure A promise to have a value stored by the code to which *closure* points.

C.addr A copy of the location to which *addr* points, where that location may contain a value, promise, or another copy.

[4] In these implementations, therefore, the sequential quads *force* behaves exactly like Henderson's force.

Figure 8.6 illustrates this representation. Part (a) of the figure shows four locations after the following code sequence is executed:

```
w $:=_p$ closure
x $:=_c$ w
y $:=_c$ x
z $:=_c$ x
```

The dotted line from the closure to w indicates that the closure will have a pointer to w in its environment. Part (b) shows the same four locations after one of them is forced, and the thread has stored a value in w and terminated.

The definition of the five tagged location operations is as follows:

```
procedure loc $:=_v$ value
  [loc] $\leftarrow$ V.value

procedure loc $:=_p$ closure
  [loc] $\leftarrow$ P.closure

procedure $loc_1$ $:=_c$ $loc_2$
  [$loc_1$] $\leftarrow$ C.$loc_2$

function val(loc)
  a $\leftarrow$ followindir(loc)
  if [a] = V.value then
    return value
  else error

procedure force(loc)
  a $\leftarrow$ followindir(loc)
  if [a] = P.closure then
    call closure
```

The subroutine *followindir* follows a chain of *C* pointers and returns a pointer to the location at the end of the chain:

```
function followindir(loc)
  if [loc] = C.addr then
    return followindir(addr)
  else
    return loc
```

The *val* operator shows a test to make sure *a* points to a value. This test can be eliminated, as it only detects bugs in the compiler: a *val* operator can only be used when the indicated location is known to be a value, with the compiler inserting a *force* operator prior to it if necessary.

The assignment operators are all very cheap in this scheme, but *force* and *val* are expensive because the chains of *C* pointers may be arbitrarily long. This can be remedied by a small change to $:=_c$:

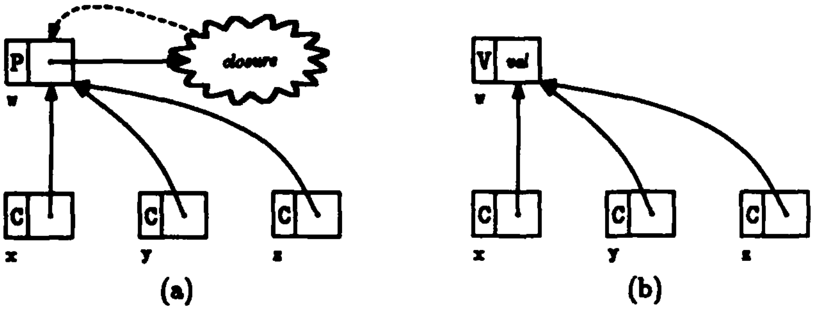


Figure 8.7: Variation on Tagged Location Scheme 1

```
**procedure** $loc_1 :=_c loc_2$
  **if** $[loc_2] = C.addr$ **then**
    $[loc_1] \leftarrow [loc_2]$
  **else**
    $[loc_1] \leftarrow C.loc_2$
```

With this modification, a location containing *C.addr* only points to locations containing either values or promises, never copies, as illustrated in Figure 8.7. The *followindir* subroutine can then be simplified to check for a single indirection:

```
**function** *followindir*(*loc*)
  **if** $[loc] = C.addr$ **then**
    **return** *addr*
  **else**
    **return** *loc*
```

Even with this variation, an indirection is created when a copy of a location containing a value is made. But because the contents of a location never change once it is assigned a value, there is no reason why the value itself cannot be copied. Again, this is a small modification to $:=_c$:

```
**procedure** $loc_1 :=_c loc_2$
  **if** $[loc_2] = P.closure$ **then**
    $[loc_1] \leftarrow C.loc_2$
  **else**
    $[loc_1] \leftarrow [loc_2]$
```

Scheme 2

In Scheme 1 and its variations, a copy of an unevaluated location is represented by an indirection, and the indirection remains even after the location to which it points receives a value.

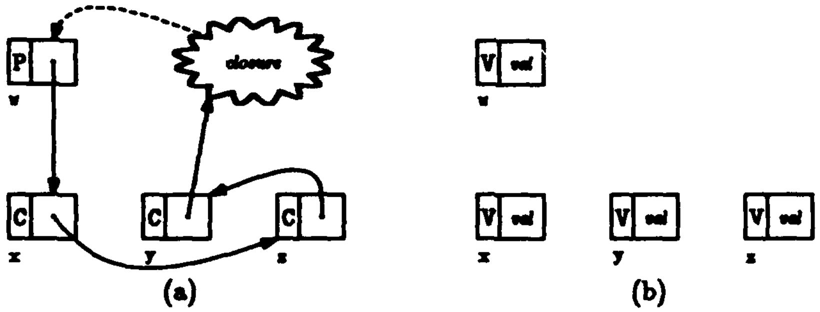


Figure 8.8: Tagged Location Scheme 2

All subsequent fetches of the copy location, therefore, must follow the indirection. This can be eliminated, however, if we arrange for the $:=_v$ operation to store the value in all locations which contain copies of the location to be stored.

One way to accomplish this would be to keep a "notifier" list with each location, containing the addresses of other locations which need copies. The operation $x :=_c w$ would add the address of x to the notifier list for w, and the operation $w :=_v$ *value* would store *value* in w and in every other location on the list. This is akin to the method for handling deferred reads in *I-structure memory* [9].

A better method requiring no additional storage is reported in [47]. The idea is to form the notifier list by chaining together the copy locations themselves, with the head of the chain stored in the location which originally contained the promise. An $:=_c$ operation simply splices in the new copy location into the chain, carrying the original closure pointer to the end of the chain, and the $:=_v$ operation stores the value in every location along the chain, beginning at the head. We again need three tags, but their interpretation differs slightly from Scheme 1:

V.value The value given by *value*.

P.closure A promise to have a value stored by the code to which *closure* points.

C.addr A promise to have a value stored by the code to which the closure found at the end of the *C* pointer chain points.

The representation is depicted in Figure 8.8 (which illustrates the same code sequences as before), and the definition of the five operations is as follows:

```
procedure $loc :=_v value$
  $old \leftarrow [loc]$
  $[loc] \leftarrow V.value$
  if $old = C.addr$ then
    $addr :=_v value$

procedure $loc :=_p closure$
  $[loc] \leftarrow P.closure$

procedure $loc_1 :=_c loc_2$
  $[loc_1] \leftarrow [loc_2]$
  if $[loc_2] \neq V.value$ then
    $[loc_2] \leftarrow C.loc_1$
```

```
function $val(loc)$
  if $[loc] = V.value$ then
    return $value$
  else error

procedure $force(loc)$
  if $[loc] = P.closure$ then
    call $closure$
  else if $[loc] = C.addr$ then
    $force(addr)$
```

Notice the recursive calls in $:=_v$ and *force*, which follow the chains as needed. The main attraction of this scheme is the simplicity of the *val* operator, which presumably is the most frequently used of the five.

8.4.2 Implementations Requiring Only Two Tags

The main drawback shared by the schemes presented in the last section is that a location may have one of three tags, resulting in fairly complex definitions of some operators tagged operators. Here we present two schemes which only require two tags. Any scheme will require at least two tags, of course, because any implementation of a non-strict programming language will have to distinguish between evaluated and unevaluated expressions.

Scheme 3

In the basic code generation schemata given in Section 8.2, the left hand sides of $:=_v$ and $:=_p$ operators were always local variables, while the left hand sides of $:=_c$ operators were always the elements of data structures or arguments to procedures.[5] If we stick to these schemata, then, local variables can only contain values or promises, and structure elements and arguments can only contain copies; locations for local variables need only one bit to encode the tag, while other locations need none at all.

The appearance of this representation is the same as in Figure 8.7, but we need to translate the $:=_c$, *force*, and *val* operators two different ways, depending on whether their operands are local variables or arguments/structure elements:

[5] The structure tag slots of data structures (offset 0) appear on the left hand sides of $:=_v$ operators, but since these locations always receive a value immediately upon creation we can treat them as untagged.

```
procedure varloc $:=_v$ value
  [varloc] ← V.value

procedure varloc $:=_p$ closure
  [varloc] ← P.closure

procedure strloc $:=_c$ varloc
  [strloc] ← C.varloc

procedure $strloc_1$ $:=_c$ $strloc_2$
  [$strloc_1$] ← [$strloc_2$]

function val(varloc)
  if [varloc] = V.value then
    return value
  else error
```

```
function val(strloc)
  if [strloc] = C.addr then
    if [addr] = V.value then
      return value
    else error
  else error

procedure force(varloc)
  if [varloc] = P.closure then
    call closure

procedure force(strloc)
  if [strloc] = C.addr then
    if [addr] = P.closure then
      call closure
  else error
```

(*varloc* refers to a local variable location, while *strloc* is a structure element or argument location.)

This scheme has the overall most efficient implementation of the five operators: there is no looping, and if error-checking is eliminated only the *force* operator tests tag bits (again, the error-checking in the code really only detects compiler bugs). The main disadvantage is that it precludes Optimization 8.5 (Copy Assignment Conversion), because it violates the restriction that structure locations and arguments can only contain copies. The code motion optimization illustrated in Section 8.3.5 is also ruled out.

This scheme is essentially that used by the Yale ALFL compiler [14], and is also the same as what goes on in a graph reduction implementation such as the G-machine LML compiler [43].

Scheme 4

Another two-tag scheme has no restrictions on what any tagged location may contain, but instead eliminates the need for a copy tag by using a promise to simulate a copy. To perform x $:=_c$ w, a small thread which forces w and stores its value into x is created, and a promise for that thread is placed in x. Forcing x, therefore, also forces w, and a subsequent *val*(x) will obtain the value of w. Since there are now only promises and values, only two tags are needed. This scheme is illustrated in Figure 8.9, and the code is as follows:

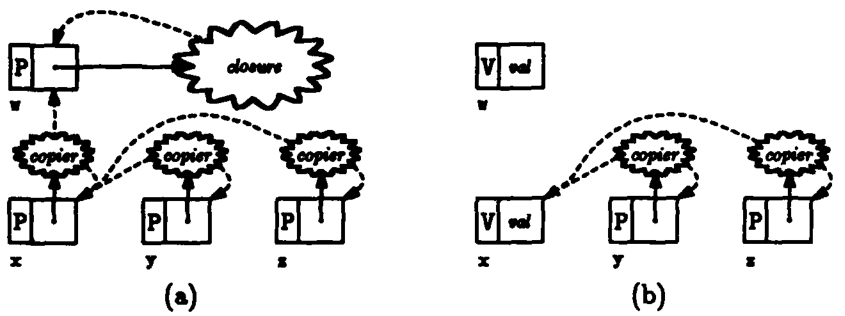


Figure 8.9: Tagged Location Scheme 4

```
procedure $loc :=_v value$
  $[loc] \leftarrow V.value$

procedure $loc :=_p closure$
  $[loc] \leftarrow P.closure$

procedure $loc_1 :=_c loc_2$
  $[loc_1] \leftarrow P.(\lambda().$
                $force(loc_2)$
                $[loc_1] \leftarrow [loc_2])$

function $val(loc)$
  if $[loc] = V.value$ then
    return $value$
  else error

procedure $force(loc)$
  if $[loc] = P.closure$ then
    call $closure$
```

A drawback of this scheme is the overhead of an extra procedure call when a copy is forced. It also fails to implement completely the definition of an unevaluated copy, which says that if **x** contains a copy of **w** then all operations performed on **x** should behave as if they were performed on **w**. In particular, if a value is stored in **w** then a subsequent *val*(**x**) should fetch **w**'s value, without having to perform a *force* operation on **x**. In the present scheme, storing a value in **w** does not affect the promise stored in **x**, so a force of **x** is needed before any attempt to do *val*(**x**). This is not a problem, however, for even with all the optimizations described earlier copies are only stored in structure elements and arguments, and we never generate code which fetches the value from such a location without first forcing it.

The code generated from Henderson's force and delay transformation [29] often resembles this scheme, for if a delayed expression is just an identifier the thread corresponding to that delay is just like a copier thread as defined above.

8.4.3 Multiprocessor Implementations

Moving from a uniprocessor to a multiprocessor implementation does not appreciably affect the way tagged data is represented; all of the schemes described above can be adapted for multiprocessors without much change. If threads are to execute concurrently, however, there must be some changes in the scheduling policy, for the policy described earlier results in only one thread executing at a time. One possibility is to retain demand-driven scheduling but relax it so that threads can start a little earlier, and so that the forcing thread can resume execution a little earlier as well. Another possibility is to abandon demand-driven scheduling in favor of eager scheduling.

Parallel Demand-Driven Scheduling

In the demand-driven schemes described for uniprocessors, a forcing thread suspends until the thread it forced terminates execution, even if the forced thread stores a value in the location long before it terminates. One way to obtain some concurrency, then, is to have the forcing thread initiate the concurrent execution of the thread named in the promise, and then wait until a value is stored in the forced location. When a value is stored, the forcing thread and the remainder of the forced thread can proceed in parallel. When a thread reaches its *stop* statement, it just dies rather than returning to the thread which initiated it.

Any of the schemes described earlier can be adapted to this scheduling policy simply by modifying the *force* operator. For scheme 1, the modification would be as follows:

```
procedure $force(loc)$
  $a \leftarrow followindir(loc)$
  if $[a] = P.closure$ then
    initiate $closure$
    wait until $[a] = V.value$
```

The *invoke* statement used for function calls needs a similar modification: it should initiate the concurrent execution of thread 1 of the called function, then wait for Res to receive a value.

Unfortunately, allowing concurrent execution of threads opens up the possibility that a second force of a location may occur between the time the first force initiates the corresponding thread and the time the location receives a value. With the code for *force* above, this results in two concurrent executions of the same thread, a wasteful and possibly hazardous occurrence. The solution is to include a bit in each closure which indicates whether the closure has already been invoked:

```
procedure force(loc)
  a ← followindir(loc)
  if [a] = P.closure then
    begin critical section
      if ¬Executing?(closure) then
        Executing?(closure) ← True
        initiate closure
    end critical section
    wait until [a] = V.value
```

Notice the use of the critical section to avoid the race that would result if two *force* statements could test the closure's bit at the same time (all that is really needed is an atomic test-and-set instruction). We have to associate the bit with the closure rather than with the forced location because the same thread can be forced through two different locations, if the thread computes more than one local variable. An analog of the executing bit can be found in every parallel demand-driven functional language implementation (see [70] and [21], as well as the *d-union* operator in [57]).

"Sparking"

The parallel demand-driven scheduling policy given above gains some concurrency, but there is still a lot of room for improvement. Consider a common code sequence like:

```
...
force y
force z
x $:=_v$ val(y) + val(z)
...
```

Under the previous policy, there will be no concurrency at all between the thread for y and the thread for z, except to the degree that the thread for y computes other values after storing the value of y. There is no reason, however, why the two threads cannot be started at the same time. To achieve this, we define an operator called *spark* that performs the initiating part of *force* but not the waiting part (the term *spark* is due to [21]):

```
procedure spark(loc)
  a ← followindir(loc)
  if [a] = P.closure then
    begin critical section
      if ¬Executing?(closure) then
        Executing?(closure) ← True
        initiate closure
    end critical section
```

Using *spark* we can rephrase the earlier code to completely overlap the computation of y and z:

```
...
spark y
spark z
force y
force z
x :=_v val(y) + val(z)
...
```

A more aggressive use of *spark* tries to move the *spark* statements to the earliest point possible while still insuring that every location that is sparked is eventually forced. Within a thread, this says that a *spark* x statement can be moved to the earliest point where it is *post-dominated* by a statement *force* x. Post-domination [4] is the mirror image of domination: a statement A is post-dominated by B if every control flow path from A to the end of the thread also includes B.

Eager Scheduling

Even with liberal use of *spark*, demand-driven scheduling policies will only initiate a thread if there is (or will be) at least one force of that thread, *i.e.*, at least one of the values the thread computes is definitely needed by some other computation. In contrast, any scheduling policy which may initiate a thread even if none of its values are ultimately used is called an *eager* policy.

The simplest eager scheduling policy initiates every thread as soon as possible, that is, immediately following the initialization code which creates the closure for it and initializes the local variables it computes. Since every thread starts executing right after creation, when a location is forced the thread which is to compute its value will already be executing concurrently, if the location does not already contain a value. This considerably simplifies the job of the *force* operator:

```
procedure force(loc)
  $a \leftarrow followindir(loc)$
  wait until $[a] = V.value$
```

The *force* operator does not even need to examine the promise stored in an unevaluated location, and so there is no reason for $:=_p$ to store a promise at all, but instead it just needs to clear the presence bit. In dataflow architectures [54, 5] and in Iannucci's architecture [40] this simplified *force* and the *val* operator are combined into a single instruction—essentially it is just a blocking read which waits for the presence bit to turn on.

Because eager scheduling may result in executing threads none of whose values are forced, it is sometimes termed "speculative" [18]. We stress, however, that it is only speculative in the sense that it may do more work than is necessary to produce the program's ultimate answer; in other words, it may do some work not done by a *lazy* evaluation of the same program. Eager scheduling does *not* do any extra work relative to *lenient* evaluation (Section 4.6), since conditional branches which prevent the execution of the unselected arm of a conditional expression are still obeyed. As we will explore in Section 8.6, even demand-driven scheduling may do extra work relative to lazy evaluation if the program is not specially partitioned for laziness.

8.5 Partitioning for Uniprocessor Demand-Driven Execution

Implementing *force* as a procedure call (Sections 8.4.1 and 8.4.2) is a very efficient method for uniprocessors, but its correct operation imposes additional constraints upon partitioning beyond those embodied in the constraint graphs. Now Theorem 5.13 shows that satisfying the constraint graphs is sufficient to guarantee that there is always an interleaving of the threads which satisfies the requirement relation for a given input to the program. Whether this interleaving is achieved, however, depends on the behavior of *force*.

During execution, the *force* operator dynamically interleaves threads, by transferring control to another thread when an uncomputed location is forced. In order to achieve an arbitrary interleaving, then, *force* must be able to transfer control to an arbitrary thread. More precisely, control transfers to a *runnable* thread, either beginning the execution of a recently initialized thread or resuming a thread which recently suspended while forcing a location that now contains a value. A *force* which suspends the current thread must be able to transfer control to at least one of these runnable threads, otherwise the computation deadlocks. Executing a *stop* statement causes a similar transfer of control.

The parallel schemes for implementing *force* (Section 8.4.3) can accommodate an arbitrary interleaving, for when a thread suspends (indicated by the **wait** statement in the definition of *force*) any of the other threads may continue in parallel. Similarly, in a uniprocessor simulation of a parallel method the suspending thread can select any of the runnable threads maintained in the simulated task queue. The uniprocessor demand-driven schemes of Sections 8.4.1 and 8.4.2, on the other hand, are much more limited: when a thread suspends it always goes to the beginning of the thread indicated by the forced location's promise, resuming only when that thread terminates. This rules out certain kinds of interleavings, as illustrated by the following program:

```
def nest x =
  {a = x + 2;
   b = a + 3;
   c = a + 4;
   d = a + 5;
   e = b + 6;
   f = e * d;
   g = f * a;
   h = f * c;
   i = h * g;
   in
     i};
```

If this program is partitioned into the two threads (e, h, i, ◇) and (a, b, c, d, f, g), the following code results:[6]

```
function nest (x)                  thread 2 of nest
⟨Initialize b, c, e, f, g⟩         force x
force b                            a :=   val(x) + 2
e  :=_v val(b) + 6                 b :=_v a + 3
force f                            c :=_v a + 4
force c                            d :=   a + 5
h  :=   val(f) * val(c)            force e
force g                            f :=_v val(e) * d
i  :=   h * val(g)                 g :=_v val(f) * a
◇  :=_v i                          stop
stop
```

Using one of the parallel schemes of Section 8.4.3, the dependence among b, e, f, and h will cause a "coroutining" execution path, as illustrated in Figure 8.10a. When the first thread starts, the

[6] Of course, this simple program would normally be partitioned into only one thread—the partitioning shown is only for purposes of illustration.

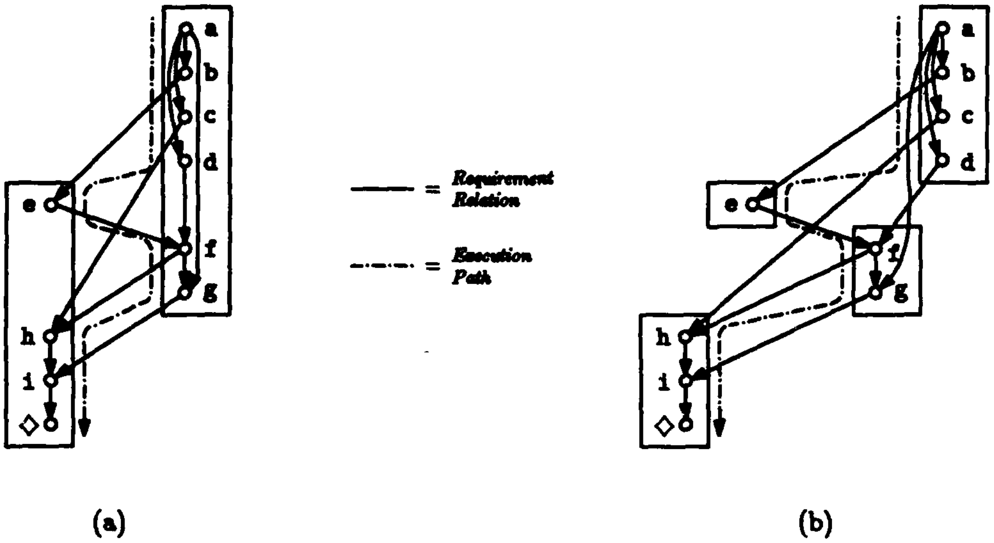


Figure 8.10: (a) "Coroutining" Interleaving; (b) Proper Partitioning for Uniprocessor Demand-Driven Implementations

force b statement immediately transfers control to the second thread, which suspends at the *force* e statement. But since b has been computed, the first thread can resume execution until it forces f, at which point the second thread can resume and terminate, which then allows the first thread to complete as well.

Now consider the execution of this program using one of the demand-driven uniprocessor schemes of Section 8.4.1. As before, the first thread suspends at the *force* b statement and calls the second thread. When the second thread reaches *force* e, there is a problem: the thread which is to compute e has already been activated, and the second thread has no way of transferring control back to it. If the second thread simply invokes the closure stored in e it will effectively create a duplicate instance of the first thread, so that e, h, i, and ◇ will be computed twice. This is clearly not satisfactory, especially considering how the duplication would multiply in the presence of recursion.

To prevent duplicated computation in uniprocessor demand-driven implementations, the *force* of an uncomputed location must always transfer control to a thread that is not already suspended. We now prove a sufficient condition on a partitioning to irsure that this is the case, but first we need to introduce some terminology to help describe execution under demand-driven uniprocessor schemes. We distinguish between *threads*, which are static pieces of code comprising function definitions, and *thread instances*, which are the dynamic instances of threads created each time a function is invoked at run time. Let l_1 be an uncomputed tagged location, which therefore contains either a promise or a copy of some other uncomputed location l_2. Define l'_1 to be either l_1 if the former, or l'_2 if the latter. In other words, l'_1 is the location found by following the chain of copies, if any, until a location containing a promise is found. When l_1 is forced, l'_1 is the location which actually appears on the left hand side of $:=_v$ in the forced thread instance. The thread instance forcing l_1 is the *parent* of the thread instance which stores l'_1, and location l'_1 is the *requestor* of the latter thread instance. If a thread computes the value of more than one tagged location, then different instances of it may have different requestors. Finally, *ancestor* is the transitive closure of parent.

Lemma 8.9 *In uniprocessor demand-driven implementations, a thread instance which forces an uncomputed location l transfers control to the thread instance containing a statement of the form l' $:=_v$ Exp.*

Proof. True because of the implementation of *force* as a procedure call.

Lemma 8.10 *Given a statement force l, let s be the next statement executed by the thread containing the force. Then in* any *implementation, uniprocessor demand-driven or not, thread instances are interleaved such that s is always executed after the statement l' $:=_v$ Exp.*

Proof. The definition of *force* says that *force l* suspends the current thread until a value is stored in l'. ∎

Theorem 8.11 *If a partitioning is consistent with the constraint graphs, and furthermore in a demand-driven uniprocessor implementation no thread instance executes a force statement after storing a value in its requestor, then no thread instance forces an ancestor.*

Proof. At any point during demand-driven uniprocessor execution, the set of thread instances consists of an active thread instance and its ancestors, where each ancestor is suspended at a *force* statement. Now suppose a thread instance (call it thread instance n) forces an ancestor n generations back by forcing the location l_A. A snapshot of the thread instances is as shown in Figure 8.11, which shows all ancestors from the instance beiɿg forced (thread instance 1) through the instance doing the forcing. The uniprocessor demand-driven implementation of force results in the execution path as shown by the dot-and-dashed ˡ:ɾe. Now the theorem says that no thread instance executes a force after storing its requestor, so the statements storing l'_2 through l'_n must follow the force statements in thread instances 2 through n. Furthermore, while l'_A is not necessarily the requestor of thread instance 1, it nevertheless must follow the

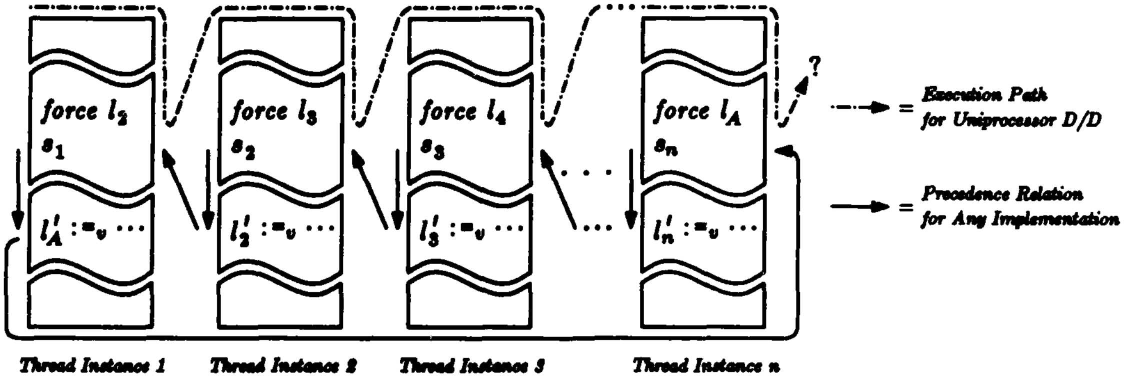


Figure 8.11: Proof of Figure 8.11

statement *force* l_2; if it did not, the *force* l_A statement in thread instance n would simply fetch a value rather than trying to force an ancestor. These precedence relations are shown by the downward-pointing solid lines in the figure. But by the previous lemma, in any interleaving of these threads execution of each statement s_i must follow the statement storing l'_{i+1}, and s_n must follow the storing of l'_A; this is indicated by the upward-pointing solid lines. No interleaving can simultaneously satisfy all of these precedence relations, so the partitioning cannot be consistent with the constraint graphs. Contradiction. ∎

On the basis of this theorem, then, to partition for uniprocessor demand-driven execution we must insure that no thread instance executes a *force* statement after storing its requestor. It is sufficient, therefore, to verify statically that no thread can execute a force statement after executing an $:=_v$ statement. This can be achieved either by a suitable partitioning algorithm, or by splitting the threads produced by the usual partitioning algorithm. We discuss the latter approach in further detail.

To split a thread for uniprocessor demand-driven execution, consider all execution paths from the beginning of the thread. If a *force* statement follows a $:=_v$ assignment, split the thread somewhere between the *force* and the $:=_v$. Note that an *invoke* statement must be treated as if it were a *force*, since it effectively forces **Res**. Continue this process on the remaining threads until no further splitting is needed. When a thread is split in two, there may be untagged local variables which are defined in the first thread and referenced in the second; these must be converted back to tagged locations by replacing the $:=$ assignment by $:=_v$, inserting *val*() as needed, and by forcing the location at the beginning of the second thread. This force is perfectly safe, as it simply reconstructs the sequential ordering of the original single thread.

Applying the splitting algorithm to the earlier example program, **nest**, yields the following four threads, whose execution is depicted in Figure 8.10b.

```
function nest (x)          thread 2 of nest    thread 3 of nest          thread 4 of nest
(Initialization)           force b             force a                   force x
force f                    e :=_v val(b) + 6   force d                   a :=_v val(x) + 2
force c                    stop                force e                   b :=_v val(a) + 3
h :=  val(f) * val(c)                          f :=_v val(e) * val(d)    c :=_v val(a) + 4
force g                                        g :=_v val(f) * val(a)    d :=_v val(a) + 5
i :=  h * val(g)                               stop                      stop
◇ :=_v i
stop
```

Notice that a and d were converted to tagged locations. There is some freedom in choosing the split points; if the second thread had been split *before* the statement computing d, then d could have remained untagged.

The preceding algorithm assumed that any $:=_v$ assignment might store the requestor of the thread. In fact, certain $:=_v$ assignments are easily shown "safe" in that they cannot possibly store the requestor. Assignments to $\mathbf{Arg}_i$ fall into this category, as do assignments which initialize structure slots, where the pointer to the structure has not been stored in a tagged location (no other thread instance could possibly have access to the structure slot). These assignments can be ignored when finding split points. Finally, we always know that the requestor of thread 1 for any function is the location ◇, and so thread 1 need only be split at points following the assignment to ◇. In the previous example, this would have allowed three threads instead of four. These improvements also show that both examples given in Section 8.3.3 will run on uniprocessor demand-driven implementations.

8.6 Lazy Evaluation

Because lazy evaluation in functional quads is just a special case of the lenient rule for selecting a redex (Section 4.6), producing object code which mimics lazy evaluation is just a special case of lenient code generation. In lazy evaluation, a variable is not reduced to a value until it is known that its value is needed for the final result of the program. If a thread is executing, therefore, it should only initiate those threads which compute values known to be required for that thread's termination; a demand-driven implementation of tagged locations must be employed. Furthermore, if we force a thread because we need the value of some variable x, that thread must not compute anything not needed to compute the value of x, for there is no guarantee that those extra values will be required by the final answer. This implies a finer partitioning beyond what is required for lenient evaluation, so that each thread only computes a single value (perhaps along with other values always needed to compute that value). In other words, each thread should only compute the value of one tagged location, and its computation must certainly require the values of all untagged locations computed in the thread.

To achieve this extra partitioning, we need only appeal to the dependence graphs which we have already computed. If a variable is untagged in a thread, then the only dependence arcs leaving the corresponding vertex of the dependence graph will be to other vertices in the same thread. Conversely, the tagged location it computes will have some outgoing arcs to vertices in other threads. To satisfy the conditions for a lazy thread, we wish to find a region in the dependence graph such that there is only one vertex with outgoing arcs to vertices outside the region, and such that all arcs contained within the region are certain arcs. These are exactly the strict regions defined in Section 7.2.2, where it was shown that there are no separation constraints between the vertices of such a region. A correct partition for lazy code is produced, therefore, by finding maximal strict regions within the dependence graph. One simple algorithm simply finds a strict region for every vertex with more than one outgoing arc or with a single outgoing potential arc; vertices with a single outgoing certain arc are included in the same region as the vertex to which they point. This corresponds to the standard delay rule used in existing lazy compilers, as it has the effect of assigning to a separate thread the argument expressions to non-strict functions and each right hand side of a letrec block.[7] A more sophisticated algorithm could potentially find larger strict regions; see the discussion in Section 7.2.2.

Once the graph has been partitioned into strict regions, it is ordered and compiled as usual. Because these lazy threads store but a single tagged location, extra splitting for uniprocessor demand-driven scheduling is never required.

8.7 Partitioning Heuristics

Throughout this section we have assumed that the graph has been partitioned and ordered before code generation takes place, but by now the reader should realize that the quality of

[7] Conditionals require special treatment in order to truly duplicate the standard delay rule. For instance, the algorithm described will always generate one or more separate threads for the arms of the conditional. The threads which compute the final value of each arm could be merged with the thread containing the left hand side of the conditional statement.

object code can vary dramatically depending on the partitioning and ordering chosen. Usually large threads are more desirable than small (this is not so clear in parallel implementations), but there are other ways that quality is affected. We list below some partitioning heuristics that can lead to improved code.

- It is generally undesirable to mix instructions from different basic blocks as it leads to excessive conditional branching. On the other hand, it *is* desirable to embed both arms of a conditional in the thread containing its left hand side.
- Vertices with only one outgoing arc should be placed in the same thread as their target so that untagged locations can be used. Preferably they should be placed as close to their successors as possible to minimize the variable's lifetime.
- Arguments of data constructors and non-strict procedures should be placed before the call to minimize the use of $:=_c$ assignments.
- The number of arcs crossing thread boundaries should be minimized, to reduce the number of tagged locations.

Finding efficient algorithms which incorporate these and other heuristics is a topic for future research. Some of these have already been addressed by others in the context of both strict and lazy evaluation [60, 61, 33].

Chapter 9

Conclusion

We have presented a method of compiling sequential code from non-strict functional languages in which non-strictness is treated separately from laziness. We take the view that the crux of prod'icing sequential code is in taking ordering decisions at compile time, and in recognizing when those decisions must be made at run time. The first step infers relationships between the subexpressions of a program by analyzing data dependences. We then convert the data dependence information into constraints upon sequential code generation, where the constraints indicate not only relative ordering of subexpressions but also which subexpressions may not be ordered at compile time. From there, we are able to pursue a variety of code generation strategies, including producing lenient code and lazy code, for a variety of target implementations. Every step of the compilation process has a counterpart in a formal system which captures the necessary constraints upon code generation, so that there is a standard by which any particular compilation algorithm can 'be judged for correctness and effectiveness. A key component of this formal system is the functional quads model of functional program execution, which has interesting properties in its own right.

We conclude by discussing the relationship of our work to other current research, and with some remarks about where our research leads.

9.1 Relationship to Other Work

To the best of our knowledge, this thesis represents the first attempt to consider non-strictness and laziness separately in the context of sequential code generation. There are, however, several research efforts which have some overlap with aspects of our work; we discuss these below.

9.1.1 Dataflow Languages and Compilers

The programming language Id [53], which arose out of dataflow work of Arvind *et al.* at MIT [5, 7], is one of the few (if not the only) examples of a functional language with non-strict, but non-lazy, semantics. We have drawn heavily from the work of this group, particularly in exploring the expressive power of non-strict non-lazy functional languages, and in compiler technology for dataflow architectures [71]. As we mentioned earlier, the operational semantics for Id given in [53] and [8] was the inspiration for functional quads, the theoretical foundation of our lenient compilation technique.

Compiling Id for dataflow architectures can be viewed as a special case of our work: in our terminology, dataflow code is concurrent, eager object code where every subexpression is assigned to a separate thread. Since every thread is of size one, dataflow compilers need not be concerned with constraints upon partitioning, nor with ordering constraints, and so dataflow compilation is largely independent of the present work. On the other hand, to control the resource requirements of dataflow programs executing on parallel architectures it is often necessary to sequentialize portions of the code [6]. The kind of dependence analysis performed here then becomes useful in insuring that such transformations preserve the semantics of the program.

9.1.2 Sarkar and Hennessy

Sarkar and Hennessy describe a technique for partitioning dataflow code into "macro-dataflow" nodes, which are essentially sequential threads [60, 61]. Their source language is the intermediate form IF1 [63], a functional language with strict semantics. Because IF1 is strict, a total ordering on the subexpressions of the program can be found at compile time based on the syntactic structure (see Section 3.4). In other words, there are no separation constraints, and they are free to partition the program at will. They develop techniques which try to minimize the amount of inter-thread communication performed by the partitioned program.

9.1.3 Serial Combinators

Hudak's serial combinators [33] represent another method for partitioning functional programs into sequential threads. Like us, Hudak starts with a non-strict language, but unlike us, Hudak only considers lazy semantics. Essentially, he starts with the threads produced by a lazy compiler, and partitions them further so that the resulting threads have no internal parallelism. He performs some analysis to balance the thread scheduling overhead against the amount of computation performed in each thread; in this respect his work is similar to Sarkar and Hennessy's above.

9.1.4 Path Semantics

The *path semantics* developed by Bloss and Hudak [13] is an alternative approach for studying the order of subexpression evaluation in functional languages. Its goals are therefore very similar to the requirement framework we developed in Chapter 5, but their work has a very different structure. Whereas requirement graphs are a partial order on the subexpressions of a program, path semantics yields a set of total orderings, each element of the set corresponding to a possible sequence of subexpression evaluations that might occur at run time. Because their representation is a set of total orderings, they can preserve the correlation information that requirement graphs do not: the set of paths computed for a function will simply not include total orderings that correspond to impossible combinations of potential requirements. On the other hand, this seems to imply that the path representation is less compact. Another difference is that while requirement graphs are defined in terms of observing the behavior of programs during execution, path semantics is defined directly in terms of axioms for primitive functions. In this respect, path semantics is most similar to the dependence graphs defined in Chapter 6, which are also based on static program properties. Currently, path semantics has no special mechanisms for dealing with data structures.

Bloss and Hudak intend path semantics to be used in a variety of optimizations for lazy functional programs, particularly the elimination of redundant forces and unnecessary delays [14]. It would be interesting to see if path semantics can serve as an alternative basis for constructing constraint graphs. One potential difficulty here is that path semantics assumes lazy, not lenient, evaluation; perhaps this could be remedied by a different choice of axioms for primitive functions. Conversely, it would be interesting to see if requirement/dependence graphs could serve as an alternative basis for the types of optimizations envisioned for path semantics.

9.2 Directions for Future Research

Throughout the thesis we have pointed out areas in which the theory or practice of lenient compilation could benefit from additional investigation. We summarize these areas here.

9.2.1 Extensions to the Formal Model

The main limitation of the requirement model developed in Chapter 5 is that it does not preserve information about *correlations* between potential requirements. In general, there will be many subsets of the potential requirements which are admissible but nevertheless cannot occur at run time. The most obvious example arises from conditionals: a conditional expression results in two potential dependence arcs, one from the output of each arm, but for no input to the program do both dependences occur simultaneously, nor for any input do neither occur.

It is not yet known whether the lack of correlation information will have a significant impact on the quality of code generated from our techniques; more experience is needed. It would also be worthwhile to see if the requirement model can be cleanly extended to include a notion of correlation. As we mentioned earlier, the path semantics of Bloss and Hudak [13] may have some relation to such a model.

9.2.2 Dependence Analysis

One of the chief strengths of our method is that the dependence graph framework is very general, in that it can exploit dependence information gained from a variety of analysis techniques. The ones we have explored in Chapter 6 are mainly simple syntactic methods, together with well-known strictness analysis techniques. Much more work is needed to investigate other ways of obtaining dependence information. We suspect that quite a bit of conventional imperative compiler technology can be harnessed, particularly subscript analysis [15] and other flow analysis techniques [3].

In Chapter 6 we have only scratched the surface in the area of dependence analysis for data structures. The multi-point approach seems very promising as it provides a unified framework in which different data types can be tracked with differing degrees of precision. Finding appropriate vertex sets for different types and developing methods for their analysis seems to be a very important area of research. We suspect that the multi-point model can also lead to a greater appreciation of the relationships between the multitude of strictness analyzers for non-flat domains that have recently proliferated [77, 39, 27].

9.2.3 Constraint Computation and Partitioning

Like most NP-complete problems, computing constraints from requirement/dependence graphs is a problem whose solutions can likely be continually improved over time. More work is needed to progress beyond the simple-minded algorithms presented here.

The formulation of code generation in terms of constraint graphs is both a blessing and a curse. On the one hand, it provides maximal freedom to a compiler, as partitioning is constrained only by what is necessary to implement the semantics, and is not influenced by the structure of the analysis which revealed those constraints (contrast this with the G-machine compiler [41, 43], in which the generation of code is inextricably entwined with the recursive descent through the source code). On the other hand, this freedom presents many more choices to the partitioning phase, which must deal with the interactions between the constraints, thread size, and inter-thread communication in choosing the best partition for a particular program. Investigating and evaluating heuristics to guide the partitioning phase presents a wide field for future effort. The work of Sarkar and Hennessy [61] may have some relevance.

9.2.4 Code Generation

The code generation techniques described in Chapter 8 are fairly straightforward, and do not attempt to take advantage of the many opportunities available for peephole optimization and the like. A lot of conventional compiler technology seems applicable here, particularly in the area of analyzing lifetimes to optimize the use of storage [50]. Most of the optimizations presented only considered program flow within a given thread. Flow analysis could be employed to analyze interactions between threads, exposing more opportunities for optimization [13]. All of these are fertile ground for further exploration.

9.2.5 I-Structures

I-structures are a non-functional language construct for array data structures which have significant advantages over purely functional arrays [8]. Although I-structures are not functional, I-structure languages can still be described by a confluent reduction system similar to functional quads (as we have mentioned, functional quads itself was actually inspired by a reduction system for I-structures), and so there is every reason to believe that the requirement graph framework developed in Chapter 5 can be extended to accommodate them.

On the other hand, dependence analysis of I-structure languages is likely to be considerably more difficult than for functional languages, because unrestricted use of I-structures leads to procedures which have side-effects. Strictness and its analysis are no longer valid criteria for deciding when the arguments to a procedure can have an effect on other computations in the caller; a different criterion that detects the relationship between argument computation and side-effects is needed (see also the footnote on page 28). With functional data structures, the computation which computes a given structure element is identifiable at the point of construction, but this is not so for I-structures. It is much more difficult, therefore, to decide the dependence relationships between various structure operations in a program. Finally, we note that because the writers of I-structure locations are not identifiable at the time of their construction, I-structure languages are *non-sequential*, and demand-driven evaluation is ruled out.

We should point out, however, that if a programming methodology is employed which restricts the non-functional uses of I-structures to the internals of a handful of functional abstractions, the remaining functional portions of a program can be analyzed exactly as if the language itself were functional. This is in fact the methodology advocated in [8].

9.3 Concluding Remarks

We believe the main contribution of our work is in providing a method of compiling functional languages which deals with non-strictness and laziness separately, so that the effects of non-strictness on object code can be isolated from the effects of laziness. This is valuable for two reasons. One is that it allows compilation which preserves non-strictness but not laziness, achieving more efficient code for programs which do not require the additional expressive power of lazy evaluation. The second is that it improves the understanding of how non-strictness and laziness are individually responsible for certain kinds of expressive power, and how they individually contribute to overhead in implementations. In particular, we have shown how a great deal of the overhead is a consequence of arguments to a function call depending on partial results from that call.

An important aspect of our work is that it tries to separate the aspects of object code quality that are traceable to a particular compilation algorithm and those that are inevitable consequences of the programming language semantics. This separation was accomplished by defining requirement graphs purely in terms of program behavior according to the operational semantics, and extracting the minimal constraints upon object code from the requirement relations. These theoretical ideals could then be used as standards by which to judge the dependence analysis, constraint computation, and partitioning algorithms used in a particular compiler. We do not claim to have provided the ultimate algorithms in any of these categories, only that their theoretical basis is sound, and that there do in fact exist algorithms which are practical and effective. We also achieve separation between the algorithms used to obtain dependence information and the algorithms which convert this information into partitioning constraints, and between the various code generation options that are possible for non-strict object code.

Finally, we believe that functional quads and its associated theory is an important step toward providing a common vantage point from which to understand a variety of functional language implementations, be they on sequential or parallel architectures, with lenient or lazy semantics, on von Neumann, dataflow, or reduction machines. Its main strength is that it exposes compilation and semantic issues such as sharing, order of evaluation, and the need to test whether an expression has become a value, while abstracting away from the details of how these are achieved in the implementation. Thus functional quads serves as a universal abstraction of functional language implementations, just as sequential quads serves as a universal abstraction of von Neumann architectures.

Bibliography

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 1(2):131–137, June 1972.

[2] A. V. Aho, J. E. Hopcraft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading MA, 1974.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1986.

[4] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19. Association for Computing Machinery, July 1970. (SIGPLAN Notices 5(7)).

[5] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[6] Arvind and D. E. Culler. Managing resources in a parallel machine. In *Fifth Generation Computer Architectures 1986*, pages 103–121. Elsevier Science Publishers B.V., 1986.

[7] Arvind and R. S. Nikhil. Executing a program on the Massachusetts Institute of Technology tagged-token dataflow architecture. Computation Structures Group Memo 271, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, March 1987.

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

[9] Arvind and R. E. Thomas. I-structures: An efficient data type for parallel machines. Technical Memo TM-178, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, September 1980.

[10] L. Augustsson. A compiler for lazy ML. *ACM SIGPLAN Notices*, 19(6):218–227, June 1984. (Proceedings of the SIGPLAN 84 Symposium on Compiler Construction).

[11] H. P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, 1984. (Revised Edition).

[12] H. P. Barendregt, J. R. Kennaway, J. W. Klop, and M. R. Sleep. Needed reduction and spine strategies for the lambda calculus. Technical Report CS-R8621, Centrum voor Wiskunde en Informatica, Amsterdam, May 1986.

[13] A. Bloss and P. Hudak. Path semantics. In *Mathematical Foundations of Programming Language Semantics (Lecture Notes in Computer Science; 298)*, pages 476–489, Berlin, April 1988. Springer-Verlag.

[14] A. Bloss, P. Hudak, and J. Young. Optimizing thunks. (Draft), 1988.

[15] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 21(6):162–175, June 1986. (Proceedings of the SIGPLAN 86 Symposium on Compiler Construction).

[16] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as Data Objects (Lecture Notes in Computer Science; 217)*, pages 42–62, Berlin, October 1985. Springer-Verlag.

[17] F. W. Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Transactions on Programming Languages and Systems*, 6(2):159–174, April 1984.

[18] F. W. Burton. Functional programming for concurrent and distributed computing. *The Computer Journal*, 30(5):437–450, October 1987.

[19] G. J. Chaitin. Register allocation & spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982. (Proceedings of the SIGPLAN 82 Symposium on Compiler Construction).

[20] C. Clack and S. L. Peyton-Jones. Strictness analysis—a practical approach. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 35–49, Berlin, September 1985. Springer-Verlag.

[21] C. Clack and S. L. Peyton-Jones. The four-stroke reduction engine. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 220–232. Association for Computing Machinery, August 1986.

[22] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on the Principles of Programming Languages*, pages 238–252. Association for Computing Machinery, January 1977.

[23] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 274)*, pages 34–45, Berlin, September 1987. Springer-Verlag.

[24] J. Fairbairn and S. C. Wray. Code generation techniques for functional languages. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 94–104. Association for Computing Machinery, August 1986.

[25] D. P. Friedman and D. S. Wise. CONS should not evaulate its arguments. In *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284, London, July 1976. Edinburgh University Press.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[27] C. V. Hall and D. S. Wise. Compiling strictness into streams. In *Conference Record of the 14th Annual ACM Symposium on the Principles of Programming Languages*, pages 132–143. Association for Computing Machinery, January 1987.

[28] S. K. Heller. *Efficient Lazy Structures in a Dataflow Machine*. PhD thesis, Massachusetts Institute of Technology, Cambridge MA, December 1988. (Expected).

[29] P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall, Englewood Cliffs NJ, 1980.

[30] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363. Assocation for Computing Machinery, August 1986.

[31] P. Hudak and S. Anderson. Pomset interpretations of parallel functional programs. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 274)*, pages 234–256, Berlin, September 1987. Springer-Verlag.

[32] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Conference Record of the 12th ACM Symposium on the Principles of Programming Languages*, pages 300–314. Association for Computing Machinery, January 1985.

[33] P. Hudak and B. Goldberg. Serial combinators: "optimal" grains of parallelism. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 382–399, Berlin, September 1985. Springer-Verlag.

[34] P. Hudak and D. Kranz. A combinator-based compiler for a functional language. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pages 122–132, New York NY, January 1984. Association for Computing Machinery.

[35] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Conference Record of the 13th Annual ACM Symposium on the Principles of Programming Languages*, pages 243–254. Association for Computing Machinery, January 1986.

[36] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Conference Record of the 13th Annual ACM Symposium on the Principles of Programming Languages*, pages 97–109. Association for Computing Machinery, January 1986.

[37] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, October 1980.

[38] R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10. Association for Computing Machinery, August 1982.

[39] R. J. M. Hughes. Backwards analysis of functional programs. Research Report CSC/87/R3, University of Glasgow, March 1987.

[40] R. A. Iannucci. A dataflow/von Neumann hybrid architecture. Technical Report TR-418, Massachusetts Institute of Technology Laboratory of Computer Science, Cambridge MA, May 1988.

[41] T. Johnsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, June 1984. (Proceedings of the SIGPLAN 84 Symposium on Compiler Construction).

[42] T. Johnsson. Lambda lifting. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 201)*, pages 190–203, Berlin, September 1985. Springer-Verlag.

[43] T. Johnsson. Target code generation from G-machine code. In *Graph Reduction (Lecture Notes in Computer Science; 279)*, pages 119–159, Berlin, October 1986. Springer-Verlag.

[44] T. Johnsson. A proposal for a lazy functional language. (Working Paper), January 1988.

[45] M. B. Josephs. Functional programming with side-effects. *Science of Computer Programming*, 7(3):279–296, November 1986.

[46] H. Richards Jr. An overview of the Burroughs *norma*. In *Proceedings of the Workshop on Implementation of Functional Languages*, pages 429–438. University of Goteburg and Chalmers University of Technology, February 1985. (Technical Report PMG-R17).

[47] R. M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In *AFIPS-NCC Proceedings*, pages 613–622, June 1979.

[48] J. W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

[49] J. W. Klop. Term rewriting systems: A tutorial. Rapportnr IR-126, Vrise Universiteit, Amsterdam, 1987.

[50] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986. (Proceedings of the SIGPLAN 86 Symposium on Compiler Construction).

[51] J.-J. Levy. An algebraic interpretation of the λ-β-K-calculus and a labelled λ-calculus. In *λ-Calculus and Computer Science Theory (Lecture Notes in Computer Science; 37)*, Berlin, March 1975. Springer-Verlag.

[52] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming (Lecture Notes in Computer Science; 83)*, pages 269–281, Berlin, April 1980. Springer-Verlag.

[53] R. S. Nikhil, K. Pingali, and Arvind. Id nouveau. Computation Structures Group Memo 265, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, July 1986.

[54] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, Cambridge MA, August 1988.

[55] S. L. Peyton-Jones. The tag is dead—long live the packet. Message on Functional Programming mailing list, October 1987.

[56] S. L. Peyton-Jones, C. Clack, J. Salkild, and M. Hardie. GRIP—a high-performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 274)*, pages 98–112, Berlin, September 1987. Springer-Verlag.

[57] K. Pingali and Arvind. Efficient demand-driven evaluation. Part 1. *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, April 1985.

[58] U. S. Reddy. Functional logic languages part I. In *Graph Reduction (Lecture Notes in Computer Science; 279)*, pages 402–425, Berlin, October 1986. Springer-Verlag.

[59] J. Rees and W. Clinger. Revised^3 report on the algorithmic language scheme. Technical report, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambride MA, 1986.

[60] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *ACM SIGPLAN Notices*, 21(6):17–26, June 1986. (Proceedings of the SIGPLAN 86 Symposium on Compiler Construction).

[61] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 202–211. Association for Computing Machinery, August 1986.

[62] M. Scheevel. NORMA: A graph reduction processor. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 212–219. Association for Computing Machinery, August 1986.

[63] S. K. Skedzielewski and J. R. W. Glauert. IF1: An intermediate form for applicative languages. Reference Manual M-170, Lawrence Livermore National Laboratory, Livermore CA, July 1985.

[64] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in IF1. In *Functional Programming Languages and Computer Architectures (Lecture Notes in Computer Science; 201)*, pages 17–34, Berlin, September 1985. Springer-Verlag.

[65] G. L. Steele. LAMBDA: The ultimate declarative. AI Memo 379, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge MA, November 1976.

[66] G. L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge MA, May 1978.

[67] G. L. Steele and G. J. Sussman. LAMBDA: The ultimate imperative. AI Memo 353, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge MA, March 1976.

[68] J. E. Stoy. *Denotational Semantics.* MIT Press, Cambridge MA, 1977.

[69] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 159–166. Association for Computing Machinery, 1984.

[70] K. R. Traub. An abstract parallel graph reduction machine. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 333–341. IEEE, June 1985.

[71] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report TR-370, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, August 1986.

[72] D. A. Turner. A new implementation techique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[73] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 85-92. Association for Computing Machinery, 1981.

[74] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architectures (Lecture Notes in Computer Science; 201)*, pages 1–16, Berlin, September 1985. Springer-Verlag.

[75] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, December 1974.

[76] P. Wadler. Report on the programming language Haskell. (In Preparation), July 1988.

[77] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science; 274)*, pages 386–407, Berlin, September 1987. Springer-Verlag.

[78] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus.* PhD thesis, Oxford University, 1971.

OFFICIAL DISTRIBUTION LIST

Director — 2 copies
Infcrmation Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Office of Naval Research — 2 copies
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

Director, Code 2627 — 6 copies
Naval Research Laboratory
Washington, DC 20375

Defense Technical Information Center — 12 copies
Cameron Station
Alexandria, VA 22314

National Science Foundation — 2 copies
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

Dr. E.B. Royce, Code 38 — 1 copy
Head, Research Department
Naval Weapons Center
China Lake, CA 93555