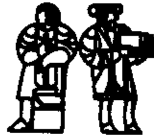
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages

Computation Structures Group Memo 291
October 1988

Kenneth R. Traub

To appear in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, held September 1989 in London, sponsored by IFIP WG 2.8 and ACM SIGPLAN/SIGARCH.

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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Compilation as Partitioning:
A New Approach to Compiling Non-Strict Functional Languages

Kenneth R. Traub*
MIT Laboratory for Computer Science
Cambridge MA 02139

October 1988

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. Consequently, the compiler must break the program into sequential fragments, or *threads*, whose relative ordering is determined at run time. Good compilers employ strictness analysis and other techniques to make threads as large as possible, to minimize run-time overhead. While partitioning a program into sequential threads is a crucial issue, existing compilers treat it as a byproduct of applying some other methodology, such as performing Henderson's force/delay transformation or generating intermediate code for an abstract graph reduction machine (*e.g.*, the G-machine or Tim).

In this paper, we present a view of functional language compilation that takes partitioning a function into sequential threads as the first order of business. The resulting framework cleanly separates issues of partitioning, of thread scheduling, of environments, and of data type representation (including first-class functions). Our method sidesteps both the force/delay transformation and abstract machines, going directly from source code to sequential three-address code. Nevertheless, nearly all of the optimizations proposed for existing approaches are easily expressed. We consider two non-strict evaluation rules: the familiar lazy evaluation, and lenient evaluation as found in the language Id. We also consider both uniprocessor and parallel processor scheduling policies.

1 Introduction

Why is compiling sequential code from functional languages so much harder than from imperative languages? It is not because functional languages have sophisticated constructs like pattern matching and list comprehensions; these are easily treated as syntactic sugar for more primitive constructs. Nor is it because functional languages support full use of higher-order functions; so do imperative languages like Lisp and Scheme, and the required compiler technology is well known [25, 22]. The crucial difference is that given a functional program, it is often impossible to tell at compile time the order in which subexpressions are to be evaluated, whereas in an imperative language the programmer explicitly declares the ordering via the textual ordering of statements. Consequently, the functional compiler must break the program into sequential fragments, or *threads*, whose relative ordering is determined at run time. Good compilers try to make threads as large as possible, to minimize the run time overhead of determining this ordering.[1]

*Funding for the Laboratory for Computer Science 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.

[1] Although small threads may be desirable for *parallel* architectures in order to increase parallelism.

One source of the thread problem is the delaying of computation required by *lazy evaluation.* Consider a function:

```
f x y z = (cons E1 E2);
```

where E_1 and E_2 are large expressions, which may have `x` , `y`, and `z` as free variables. Under lazy evaluation, E_2 is not to be evaluated until a `tl` operation is performed on the returned cons cell; if a `tl` is never performed, E_2 is never evaluated. This implies the code to compute E_2 cannot be a part of the main code for `f`, but instead must be compiled as a separate piece of code (often called a *thunk*), to which control is transfered when the first `tl` operation takes place. Notice that even though E_2 is executed conditionally, this is a fundamentally different problem from compiling a conditional construct: it is simply not possible to use a conditional branch to control the evaluation of E_2, since the decision of whether to evaluate it is taken outside the body of `f`.

Even without lazy evaluation, the thread problem remains. Consider the function `abc`:

```
abc 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};
```

(The `{... in ...}` construct is a mutually recursive equation group, or "letrec" block.) It is easily shown at compile time that all of `abc`'s subexpressions are needed to compute `c`, so laziness (avoiding unnecessary computation) is of no consequence here. Nevertheless, a single sequential thread cannot be produced for `abc`, because for some inputs `a` and `aa` must be computed before `b` and `bb`, while for other inputs the reverse is true. Again, this is not simply a matter of inserting conditional branches; this is perhaps more evident if the bindings for `a` and `b` are replaced by `a = f x bb` and `b = g x aa`, so that the ordering between `aa` and `bb` depends on what happens inside of `f` and `g`. At least two threads are needed for `abc`: for example, one for `b` and `bb`, and one for the remainder.

In this paper, we present a view of functional language compilation that takes partitioning a function into sequential threads as the first order of business. After partitioning, there are three areas of design decisions for the code generator:

- How are threads dynamically scheduled at run time, and how do they communicate?
- How are threads closed over their environments (*i.e.*, how do they gain access to their free variables)?
- How are data types represented, including data structures and first-class functions?

We will concentrate mainly on partitioning and the issues related to thread scheduling. Issues of environment representation are adequately discussed elsewhere [25, 22]. We will present some simple data type representations for purposes of illustration, but none of our discussion will depend on a particular representation. All functional language compilers face these design decisions in one

form or another; the unique aspect of our work is the extent to which they are treated orthogonally. Furthermore, our method sidesteps both the Henderson force/delay transformation [14] and abstract graph reduction machines [19].

We point out that the thread problem does not arise for *strict* functional languages (*e.g.*, the functional subset of Scheme), which require that the arguments of a function call or data constructor be completely evaluated before the function is called or the structure built, and similarly require that all right-hand sides in a block be evaluated independently. Our discussion here is confined to *non-strict* languages, which lack this restriction. Non-strictness is a characteristic of lazy evaluation, and also *lenient* evaluation as found in the language Id [23]. Lenient evaluation differs from lazy evaluation in that the decision of *whether* to evaluate a subexpression is governed only by conditional expressions, not by whether a subexpression is required to produce the answer. Manipulation of infinite objects such as streams is not possible under lenient evaluation without the use of annotations [13], but lenient evaluation may lead to larger threads.

We begin in Section 2 by introducing a minimal kernel functional language called *functional quads*, along with its operational semantics. The semantics defines the behavior that we wish our compiled code to achieve, and also serves to describe precisely lazy and lenient evaluation. In Section 3 we show how to produce object code in which each subexpression is placed in a separate thread, and discuss various scheduling policies. Creating larger threads for lazy and lenient evaluation are the topics of Sections 4 and 5, respectively. Section 6 concludes by comparing our compilation method to existing techniques.

Most of the topics discussed herein are explored in greater depth in [28].

2 Functional Quads

The kernel functional language used in this paper is called *functional quads*, and its syntax is given in Figure 1. A point of notation: the syntactic categories *Identifier* and *StructTag* are partitioned into subsets according to arity; $Identifier^{(2)}$ is the set of identifiers of arity 2, for example. 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. Like lambda calculus, functional quads allows full use of higher-order functions and functions as first-class values (through the *Partial* syntax). But, functional quads also includes primitive arithmetic, data structures, conditionals, and mutually recursive equation groups (via *Block*), reflecting the desire to give special treatment to all of these in compiled code.

To illustrate functional quads, here is a definition for factorial, expressed in Id and in functional quads:

Id	*Functional Quads*

<pre>fact x =
  if x <= 0 then
    1
  else
    x * (fact (x - 1));</pre>
	<pre>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⁾};</pre>

Scalar	::=	*Number* | `true` | `false`	
Struct	::=	$< \mathit{StructTag}^{(n)} \underbrace{, \mathit{Identifier}^{(0)} \ldots , \mathit{Identifier}^{(0)}}_{n} >$	$n \geq 0$
Partial	::=	$(\mathit{Identifier}^{(n)} \underbrace{\mathit{Identifier}^{(0)} \ldots \mathit{Identifier}^{(0)}}_{i})$	$0 \leq i < n$
Value	::=	*Scalar* | *Struct* | *Partial*	
Primary	::=	$\mathit{Identifier}^{(0)}$ | *Value*	
Simple	::=	*Primary* | `const` *Value* | *Primary Op Primary* | `if` *Primary* `then` *Block* `else` *Block* | `sel_`*t*`_`*i* *Primary* | `is_`*t*`?` *Primary* | *Primary* $\mathit{Identifier}^{(0)}$	
Op	::=	`+` | `-` | `*` | `/` | `==` | `<` | ...	
Block	::=	`{` *Binding* ; *Binding* ; ... `in` $\mathit{Identifier}^{(0)}$ `}`	
Binding	::=	$\mathit{Identifier}^{(0)}$ `=` *Simple* | $\mathit{Identifier}^{(n)} \underbrace{\mathit{Identifier}^{(0)} \ldots \mathit{Identifier}^{(0)}}_{n}$ `=` *Block*	$n \geq 1$
State	::=	*Binding* ; *Binding* ; ...	

- 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.
- All identifiers appearing on left hand sides must be pairwise distinct, even across the boundaries of blocks.

Figure 1: Syntax of Functional Quads

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

$$\Diamond^{(0)} \texttt{ = (fact}^{(1)}\texttt{) 5; fact}^{(1)}\texttt{ x}^{(0)}\texttt{ = \{...\};}$$

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 restricted syntax for expressions gives programs an appearance akin to the sequential "three-address," or "quads," notation used to describe sequential object code, where each line describes a single computation, hence the name "functional quads." There is a one-to-one correspondence between "bindings," "subexpressions," and "left-hand-side occurences of identifiers." It should be emphasized, however, that unlike sequential quads the semantics of functional quads is declarative rather than imperative; in particular, there is no significance to the *order* of bindings in functional quads.

2.1 Semantics

We give the semantics of functional quads operationally, as an abstract reduction system [16].[2] If a and b are two states, then $a \vdash b$ (read "a reduces in one step to b") if b is a state obtained by performing one step of evaluation on a. The relation $\vdash$ is defined 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 that 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 $\vdash$ holds for the following pairs of states (among others):

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

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.

$$X = Y;\ Y = V; \implies X = V;\ Y = V; \qquad \text{(R1a)}$$

$$X = Y\ Op\ P;\ Y = V; \implies X = V\ Op\ P;\ Y = V; \qquad \text{(R1b)}$$

$$X = P\ Op\ Y;\ Y = V; \implies X = P\ Op\ V;\ Y = V; \qquad \text{(R1c)}$$

$$\begin{array}{l} X = \texttt{if}\ Y\ \texttt{then \{...\} else \{...\};} \\ Y = V; \end{array} \implies \begin{array}{l} X = \texttt{if}\ V\ \texttt{then \{...\} else \{...\};} \\ Y = V; \end{array} \qquad \text{(R1d)}$$

$$X = \texttt{sel_}t\texttt{_}i\ Y;\ Y = V; \implies X = \texttt{sel_}t\texttt{_}i\ V;\ Y = V; \qquad \text{(R1e)}$$

$$X = \texttt{is_}t\texttt{?}\ \ Y;\ Y = V; \implies X = \texttt{is_}t\texttt{?}\ \ V;\ Y = V; \qquad \text{(R1f)}$$

$$X = Y\ Z;\ Y = V \implies X = V\ Z;\ Y = V \qquad \text{(R1g)}$$

Collectively, these rules allow an identifier to be substituted by the value to which it is bound, for all contexts where a value is needed for some other rule to apply. Because only values are substituted, the computation which reduces an identifier to a value is shared among all references to that identifier.

[2]The reduction system presented here is a slight variation on the system presented in [3]. The mathematical properties are explored in great detail in [28], which includes a confluence proof.

X = const V $\Longrightarrow$ X = V (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 Section 3.1.

X = if true then {$B_{t,1}$;...;$B_{t,n}$ in Y_t} else {...}; $\Longrightarrow$ X = Y_t; $B_{t,1}$;...;$B_{t,n}$; (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 restriction.

X = V_1 + V_2; $\Longrightarrow$ X = V_3; (R4)

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

X = sel_t_i <t,Y_1,...,Y_i,...,Y_n>; $\Longrightarrow$ X = Y_i; (R5)

X = is_t? <t,Y_1,...,Y_n>; $\Longrightarrow$ X = true; (R6a)

X = is_t? V; $\Longrightarrow$ X = false; (R6b)

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

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

where $1 \leq i < n$.

X = ($F^{(n)}$ Z_1 ... Z_{n-1}) Z_n;
$F^{(n)}$ Y_1 ... Y_n = {B_{F1};...;B_{F_m} in Z_F}; $\Longrightarrow$ X = Z'_F;
$F^{(n)}$ Y_1 ... Y_n = {...};
B'_{F1};...B'_{Fm};
Y'_1 = Z_1;...;Y'_n = Z_n; (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, at any level of nesting. (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.)

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. By analogy to term rewriting systems, we call the subexpression that is replaced a *redex*; 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.

Executing a program consists of successive application of rewrite rules to an initial state until the answer, $\Diamond$, becomes a value. Here is an example, in which the selected redex is underlined at each step:

```
    ◇ = 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;
```

Notice that the call to procedure `f` is executed before the argument `a` is reduced to a value, illustrating how non-strictness is achieved through the use of identifiers. Data structures are similarly non-strict because they can be manipulated as values even when they contain identifiers that are not yet reduced to values.

The strategy for choosing which redex to reduce next does not affect the value of the answer obtained from a given initial state, although it may affect whether an answer is obtained at all. As we will see, the strategy also has an effect on the efficiency of compiled sequential code designed to mimic the strategy. We will consider two strategies here.

Under the *lazy strategy*, the next redex to reduce is chosen by the following function:

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

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

$\mathcal{L}$ basically traces its way back from the answer until it finds a redex required to make further progress (a "needed" redex [17]). It can be shown that this strategy always reduces the answer to a value when it is possible to do so [28].

The other strategy we consider is the *lenient strategy*, which simply says that at each step *any* redex in the state may be reduced. This is not quite as anarchic as it sounds, as it does guarantee that nothing in an arm of a conditional is executed until the predicate is known: only after the predicate is known do redexes in the arms of a conditional become part of the state, and therefore subject to reduction. Though we have the freedom to choose any redex at each step, in general it is not possible at compile time to determine in what order the subexpressions of a function will

become redexes. Thus the lenient strategy does not eliminate the difficulty of generating sequential code.

Because the lazy strategy always finds an answer when it is possible to do so, it is strictly more powerful than the lenient strategy. From the programmer's point of view, both strategies have the property that arms of a conditional are not executed until the predicates are known. The programmer can rely on conditionals to terminate recursion. The lazy strategy gives the additional guarantee that no subexpression is executed unless known to contribute to the answer. This allows a recursion in the producer of a data structure to be terminated by the consumers of that data structure; especially, a producer which appears to construct an "infinite" list only performs a finite amount of computation if only a finite prefix of the list is read. This power can be recovered in the lenient strategy to some extent by introducing explicit "delay" annotations [13]. Even without such annotations, however, there appear a large number of applications which require non-strictness but not laziness, and so the lenient strategy will suffice (see, for example, the "circular programs" of Bird [5], and the array programs in [3]). As we will see, the advantage of the lenient strategy is that it leads to larger sequential threads, and therefore less run-time overhead.

We note that if "fair scheduling" is enforced, then the lenient strategy always finds an answer when the lazy strategy does, though possibly with a finite amount of extra work. We will not, however, guarantee fair scheduling in all of the lenient implementations we present.

3 Sequential Quads and Singleton Threads

We now show how to compile sequential code which mimics the behavior of the functional quads reduction system. There will be a one-to-one correspondence between identifiers bound in the state of the reduction system and memory locations in the compiled implementation. The components of *Struct* values, too, will be reflected as memory locations. Now when a function is invoked in functional quads (rule R8), a copy of its body is added to the state with new identifiers created for its local variables. In the compiled implementation, we will have "pure" code for each function which is shared among all its invocations, but each time the function is invoked we allocate new memory locations corresponding to the new identifiers added to the functional quads state. The pure code, operating on a particular set of newly allocated locations, will perform computations corresponding to the reductions performed on the bindings added to the state by rule R8.

Because the order in which reductions are performed cannot, in general, be determined at compile time, the code for a function will actually be a collection of several sequential threads, each mimicking a subset of the reductions performed for a complete invocation. Within each thread the relative order of reductions is fixed, but the total order for all the function's reductions can vary through different interleavings of the function's threads. In particular, different invocations of the same function may have different interleavings. The actual order resulting for a given invocation depends both on the sequential order within each thread and on the scheduling mechanism which interleaves them; thus, the combination of both of these determines which redex selection strategy we mimic.

In the remainder of this section, we consider code in which there is a one-to-one correspondence between bindings (*i.e.*, subexpressions) in a function definition and the threads produced from it. Partitionings which assign more than one binding to a thread are the subject of the next two sections.

3.1 Sequential Code From Bindings

What code do we generate for a binding? Consider a binding of the form X = Y + Z in the state. Three reductions affect this binding. The first two are Rule R1b and Rule R1c, which substitute values for Y and Z, yielding the binding X = V_1 + V_2. Following that, Rule R4 transforms the binding to X = V_3, at which point no further reduction is possible. The life of all bindings follow a similar pattern: some R1 rules may bring values into its right hand side (depending on the type of expression), after which one of the rules R2 through R8 transforms it to the form X = V, with no further reduction possible on that binding.

Given the foregoing, the sequential thread generated for each binding consists of code which mimics any R1 rules needed according to the expression type, followed by code which mimics the R2, R3, ..., or R8 rule, as appropriate. Remember that each time a function is invoked we intend to allocate memory locations for each of the local variables. We can therefore mimic the execution of an R1 rule which substitutes the value of X as fetching from the location corresponding to X. Similarly, a R2 through R8 rule acts by computing some value and storing it in the location corresponding to the left hand side identifier. The situation is slightly more complex because the relative order in which threads execute is not known at compile time: code for an R1 rule must wait until a value has actually been stored in the location before fetching, and code for the other rules must indicate that values have been stored. This is accomplished by adding *presence bits* to memory locations that indicate whether they are "empty" or "full." A location with a presence bit is called a *tagged location.*

The sequential code generated for each type of binding (excluding constructs for higher-order functions) is given in Figure 2; for simplicity, we have assumed that the `const` statement is used whenever a value is to be included in a function definition. The notation for sequential code is the standard "quads" notation [1], to which we have added five primitives for manipulating tagged locations:

'*TaggedLoc* $:=_v$ *Expression*' Computes the value of *Expression*, and stores it in *TaggedLoc* with the presence bit set.

'*force TaggedLoc*' Suspends execution of the current thread until the presence bit of *TaggedLoc* is set.

'*val*(*TaggedLoc*)' Valid only when the presence bit of *TaggedLoc* is set, retrieves the value stored there. Used as an operand, not as a statement.

'$TaggedLoc_1$ $:=_c$ $TaggedLoc_2$' Makes $TaggedLoc_1$ be a "copy" of $TaggedLoc_2$, so that any *force* or *val* operation subsequently performed on $TaggedLoc_1$ behaves as if performed on $TaggedLoc_2$. This is more than simply copying the *contents* of $TaggedLoc_2$ into $TaggedLoc_1$; it implies some sort of indirection is stored in $TaggedLoc_1$.

'*TaggedLoc* $:=_p$ *Closure*' Resets the presence bit of *TaggedLoc*. This construct does not appear in Figure 2 because it is only used in initialization code. The right hand side can be used to indicate what thread will ultimately do a $:=_v$ store into *TaggedLoc*; we discuss this later.

One advantage of this notation is that it abstracts away from the details of how presence bits are implemented, and also from the scheduling policy. The scheduling policy is primarily determined by the implementation of the *force* operator, as that operator interrupts the execution of a thread. Another advantage is that it clearly distinguishes between locations with presence bits (tagged locations) and those without ("untagged" locations, such as `temp` in Figure 2). Any use of tagged

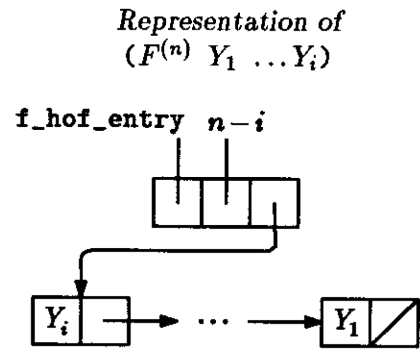
X = Y	*force* Y X $:=_v$ *val*(Y)
X = Y_1 + Y_2	*force* Y_1 *force* Y_2 X $:=_v$ *val*(Y_1) + *val*(Y_2)
X = `const` C	X $:=_v$ C
X = `const` $<t, Y_1, \ldots, Y_n>$	`temp` $:=$ *allocate* $n+1$ `temp[0]` $:=_v$ t `temp[1]` $:=_c$ Y_1 ... `temp[`n`]` $:=_c$ Y_n X $:=_v$ `temp`
X = `sel_`t`_`i Y	*force* 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 $\ldots Y_{n-1}$) Y_n	*begincall* `f` `Arg`$_1$ $:=_c$ Y_1 ... `Arg`$_n$ $:=_c$ Y_n *invoke* `f` X $:=_v$ *val*(`Res`) *endcall* `f`
X = `if` Y `then` {$B_{t,1}$;...;$B_{t,n}$; `in` Y_t} `else` {$B_{e,1}$;...;$B_{e,m}$; `in` Y_e}	*force* Y *if* *val*(Y) *goto* `L1` *force* Y_e X $:=_v$ *val*(Y_e) *goto* `L2` `L1:` *force* Y_t X $:=_v$ *val*(Y_t) `L2:` ... + conditional branches around code for $B_{t,1}, \ldots, B_{t,n}$ and $B_{e,1}, \ldots, B_{e,m}$

Figure 2: Basic Code Generation Schemata

```
X = Y1 Y2              force Y1
                       ap         :=   val(Y1)
                       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  Y2
                       n_chn[1]   :=v  chn
                       n_ap[2]    :=v  n_chn
                       X          :=v  n_ap
                       goto L2
                  L1:  begincall (entry)
                       Arg1       :=v  chn
                       Arg2       :=c  Y2
                       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
$(F^{(n)}\ Y_1\ \ldots Y_i)$



H.O.F. entry code for $F^{(n)}$

```
function f_hof_entry (chn, last)
begincall 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])
...
Arg1       :=c  temp[0]
invoke f
temp       :=   val(Res)
endcall f
◇          :=v  temp
```

Figure 3: Basic Code Generation Schemata for Higher Order Functions

locations will always entail some overhead relative to untagged locations; one of the chief benefits of creating larger threads will be the reduced use of tagged locations.

It is important to understand how non-strictness is achieved in our sequential code. In the functional quads reduction system, non-strictness arises because function application and data structure manipulation can take place even when the argument identifiers or data structure components have not been reduced to values. In the sequential code, this is reflected by the use of the $:=_c$ operator, which effectively "copies" a location even if it does not yet contain a value.

Missing from Figure 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 of partial application values, patterned after [27]. The form of a partial application object is depicted in the upper right corner of Figure 3. Applying it 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 the new argument added to the head of the argument list. 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.

3.2 Threads

The complete code for a function consists of the threads for each binding plus an initialization thread. Here is a small function:

```
f x y = {w = x + 1; z = w / x; a = const <cons,z,y>; in a}
```

And here is the sequential code produced for it, assuming one binding per thread:

```
function f (x, y)
(Allocate w, z, a)
temp1 := (Close thread 2 over x, w, z)
temp2 := (Close thread 3 over x, w)
temp3 := (Close thread 4 over z, y, a)
z     :=p temp1
w     :=p temp2
a     :=p temp3
force a
◇     :=v val(a)
stop
```

```
thread 2 of f
force w
force x
z :=v val(w) / val(x)
stop
```

```
thread 3 of f
force x
w :=v val(x) + 1
stop
```

```
thread 4 of f
temp    := allocate 3
temp[0] :=v cons
temp[1] :=c z
temp[2] :=c y
a       :=v temp
stop
```

(◇ stands for whatever location is used to return results, according to the calling convention chosen for `f`. Similarly, `x` and `y` stand for the locations used to pass arguments.) By convention, the caller always transfers control to thread 1, whose job it is to initialize the other threads and return the result. The first steps of initialization allocate storage for the local variables `w`, `z`, and `a`, and close the other three threads over their free variables. We take no position on how the environments for the threads are represented, whether they are shared, or whether they are allocated in registers, stack, or heap; all of these issues are adequately discussed elsewhere [22, 25]. The $:=_p$ assignments reset the presence bits for `w`, `z`, and `a`. For certain scheduling policies discussed below, they also record pointers to the thread closures, so that they can be invoked on demand.

Not all initialization need be done at the beginning of thread 1. In particular, the initialization of threads and locations contained in an arm of a conditional is best done after the predicate is computed, so that no effort is wasted initializing threads and locations for the branch not taken. This also eliminates the need to surround those threads with conditional branches (see the conditional schema in Figure 2).

3.3 Implementing Tagged Locations on Uniprocessors

An explanation of how to implement tagged locations completes the description of code generation. Many variations are possible, but we will illustrate in detail a scheme appropriate for a conventional von Neumann machine, which can only execute one thread at a time and has no special hardware support for switching among threads. Threads are switched explicitly 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.

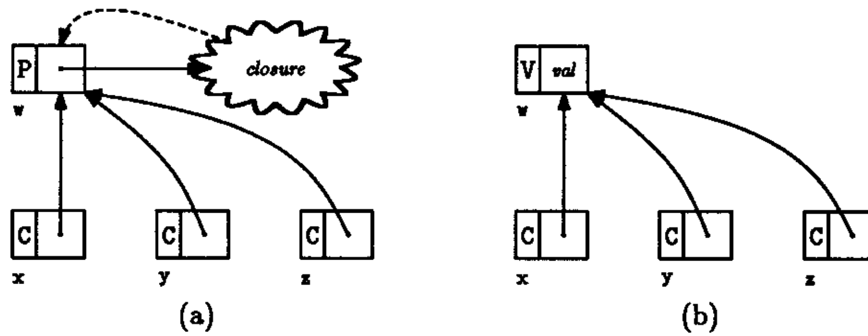


Figure 4: A Uniprocessor Demand-Driven Tagged Location Scheme

The *force* operator is thus a kind of procedure call, where the procedure to be called is indicated by the closure stored in the forced location. 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; lazy evaluation requires both demand-driven scheduling *and* a suitable partitioning.

The simplest scheme uses three tags (*i.e.*, three possible combinations of presence bits per tagged location) corresponding to the three types of assignment operator. To illustrate, Figure 4a shows four tagged 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 given below. To avoid being tied to any particular instruction set, we will use a "pseudo-algol" notation to describe these procedures. 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 that the statement [*loc*] $\leftarrow$ *V.value* stores 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. Of course, the "procedures" below are procedures for illustrative purposes only; we intend that the code for each operator replaces each appearence of that operator in the abstract object code.

```
procedure loc :=_v value
  [loc] ← V.value

procedure loc :=_p closure
  [loc] ← P.closure

procedure loc1 :=_c loc2
  if [loc2] = C.addr then
    [loc1] ← [loc2]
  else
    [loc1] ← C.loc2
```

```
function val(loc)
  a ← followindir(loc)
  if [a] = V.value then
    return value
  else error

procedure force(loc)
  a ← followindir(loc)
  if [a] = P.closure then
    call closure
```

(The error check in *val* is not really necessary, since the code we generate always forces a location before applying *val* to it.) The implementation of $:=_c$ is designed so that an indirection never points to another indirection, and the subroutine *followindir* follows any indirection that might be present (again, *followindir* is a "subroutine" only for illustrative purposes):

```
function followindir(loc)
  if [loc] = C.addr then
    return addr
  else
    return loc
```

There are many variations on this scheme possible which optimize various cases; for example, it is possible to make *val* faster at the expense of $:=_v$ and $:=_c$. It is also possible to devise schemes with only two possible tags, thus requiring one presence bit instead of two. See [28] for details.

With one binding per thread, this scheme implements lazy evaluation, as the reader can verify by comparing the lazy evaluation redex selection rule from Section 2 to the appearences of *force* in Figures 2 and 3.

3.4 Implementing Tagged Locations on Parallel Processors

Without going into too much detail, we note that parallel schemes are easily obtained by redefining the *force* operator to fork off a concurrent task to evaluate the closure, rather than calling the closure as a subroutine. Some sort of locking mechanism is also needed to prevent additional initiations of a thread between the time it is first forced and the time it completes its $:=_v$ stores. Additional parallelism can be obtained by inserting *spark* statements into the threads, where *spark* is a variant of the parallel *force* that initiates the concurrent task but does not wait for the location to contain a value (the term "spark" is due to [11]). If demand-driven behavior is to be preserved, *spark* statements cannot be inserted indiscriminantly; a location should only be sparked if it eventually will be forced. With sparks inserted in this way, parallel scheduling selects the same set of threads for execution as does uniprocessor scheduling, and so it mimics the same redex selection strategy (for a given partitioning).

An alternative way of obtaining parallelism is to initiate the concurrent execution of *every* thread closure as soon as it is created; this is called *eager* scheduling. Since a thread is always running by the time it is forced, the job of *force* is considerably simplified under this policy; in dataflow architectures [24, 2] and in Iannucci's hybrid architecture [18] 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 cannot mimic the lazy strategy no matter what partitioning is chosen. But because conditional branches are included in threads to prevent premature execution of code taken from the arms of conditionals, it does mimic the lenient strategy. We stress this because the term "eager" is sometimes used to refer to a strategy of evaluating both arms of a conditional simultaneously [26], which is not the case here.

4 Partitioning for Lazy Evaluation

We now describe larger threads that still retain the same behavior as one-binding-per-thread under demand-driven evaluation, and therefore still mimic the lazy strategy. Such threads are constructed by starting with one-binding-per-thread code, and applying the following rule to combine threads:

- If, among all a function's threads, there is a single statement of the form *force* `x` for some variable `x`, that statement can be replaced by the body of the thread which computes `x` (has a statement of the form `x` $:=_v$ ...).

Taking the code from Section 3.2 as an example, thread 3 can be merged into thread 2, and thread 4 into thread 1, yielding:

```
function f (x, y)
⟨Allocate z⟩
temp1    :=   ⟨Close thread 2 over x, z⟩
z        :=p  temp1
temp     :=   allocate 3
temp[0]  :=v  cons
temp[1]  :=c  z
temp[2]  :=c  y
a        :=   temp
◇        :=v  a
stop
```

```
thread 2 of f
force x
w :=  val(x) + 1
force x
z :=v w / val(x)
stop
```

Notice that as a result of merging there are tagged locations that only appear in one thread, excluding initialization. These can be converted to less expensive untagged locations; this is reflected in the example by the absence of initialization, *val*, and $:=_v$ for the variables `w` and `a`.

All of the other techniques for creating larger threads are just ways of exposing opportunities to apply the replacement rule above. Flow analysis and code motion can reduce the number of forces of a particular location, by noting that a second *force* of a location has no effect whatever. For example, in thread 2 above the first *force* `x` statement dominates (in the flow-control sense [1]) the second *force* `x`, so the latter may be eliminated. Another example: force statements which appear in both sides of a conditional branch can often be moved to a point before the branch, where they can be combined. Path analysis [6] may allow similar transformations by recognizing domination relationships *between* threads, rather than within them. Eliminating *force* statements may expose opportunities for combining threads, if after eliminating them there is only one force of a given local variable. Of course, eliminating needless *force* statements is beneficial even when it creates no opportunities for combining threads.

Strictness analysis [10] also exposes opportunities for combining threads, not by eliminating *force* statements, but by inserting them. If a function is strict in an argument, then a caller can force the corresponding actual parameter before making the call. For example, here is the code for `a = (f x y z)`, assuming `f` is strict in its second argument:

```
force y
begincall f
Arg1 :=c x
Arg2 :=v y
Arg3 :=c z
invoke f
a    :=v val(res)
```

The *force* y may now be a candidate for replacement. Even if it is not, inserting the *force* may be beneficial. If strict arguments are uniformly forced in all first-order calls to a function, including the first-order call that is part of higher-order entry code (Figure 3), strict arguments may be passed in untagged locations, and all forces of those formals may be eliminated from the code for the function. (Passing strict arguments as values and the entry code method for making it work in the presence of higher-order functions are due to [7].)

The use of strictness analysis is not limited to first-order calls: higher-order strictness analysis [8] can insert *force* statements into code for general applications, and strictness analysis of data structures [30, 12] can insert them into constructor code.

Despite the simplicity of these transformations, they actually cover all of the cases needed to achieve threads as large as those produced by the ALFL and G-machine compilers.

5 Partitioning for Lenient Evaluation

The lenient strategy allows much more freedom in choosing a redex, and correspondingly allows much more freedom in the composition of threads. In lazy evaluation, code for a function must be partitioned so that computations not required to produce the final result are isolated from computations that are, for every possible invocation of the function. In lenient evaluation, we need only guard against placing two subexpressions in a thread in an order which contradicts what is actually required for some invocation. In other words, we need only avoid deadlock, not extra computation.

To illustrate how this leads to larger threads, consider the following fragment:

```
a = ...
c = f a b;
...
d = g a;
```

Assume that f is strict in its first argument; then the thread containing the code for c can have a *force* a statement. Now under lazy evaluation, the code for a cannot be included in the thread for c, as a reference to a appears in the thread for d. (If a and c were combined, and computing d forces a, then c would be evaluated whether needed for the final answer or not.) Under lenient evaluation, however, the code for a and c *can* be merged, since it is acceptable to compute c regardless of whether it is needed (remember that a conditional branch will be included, however, if the binding for c is within the scope of a conditional).

Another example: suppose the binding for d did not exist in the above example, but that f is not strict in its first argument. Even though there is only one reference to a, under lazy evaluation it is still not acceptable to merge the threads for a and c, since a might not be needed to compute the value of c. Under lenient evaluation, however, the threads can be merged, as long as we can determine which should be computed first. If a does not require the value of c, then computing a

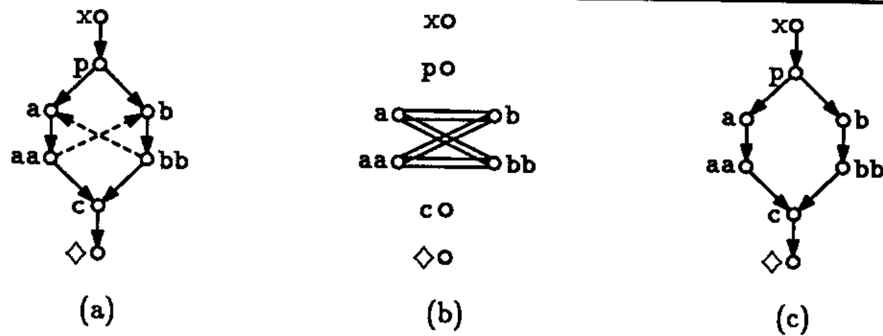


Figure 5: (a) Dependence Graph for abc; (b) Separation Graph; (c) Ordering Constraint

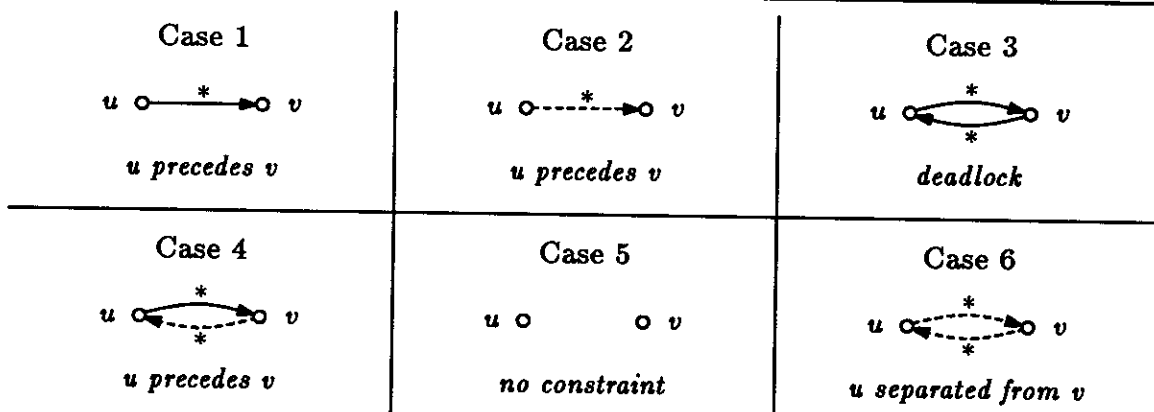


Figure 6: Partitioning Constraints Implied by Dependence Graph Paths

before `c` will always work. On the other hand, `f` could turn out to be the `cons` function, and `a` could be `(tl c) + 5`, in which case `a` would have to appear *after* `c`. In still other situations, both orderings might be possible, and the threads could not be safely merged.

Partitioning for lenient evaluation involves the construction of *dependence graphs* to trace the precedence relationships that hold between subexpressions for various invocations. To illustrate dependence graphs, the dependence graph for Section 1's `abc` function is shown in Figure 5a. Each vertex corresponds to an identifier. A solid edge from a vertex u to another vertex v means that the value of u must be computed before the value of v can, for all invocations of the function. A dashed edge, on the other hand, means u must be computed before v for some, but not necessarily all, invocations. Any analysis technique may be used to derive this dependence information, but it is often useful to phrase the questions in terms of *strictness*: given a binding `a = f x`, if `f` is provably strict in `x` a solid edge may be drawn from `x` to `a`, if `f` provably ignores `x` no edge is needed, and otherwise a dashed edge must be drawn.

The constraints on partitioning are derived by considering *paths* in the dependence graph; specifically, between each pair of vertices, in each direction, there might be a completely solid path, a path that includes at least one dashed edge, or no path at all. This totals six possible configurations (plus three symmetric variations), illustrated in Figure 6. In Cases 1 and 2 we conclude that u must appear before v if assigned to the same thread, since for at least some invocations u must be computed before v. Case 3 implies a deadlock for all inputs, resulting from an erroneous program such as `a = b + 1; b = a + 1`. Case 4 could mean the function deadlocks

in some, but not all, invocations, or perhaps the dashed path resulted from imprecise dependence analysis. In any event, the appropriate action is to treat it like Cases 1 and 2. Case 5 indicates no constraint at all. Case 6 is the most interesting, as it says that for some invocations u must precede v, while for others the reverse is true. To allow for both possibilities, u and v must be assigned to separate threads.

The results of the case analysis are summarized in two other graphs, which have the same vertex set as the dependence graph. The *separation graph* has an undirected edge between pairs of vertices that fall into Case 6. The *ordering constraint* has a directed edge between vertices falling into Cases 1, 2, or 4. These graphs are illustrated for abc in Figures 5b and 5c, respectively (transitive edges have been omitted from the ordering constraint, for clarity). Any k-coloring of the separation graph is therefore a legal assignment of subexpressions to threads. Once assigned to threads, the ordering within each thread is computed by topologically sorting with respect to the ordering constraint.[3] One legal coloring of Figure 5b assigns b and bb to one thread and the remainder to another; because the chromatic number of the graph is two, at least two threads are required.

Dependence graphs and their use in partitioning for lenient evaluation is discussed in much greater detail in [28], but it is worth pointing out one or two other aspects of dependence graphs. Identifiers representing other than scalar data types require more than one vertex in the dependence graph, as a vertex must be included for each independent value. For example, if a variable holds a two-tuple of integers, then three vertices are needed: one for each component, and one for the tuple itself.[4] Now for recursive types this means an infinite set of vertices is required, but there is a theorem which shows that vertices can be coalesced, at the expense of converting some solid edges to dashed. Thus, the vertices for any given type can be coalesced into a finite set—for example, one vertex for the "top level" and one for all subcomponents, sub-subcomponents, *etc.*—with the choice of set determining the precision with which depedence through data structures is to be tracked. In practice, the set used will largely be determined by the dependence analysis technique used (*cf.* Hall's strictness patterns [12]).

The other point worth making is that in general it will be necessary to include dashed edges from the vertex representing the result of a function to the vertices representing the arguments, reflecting the possibility that the function may be called from a context in which the result is fed back to the arguments. This introduces many cycles into the dependence graph, and therefore many separation constraints, implying many small threads. Here, then, is the price of non-strictness: the potential for feedback is precisely why non-strict languages are more expressive than strict languages [5, 3], but it also precisely what makes the order of subexpression evaluation so hard to predict. This suggests that analyzing when feedback actually takes place could have a significant impact on the quality of compiled code.

6 Conclusion

We have presented a framework for compiling non-strict functional languages which views partitioning of a program into threads as the central problem. Once threads are partitioned, separate

[3] A technicality: care must be taken if two vertices in Case 5 are assigned to the same thread, for when they are ordered within that thread an edge is effectively added to the dependence graph. Algorithms which correctly deal with this issue are discussed in [28].

[4] Three vertices are needed regardless of whether the denotational semantics uses a lifted domain for two-tuples. If the domain is not lifted, it only means that a $\perp$ and $\langle\perp,\perp\rangle$ cannot be distinguished *within the language*. Even if this is the case, the *implementation* will be able to distinguish them: one corresponds to an uncomputed location, while the other corresponds to a location containing a two-tuple, each of whose components is uncomputed.

decisions can be taken regarding how those threads are scheduled and communicate with one another, how they are closed over their environments, and how data types are represented. We added tagged locations to the well-known quads notation for sequential code to accomodate the synchronization and inter-thread communication requirements of multi-thread code. The notation concisely exposes all of the issues of partitioning and the optimization of thread overhead while abstracting away from the scheduling mechanism and environment representation. We then considered three scheduling mechanisms—uniprocessor demand-driven, parallel demand-driven, and parallel eager—and showed how they are reflected as implementations of the tagged location primitives. While we began by considering threads no larger than a single subexpression, we showed how larger threads could be constructed for both lazy and lenient evaluation, noting that lazy evaluation also requires a demand-driven scheduling mechanism. The benefits of large threads were reflected in the sequential code by the decreased use of tagged locations. We saw that the more flexible lenient evaluation order allowed larger threads to be created, though at the expense of some expressive power for the programmer.

How does our approach compare to existing compilation techniques? One of the two popular methods is the abstract machine approach, typified by the LML compiler [4]. The idea there is to compile the program into code for an abstract machine, the G-machine [19], and thence into target code [21]. The G-machine performs supercombinator graph reduction [29], with a feature that allows the "short-circuiting" of graph reduction by directly evaluating arithmetic expressions. The equivalent of partitioning comes about when chunks of a function are "wrapped up" in a subroutine so that they can be compiled into this efficient, short-circuited code. Each such chunk therefore yields a sequential thread (a G-code sequence), closed over its environment through the lambda-lifting transformation [20]. The decisions regarding environment and data type representation as well as presence bits and scheduling policy are all tied up in the graph reduction model. Optimizing these aspects therefore requires extensions to the G-machine; thus there have recently proliferated such enhancements as the "spineless" G-machine, the "tagless" G-machine, *etc.* Particularly noteworthy is the "spineless" G-machine [9], which limits updating to locations that are shared ("shared application nodes"); there is a strong correpondence between shared application nodes and our tagged locations. The G-machine has the advantage of dealing with partial applications in a highly optimized way, especially in the case where an unshared partial application is returned from a procedure where it is immediately applied to more arguments by the caller.

The other popular method is represented by the ALFL compiler [15], in which a highly optimized version of the Henderson force/delay transformation [14] is used to convert a lazy program into a form acceptable to the Orbit compiler for Scheme [22]. This method is very much in the sprit of Section 4, as each "delay" expression is nothing more or less than a sequential thread, and equivalent techniques are used to produce them. Like our method, partitioning is isolated from the decisions relating to scheduling, environments, and data types; specifically, these decisions are left up to the Orbit compiler. Tagged location code generated by the ALFL compiler effectively has the property that local variables always contain promises or values ($:=_p$ and $:=_v$ assignments), while arguments and structure locations always contain copies ($:=_c$ assignments), so that only one presence bit is needed (see [28]). Interestingly, the storing of a value in a tagged location is the responsibility of the *forcing* thread, rather than the forced thread. While this rules out a thread computing more than one tagged location as is allowed in our scheme, it permits the forcer to omit the storing altogether when it can be shown that no subsequent forces will be done [7].

The LML and ALFL compilers both implement lazy evaluation. Iannucci gives a partitioning algorithm for lenient evaluation [18], which is a greatly simplified version of the methods outlined in Section 5. Data is not yet available on how the performance of our method (in terms of thread size) compares to Iannucci's, or even to lazy partitioning algorithms.

The compilation-as-partitioning view seems to lead to a much cleaner of separation of issues in compiling non-strict functional languages, whether for lazy or lenient evaluation, for sequential or parallel execution. More work is needed to put these ideas into practice.

Acknowledgements

The author would like to thank Arvind and Jonathan Young for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1986.

[2] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[3] 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.

[4] 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).

[5] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[6] 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.

[7] A. Bloss, P. Hudak, and J. Young. Optimizing thunks. (Draft), 1988.

[8] 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.

[9] G. L. Burn, S. L. Peyton-Jones, and J. D. Robson. The spineless g-machine. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 244–258. Association for Computing Machinery, July 1988.

[10] 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.

[11] 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.

[12] 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.

[13] S. K. Heller. *Efficient Lazy Structures in a Dataflow Machine.* PhD thesis, Massachusetts Institute of Technology, Cambridge MA, December 1988. (Expected).

[14] P. Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall, Englewood Cliffs NJ, 1980.

[15] 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.

[16] 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.

[17] G. Huet and J-J. Levy. Call by need computations in non-ambiguous linear term rewriting systems. Research Report 359, IRIA Laboria, Le Chesnay, France, August 1979.

[18] R. A. Iannucci. A dataflow/von Neumann hybrid architecture. Technical Report TR-418, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, May 1988.

[19] 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).

[20] 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.

[21] 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.

[22] 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).

[23] R. S. Nikhil. Id Nouveau reference manual part II: Operational semantics. Computation structures group memo, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, April 1987.

[24] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Massachusetts Institute of Technology, Cambridge MA, August 1988.

[25] G. L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge MA, May 1978.

[26] S. Tighe, K. Zink, and R. Brice. A flexible architectural study methodology. In *Graph Reduction (Lecture Notes in Computer Science; 279),* pages 297–311, Berlin, October 1986. Springer-Verlag.

[27] 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.

[28] K. R. Traub. Sequential implementation of lenient programming languages. Technical Report TR-417, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, September 1988.

[29] D. A. Turner. A new implementation techique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[30] 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.