Lazy Evaluation and the Logic Variable

Keshav Pingali*

87-877
November 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This author is supported by an IBM Faculty Development Award and by NSF grant CCR-8702668. This paper will be published in the proceedings of the Institute on Declarative Programming held at the University of Texas, Austin, August 24th-29th, 1987.

Lazy Evaluation and the Logic Variable

Keshav K. Pingali

Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

October 31, 1987

Abstract

Functional languages can be enriched with logic variables to provide new computational features such as incremental construction of data structures. In this paper, we present a novel application for logic variables that highlights their importance: we argue that they are essential for efficient implementations of pure functional languages. This point is made by demonstrating that logic variables are required for explicating the process of demand propagation in lazy evaluation of functional programs. There are two applications of this result. For dataflow researchers, it offers a simple and efficient implementation of laziness on dataflow machines. For researchers investigating lazy graph reduction, it suggests new strictness analysis algorithms in which logic variables play an important role.

*The author is supported by an IBM Faculty Development Award and by NSF grant CCR-8702668. This paper will be published in the proceedings of the Institute on Declarative Programming held at the University of Texas, Austin, August 24th-29th, 1987.

1 Introduction

In a functional language, an identifier obtains its value as the result of evaluation of a single applicative expression. In contrast, a *logic variable* in logic programming languages obtains its value incrementally by the intersection of successively applied constraints. The incorporation of logic variables into an otherwise functional language provides the programmer with a powerful tool for writing elegant and efficient programs for problems such as the construction of large arrays in scientific programming [5], owner-coupled sets in database programming [16] etc. which are difficult to write in a purely functional language.

In this paper, we argue that logic variables play a key role in compile-time analysis and implementation of modern functional languages; *i.e.*, functional languages that support *non-strict* data constructors. Informally, a constructor is said to be non-strict if it can produce some output even if one or more of its inputs are undefined. The classic example of such a constructor is the non-strict *cons* introduced by Friedman and Wise [8]. If *cons* is non-strict, the expression *car(cons(1, e))*, where *e* is some non-terminating computation, returns *1*; in contrast, this expression would be undefined in a language like LISP in which *cons* is strict. In conjunction with higher-order functions, non-strict data constructors can be used to write modular and elegant programs[22] and they are an important feature of most modern functional languages. Unfortunately, non-strict constructors are considerably more difficult to implement than their strict counterparts. Using non-strict constructors, a programmer can define infinite data objects such as infinite lists and trees (*e.g.*, in the Miranda program $x = 1{:}x$, where $:$ is the non-strict, infix cons operator, x is an infinite list of 1's). A naive implementation would attempt to construct the entire list, resulting in dire consequences! A more successful strategy is to use lazy evaluation. Although x is an infinite list, it may be the case that only a few of its elements are required to produce the output of the program. A lazy evaluator computes only those elements, not the entire list. This is achieved by suspending computations until it is established that they are needed to produce the output of the program. The process of determining which computations are required to produce the output is called *demand propagation*. During the course of program execution, the result of a suspended computation may be demanded more than once. To avoid repeated evaluation, most lazy interpreters use the principle of *graph reduction* - the first time a suspended computation is demanded, it is evaluated and its code is overwritten by the result. No computation is done on subsequent demands.

Considerable experience has been gained over the past few years in implementing sequential and parallel lazy evaluators[23,18,15,14,12,9]. This experience has revealed two problems. First, there is a definite space and time penalty associated with suspending computations. When a computation is suspended, its environment must be saved and this takes up storage. Moreover, every access to the result of a suspended computation involves a check to ensure that the computation has been performed and the result is available.

A second problem is that lazy evaluation of a program usually does not exhibit much parallelism [6]. One way to ameliorate these problems is to avoid delaying computations whenever it is safe to do so: instead of being completely lazy, the interpreter performs some computations eagerly and some lazily. To identify those computations that need not be delayed, the compiler performs *strictness analysis* to locate computations that are definitely required to produce the output of the program. At run-time, the interpreter evaluates these computations eagerly. Unfortunately, the complexity of algorithms that have been proposed for strictness analysis is exponential in the size of the program [6,10].

In this paper, we present a different approach to the efficient implementation of lazy evaluation. Rather than start with a lazy interpreter and attempt to compile in eagerness by doing strictness analysis, we start with an eager (data-driven) interpreter and compile in laziness. To do so, we formalize the notions of demand and demand propagation, and introduce code for demand propagation explicitly into programs rather than burying it in the interpreter. Interestingly enough, there are two very different ways of introducing demand propagation code. The first method is 'functional' in that sense that all the operators required for demand propagation are pure functions from inputs to outputs. This method is described in Section 3 of this paper and is similar to the algorithm described in [19] where it was presented as a program transformation. A surprising aspect of this transformation is the need for a least upper bound operator that behaves very much like unification (which is the least upper bound in the domain of first-order terms under the instantiation ordering). The need for such an operator can be understood by analogy with Milner's algorithm for compile-time type-checking. To deduce the type for a formal parameter of a function, Milner's algorithm examines the body of the function to see how the parameter is used; each use (occurrence) of the parameter contributes some type information, and a type is inferred for the formal parameter by 'merging' all this information using unification. The algorithm for introducing demand propagation code is similar: each use of a formal parameter contributes some demand information, and the overall demand for the parameter is obtained by merging all this information using an operation similar to unification. What is surprising about this is that conventional lazy evaluators based on graph reduction do not need unification to perform demand propagation at run-time. Is the need for such an operator an unavoidable result of making demand propagation code explicit (rather than burying it in the interpreter) or is it a peculiarity of the method we chose for demand propagation? In this paper, we provide a somewhat paradoxical answer to this puzzle. In Section 4, we present a new algorithm for introducing demand propagation code; this algorithm can also be thought of as a program transformation, but the target language is a 'hybrid' language in which logic variables are integrated into a functional language. Surprisingly enough, transformed programs are a subset of programs in this language that do not need the full power of unification! Moreover, there is a close connection between this transformation and demand propagation in conventional parallel graph reduction interpreters such as the one of Keller et al [15].

One advantage of formalizing demand propagation and introducing demand propagation code explicitly into programs is that it becomes subject to standard program optimizations. This has been studied by Amamiya *et al* [17] who use conventional global optimization techniques such as constant propagation and value subsumption to eliminate demand propagation code whereever possible. There appears to be a close connection between this approach and a recent algorithm for strictness analysis called 'backward analysis'[11]. We believe that this algorithm can be improved and simplified by viewing strictness analysis as an optimization of demand propagation code. There are also reasons to believe that the complexity of such an algorithm would be far better than that of conventional algorithms for strictness analysis [11]. Moreover, the need for logic variables in explicating demand propagation suggests new 'backward analysis' algorithms in which logic variables play an important role.

The rest of the paper is organized as follows. In Section 2, we introduce a simple, first-order functional language F and show how F programs can be executed on an abstract data-driven interpreter. In Section 3, we show how demand propagation code can be introduced into F programs. This is presented as a transformation on F programs which is similar to that described in [19]. In Section 4, we describe the new demand propagation algorithm which is presented as a transformation of F programs to programs in a hybrid language with logic variables. Although the target language has logic variables, transformed programs are a special sub-set which do not need the full power of unification. We extend the technique to handle higher-order functions. In Section 5, we discuss two applications of this transformation. First, we show how it can be used to implement laziness on a concrete dataflow architecture, and contrast the resulting system with the graph reduction approach. Second, we discuss some implications for strictness analysis.

2 A Simple First-order Functional Language

The techniques described in this paper are very general and apply to any functional language such as Miranda [21] or LML [12]. However, to focus attention on the main issues, we will consider a simple, 'desugared' functional language called language F. The main aspect of the desugaring is that every expression in an F program has a name. As we will see, this makes the discussion of demand propagation easier to describe and comprehend.

2.1 Syntax of F

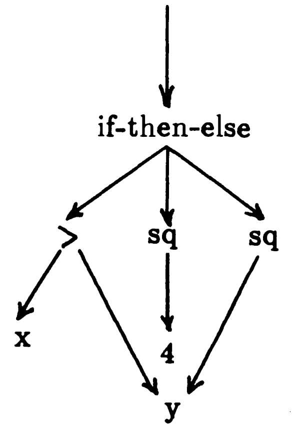
The syntax of F programs is shown in Figure 1. An F program consists of a set of function definitions (f1,...,fm) and a function invocation (of some function fk) whose result is the output of the program. The body of a function is a single *letrec* in which there is a definition

```
program ::=
            f1(id,..,id) = letrec def...def in id,...,id
            ....
            fm(id,...,id) = letrec def...def in id,...,id

            =>fk(constant, constant,...constant)

def ::= id,...,id = exp

exp ::= constant| id| id op id| if id then id else id|
        cons(id,id)| car(id)| cdr(id)|
        fj(id,..,id)|
        fork-i(id)
```

Figure 1: Syntax of Language F

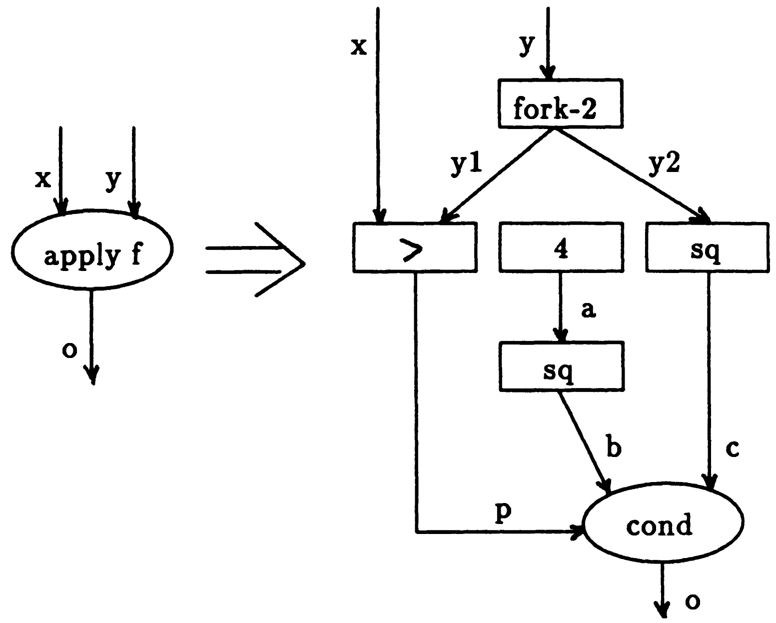
for each identifier introduced in its scope and a list of identifiers whose values are to be returned as the outputs of the function. Expressions are built from constants and identifiers by the application of pre-defined and user-defined functions. Pre-defined functions include the usual operators like +, * etc.; in addition, we provide a data constructor *cons* which is non-strict in both of its arguments. *Car* and *cdr* are the destructuring operators. In F programs, a unique name must be given to every *use* of an identifier. This is accomplished by the introduction of *fork* operators - for each integer i greater than 1, the operator *fork-i* is the identity operator with i outputs. To clarify the point of this desugared syntax, it is useful to consider how a program in a language like LML can be translated into an F program. Ignoring higher-order functions for the moment, the translation is obtained quite simply by drawing the abstract syntax graph for the expression, giving a name to each arc in the graph (*i.e.*, sub-expression in the original program) and creating a definition for each name. An example of such a translation is shown in Figure 2. Nodes that have more than one arc pointing to them (such as the node labeled y) correspond to names in the original program whose value is used in more than one place : to give each use a different name, we introduce *fork* operators. The resulting F program is shown in Figure 2(b). We will not describe this translation formally since we will work only with F programs.

f(x,y) = if x > y then sq(4) else sq(y)



(a)LML Program and Parse Tree

```
f(x,y) = letrec
           y1,y2 = fork-2(y)
            p = x>y1
            a = 4
            b = sq(a)
            c = sq(y2)
            o = if p then b else c
         in o
```



(b) Corresponding F Program and Dataflow Graph

Figure 2: Translating a Conventional Functional Program into an F Program

2.2 Operational Semantics of F Programs

F programs can be given an operational semantics using dataflow. Although there are a wide variety of dataflow models [3,7], the differences between them, for the most part, are not relevant to our discussion. Therefore, we will carry out our development without reference to any particular model, and in the final section, discuss an implementation on the tagged-token dataflow interpreter[3]. The dataflow graph for a function in an F program is obtained by drawing a node for each definition in the body of the function, labeling the node with the function on the righthand side of the definition, labeling its outputs with the identifiers on the lefthand side on the definition and connecting these outputs to nodes which use them. For definitions of the form $x = y$, the node is labeled with the *identity* operator. For expository purposes, we will assume that nodes that represent application of a user-defined function, such as *fk*, are labeled *apply fk*. Figure 2(b) shows the dataflow graph for the F program considered earlier.

The operational behaviour of dataflow graphs is specified by means of 'firing rules' which describe the actions of nodes when they receive tokens on one or more of their inputs. Rather than describe the firing rules formally, we will use the graph of Figure 2(b) to explain the operational behaviour of dataflow graphs. Suppose that function f is called with two arguments *e1* and *e2*. These arguments are evaluated in parallel until one of them (say *e2*) produces a value - *i.e.*, generates a token. When the *apply f* operator receives this token, it creates a copy of the dataflow graph for f and assumes responsibility for the forwarding of arguments and the returning of results - in effect, the dataflow graph for f gets spliced into the original graph in place of the *apply f* operator. Once this is done, operators in the body of f can begin to execute. The token carrying the value of y is duplicated by the *fork-2* operator and sent to the $>$ operator and the *sq* operator. The *sq* operator can execute, producing a token carrying the value of c. When the constant operator labeled *4* executes, it produces a token carrying the value *4* as output[1] - this value is squared to produce a token carrying the value of b. When the $>$ operator receives the values of x and $y1$, it fires, computing the value of the predicate p. Depending on the value of the predicate, the *cond* operator produces a token carrying either the value of b or the value of c. This value is returned to the caller as the result of the function application. It is important to realize that *cond* is non-strict - for example, if p is *true*, and the value of b has been produced, the *cond* operator can produce the output without waiting for the value of c. If and when the value of c is computed, the *cond* operator executes again and simply absorbs the token at its input without producing any output.

We now discuss the implementation of the data structure operations. The *cons* operator requires careful treatment in dataflow to ensure that non-strict semantics are respected.

---

[1] The implementation sends a dummy token to each constant operator in a function graph when the function is invoked, which causes it to fire and produce the appropriate constant.

Consider a definition x = *cons(a,b)* where b is the output of some non-terminating computation. To be faithful to non-strict semantics, the value of a must be available to the consumers of x even though the value of b is never produced. Thus, the *cons* operator cannot wait for both of its input before firing. This difficulty is circumvented by allocating storage for data structures in a special structure storage called *I-structure storage*[2]. The first time a *cons* operator receives an input, it allocates a cons cell in I-structure storage, stores the value it received in the appropriate field of this cell and outputs a token carrying the address of the cell. Subsequently, if a token is received on the other input, its value is stored in the other field of this cell. A *car* or a *cdr* operation executes when it receives a token carrying the address of some cell in structure memory - the appropriate field of this cell is read and the value is output. There is potential for a read-before-write hazard here since a *car* or a *cdr* operation may be executed on a cons cell before a value has been stored in that field. This hazard is avoided with the help of a smart structure controller: if an attempt is made to read an empty field, the controller defers the access until the corresponding write occurs.

Notice that at most one token flows down an arc in the dataflow graph. We will exploit this fact when we discuss a concrete implementation on the tagged-token dataflow machine.

2.3 Denotational Semantics

A denotational semantics for F programs is straight-forward. Let $\mathcal{A}$ be the domain of atoms and let V be the domain that satisfies the equation $V = \mathcal{A} \oplus V \times V$. The coalesced sum is required because there is no difference operationally between $\perp$ and *cons*($\perp$, $\perp$) (storage is allocated for a cons cell only if at least one of its inputs has been produced). Let the operators *+, *, ..cons, car, cdr,* etc. have their usual interpretation in this domain. ***fork-n*** is interpreted as a function from V to V^n which acts as the identity function on each component of its range. A function *fk* with i inputs and j outputs can be interpreted as a monotonic and continuous function from V^i to V^j. The program is interpreted as a set of equations to be solved for the denotations of *f1, f2..fm.* It is a standard result that such a set of equations has a least solution, which gives the meaning of the program.

Using a result of Kahn [13], it can be shown that the results produced by a dataflow interpretation of an F program are consistent with the denotational semantics of the program. This is subject to two caveats. The first caveat is that there must be some notion of fair scheduling: the execution of a node that is ready to fire cannot be postponed indefinitely. The second caveat has to do with constant functions - *i.e.*, user-defined functions that produce some constant value as output regardless of their inputs. The firing rule that we have described for function application invokes the function when at least one of its actual parameters has been computed. This means that a function application of the form *f(a)*, where a is the output of some non-terminating computation, would never be performed; if f is a constant function, this would not be in consonance with the denotational semantics of f. This is not a serious problem since such functions can always be compiled to take an additional parameter which is a constant supplied by the implementation ('trigger'). Thus, the dataflow interpreter is faithful to non-strict semantics and will produce correct results. However, this is cold comfort - if the programmer uses the non-strict *cons* to define infinite data objects, the dataflow interpreter may eventually produce the results of the program but it would perform an unbounded amount of computation over what would have been performed by a lazy interpreter! We will fix this problem over the next few sections.

3 Functional Demand Propagation

To give the reader an intuitive feel for our techniques, we first show how unnecessary computation can be avoided in the program of Figure 2. We then discuss our transformation which is a systematic way of achieving this effect for any F program. This transformation is similar to the one discussed in [19].

3.1 Introducing Demand Propagation Code

To prevent the dataflow interpreter from performing unnecessary computation, it is necessary to 'control' the flow of data in such a way that an operator receives data on its inputs only if it has been established that its output is required to produce the result of the program. Let us see how this can be achieved for the program in Figure 2. To avoid unnecessary computation, the *sq* operator whose output is b must receive a token carrying the value *4* only if the predicate p is *true*. The value of y is needed to compute p, but a token carrying this value should sent to the *sq* operator that produces the value of c only if p is *false*. This is achieved by the transformed program shown in Figure 3 which mimics the way a lazy interpreter propagates demands. The 'backward' graph, called the demand graph, is responsible for demand propagation. For every line x in the original graph, there is a demand line labeled dx (*de*mand line for x). Demand lines will be shown as thick lines to distinguish them from data lines. Tokens carrying a distinguished value d (for demand) flow on lines in the demand graph. The transformation ensures that a data token flows on line x only if a demand token has been produced on line dx. To reduce the complexity of the figures, *fork* and *identity* operators in the demand graph are not shown explicitly - an implicit *fork* operator is present wherever there is fan-out in the demand graph.

Let us work through the execution of the transformed program. The original function f had two inputs and one output; the transformed function Tf has one new input (the demand line for the output of the function) and two new outputs (demand lines for the

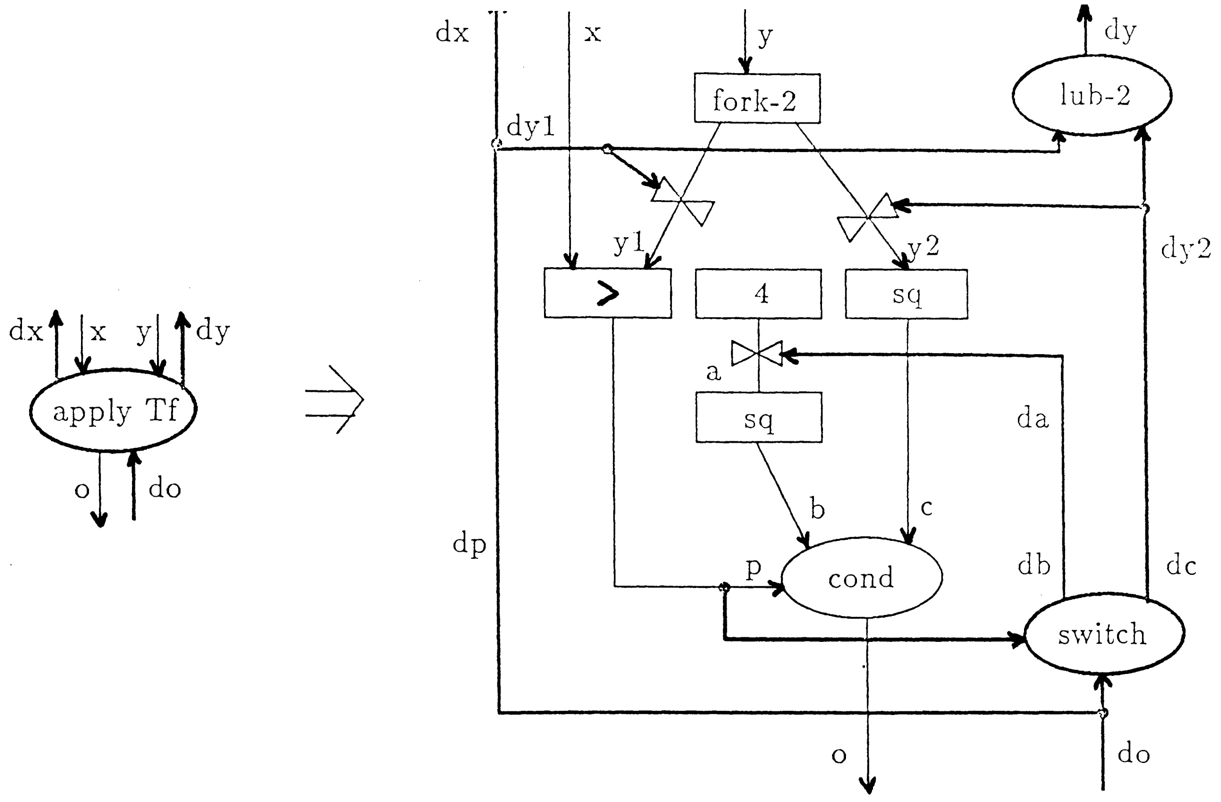


Figure 3: Introducing demand propagation code into the program of Figure 2(b)

inputs of f. When the *apply Tf* operator receives a demand token on line *do*, it creates a copy of the graph of the transformed function *Tf* and forwards the demand token to the body of the function. This causes a demand token to be generated on line *dp* which generates demand tokens on lines *dx* and *dy1*. The token on line *dx* is returned to the caller where it causes the value of x to be computed. The token on line *dy1* causes the *lub-2* operator to fire. The *lub-2* operator combines demands for the outputs of the *fork* into demand for the input of the *fork*. It produces a demand token at its output the first time it receives a demand token on one of its inputs. Subsequently, tokens received on other inputs are consumed without any output (as we will see, this operator can be described denotationally by the least upper bound function). Thus, a demand token is produced on line *dy* which causes the value of y to be computed. The token carrying the value of y must be sent to the $>$ operator but not to the operator labeled *sq* unless the predicate p turns out to be *false*. This is accomplished by the *gate* operators shown as $\bowtie$ in the graph. The *gate* operator behaves like a K combinator that is strict in both of its arguments: it has two inputs and it outputs the token it receives it on its first input when it receives a token on its second input. Thus, a token carrying the value of y is output on line *y1* but not on line *y2*. Once the predicate is computed, a demand token must be sent on either line *db* or on line *dc*. This is accomplished by the *switch* operator which uses the predicate p to output the demand token at its input either on line *db* or on line *dc*. This results in the computation of either b or c and that value is output by the *cond* as the result of the function application.

The effect of lazy evaluation has been obtained by program transformation: data-driven evaluation of the transformed program performs only those computations that would have been done by a lazy evaluation of the original program. Suspension of computations is achieved by controlling the flow of data, and no data value is computed more than once. How was the transformed program obtained in the first place? A systematic method for generating it is shown in Figure 4. Each definition in the source program is replaced with new definition(s) as shown in this figure. [2] We leave it to the reader to verify that the dataflow graph shown in Figure 3 is obtained by applying this transformation to the program in Figure 2. It is obvious from Figure 3 that there is considerable scope for optimization in the demand graph. These optimizations and their relation to strictness analysis will be discussed in Section 4.

The approach introduced above can be extended easily to data structures built from the non-strict *cons*. Demand for an atomic value (such as an integer or boolean) was represented by a token carrying the atomic value *d*. Similarly, a structure consisting of *d* values can be used to represent demand for the elements of a data structure. For example, if *x* is the data structure *cons(1,cons(2,cons(3,...)))* (*i.e.*, the infinite list of integers), the demand structure *cons(d,cons(⊥,cons(d,⊥)* represents demand for the first and third elements of *x*.[3] Given this intuition, the demand propagation code shown for data structure operations in Figure 4 should be easy to comprehend. If *dx* is the demand for the output of the *cons*, *car(dx)* gives the demand for the left input of the *cons* and *cdr(dx)* gives the demand for the right input of the *cons*. The *car* operator never makes a demand for the *cdr* of its input; therefore, if *dx* is the demand for its output, the demand for its input is simply *cons(dx,⊥)*. Similar considerations apply to the *cdr* operator. *Lub* operators, that were introduced to combine demands at *forks*, must be extended to work on demand structures: for example, if *cons(d,⊥)* and *cons(⊥, cons(⊥, cons(d,⊥)* are the demands for the outputs of a *fork*, the *lub* operator must combine them and produce the demand structure *cons(d , cons(⊥, cons(d,⊥)* as the demand for the input of the *fork*.

To give an abstract semantics for transformed programs, let $\mathcal{DA}$ be the two element domain given by the set $\{\perp, \mathrm{d}\}$ where $\perp \sqsubseteq \mathrm{d}$ and let $\mathcal{D}$ be the domain that satisfies the equation $\mathcal{D} = \mathcal{DA} \oplus \mathcal{D} \times \mathcal{D}$. It is natural to interpret the *lub* operator as the least upper bound function in this domain. Unfortunately, two elements in this domain need not have a least upper bound (consider the elements *d* and *cons(d,d)*). This problem arises operationally only in programs which are not type correct - if one output of a *fork* receives a demand for a scalar while another receives a demand for some elements of a data structure,

[2] We have permitted ourselves a little flexibility in the syntax of transformed programs - we do not introduce *fork* operators explicitly on demand lines. In the actual implementation, *constant* operators are handled a little differently - in Figure 3, the line *da* would actually be input to the operator labeled *4* and this operator would produce a token carrying the value *4* when it received a '*d*' token on its input. To avoid having to introduce yet another operator in transformed programs, we have chosen the current presentation.

[3] There is no notion of issuing a single atomic demand for all the elements of a data structure.

Definition in Source	Definition in Transformed Program
x = constant	x = gate(constant,dx)
x = y	x = y dy = dx
x = y op z	x = y op z dy = dx dz = dx
x = if a then b else c	x = if a then b else c da = dx db,dc = switch(a,dx)
x = cons(y,z)	x = cons(y,z) dy = car(dx) dz = cdr(dx)
x = car(y)	x = car(y) dy = cons(dx,⊥)
x = cdr(y)	x = cdr(y) dy = cons(⊥, dx)
x1,..xn = fork-*n*(y)	t1,...,tn = fork-*n*(y) x1 = gate(t1,dx1) ... xn = gate(tn,dxn) dy = lub-*n*(dx1,...,dxn)
x1,...,xn = fi(y1,...,ym)	x1,...,xn,dy1,...,dym = Tfi(y1,...,ym,dx1,...,dxn)

Figure 4: Functional Transformation for Introducing Demand Propagation Code

the demands cannot be 'merged' together in any sensible way. To handle this technically, we add an over-defined element $\top$ to $\mathcal{D}$ which is above all elements of $\mathcal{D}$. If we let D be this new domain, the operators *gate, switch* and *lub-n* can be interpreted as follows:

gate: $\mathrm{V} \times \mathrm{D} \rightarrow \mathrm{V}$
gate(x,y) $=$ if y $= \perp$ then $\perp$ else x

switch: $\mathrm{V} \times \mathrm{D} \rightarrow \mathrm{D} \times \mathrm{D}$
switch(b,x) $=$ if b then x, $\perp$ else $\perp$, x

lub-n: $\mathrm{D}^n \rightarrow \mathrm{D}$
lub-n(x1,...,xn) $=$ x1 $\cup$ .... $\cup$ xn

Cons, car, cdr and *fork-n* are treated as overloaded symbols that are interpreted in D as well as in V. Each operator in the transformed program is a monotonic and continuous function; therefore, transformed programs can be assigned meanings using the standard techniques discussed in Section 2.

The *lub* operator is similar to the unification operator on first-order terms in logic programming languages. Consider the set of first-order terms built out of a two-place constructor *cons* and an atom *d*. A demand structure such as *cons(d, cons(*$\perp$*, cons(d,* $\perp$*)))* can be translated to a first-order term by replacing each occurrence of $\perp$ with a unique variable (say *cons(d, cons(x, cons(d,y)))*. It is easy to verify that the *lub* operator on demand structures corresponds to unification on such first-order terms. Strictly speaking, this operation is a special case of unification (like pattern matching) since *co-referencing*, the occurrence of a variable more than once in a term, cannot happen.

To demonstrate the correctness of the transformation, we would have to show that a dataflow interpretation of the transformed program performs the same computations as a lazy interpreter does on the original program. This measure excludes work done in demand propagation. We refer the reader to [19] where a similar result is shown for a stream language.

3.2 Discussion

The transformation described in this section has demonstrated that the various aspects of lazy evaluation, such as suspension of computations, demand propagation and ensuring that suspended computations are done at most once, can be achieved in the dataflow model by introducing demand propagation code explicitly into programs. On the other hand, the need for demand structures and for a *lub* operator that, in effect, performs unification of demand structures is puzzling because these notions are absent in reduction based implementations of lazy evaluation. A plausible conclusion is that lazy evaluation is tied up intimately with the notion of reduction, so any attempt to model it in a system that does not permit over-writing of code with data must introduce some idiosyncrasies which are a reflection of the system rather than the phenomenon being modeled. However, this 'fatalistic' conclusion does not provide any insight and in fact, is overly pessimistic. In the next section, we show that the need for demand structures arises from the 'functional' nature of the transformation of Figure 4; a new transformation for which the target language is a functional language enriched with logic variables eliminates the need for demand structures and their unification.

4 Using Logic Variables for Demand Propagation

Logic variables can be incorporated into functional languages in a variety of styles. In this section, we adopt the approach taken in our earlier work on *Id Nouveau*[5]. We then show how these variables can be used profitably for demand propagation.

4.1 Logic Variables and Functional Languages

In a functional language, data structures are values which are defined as the result of evaluating a single applicative expression. For example, the evaluation of the ML expression *val x = {name = 'Miranda', age = 2, used = true}* allocates storage for a record with three fields *name, age* and *used*, stores the specified values into these fields and binds the identifier *x* to this record. Once the record has been created, it cannot be mutated. In imperative languages, on the other hand, the usual programming style is to use a command to allocate storage for a data structure and then use other commands to write into this storage. Moreover, the freedom to 'side-effect' a data structure is absolute and an element of a data structure may hold many different values over the lifetime of the data structure. Logic variables are a compromise between these two extremes; intuitively, they provide the programmer with 'write-once' storage. As in imperative languages, storage for a data structure may be allocated without specifying the values of its elements. However, once an element of a data structure has acquired a value, it cannot be reassigned to another value later in the program. For example, in a language like Id Nouveau, the programmer can write

....

```
x = {name = ?, age = ?, used = ?};
....
x.name = 'Miranda';
....
x.age = 2;
....
x.used = true;
....
```

Storage for the record is allocated by the first statement, while the values of its fields are defined by the other statements. In logic programming parlance, this is explained using 'logic variables': when the record is allocated, the fields are initialized to 'logic variables' (say *L1, L2* and *L3*) which are initially unbound. Logic variables are bound to values as the result of execution of other statements in the program. The scope of these bindings is global - for example, if a record is passed as a parameter to a function which defines some of its fields, the bindings stay in effect even after the execution of the function has terminated. This has the flavour of 'call-by-reference' in imperative language. Unlike in imperative languages, though, a logic variable cannot be redefined once it has acquired a value. Looked at another way, each statement in the program is a constraint that gives partial information about the value of x and the intersection of these constraints defines the value of x. Thus, the first statement is a constraint that is satisfied by any record with three fields *name, age* and *used.* The second statement is a constraint that is satisfied by any record whose *name* field contains the string *'Miranda'.* A similar interpretation holds for the other statements. By intersecting these constraints, we can deduce that x must be the record {*name = 'Miranda', age = 2, used = true*}. Thus, the record obtains its value not by the execution of a single expression but the co-operative effort of a number of statements in the program. The view-point of constraints provides a nice rationale for the prohibition of multiple assignments to record fields: if the programmer wrote *x.age = 2; x.age = 3*, the program should return the value 'error' since there is no record that satisfies both constraints. A benign case of multiple assignment occurs when a constraint occurs more than once in the program. It is permitted to write *x.age = 2* in two places in the program since there is no contradiction in repeating the same constraint. Appropriately enough, we will call these data structures '*monotonic records*': they are objects whose state changes as the program executes, but each state change increases the information in the object.

To understand how programs in such a language can be executed, let us replace the functional *cons* in language F with monotonic records and discuss how these programs can be executed on a dataflow interpreter. For our purpose, we do not need general records; therefore, we will introduce a single kind of monotonic record with four fields *car, cdr,*

```
program ::=
            f1(id,..,id) = letrec def...def in id,...,id
            ....
            fm(id,...,id) = letrec def...def in id,...,id

            ⇒fk(constant, constant,...constant)

lhs ::= id | id.car| id.cdr| id.d-car| id.d-cdr |
def ::= lhs,...,lhs = exp

exp ::= constant| id| id op id| if id then id else id|
        L-cons()| id.car| id.cdr| id.d-car| id.d-cdr |
        fj(id,..,id)|
        fork-i(id)
```

Figure 5: Syntax of Language FL

d-car and *d-cdr*. The role of the *d-car* and *d-cdr* fields will become clear later in this section. Figure 5 shows the syntax of this hybrid language, language FL. A record is allocated by the *L-cons* operation (for *L*ogical *cons*). Unlike the functional *cons* operation, this operation does not have any arguments since the values of the fields of a monotonic record do not have to be specified at the time of allocation. If x is a monotonic record, its fields may be read by the selection operations *x.car, x.cdr, x.d-car* and *x.d-cdr.* The fields may be given values by definitions of the form $x.car = e$ where e is some expression.

To implement FL programs on a dataflow interpreter, we need a structure memory similar to *I-structure memory*, which was discussed in Section 2.2, and dataflow nodes for allocating, reading and writing monotonic records. The *L-cons* operation is translated to a node labeled *allocate* in the dataflow graph, which allocates a record in structure memory and returns a pointer to it. The definition $x.car = y$ is translated into a node labeled *store-car* which has two inputs x and y. When this node receives tokens on both its inputs, it executes, sending a write request to the structure memory. If the *car* field of record x is empty, the structure memory simply writes the value of y into that field; otherwise, it unifies this value with the value stored in the *car* field. If unification fails, the entire program aborts. The *store-car* operator has no output since its purpose is to store a value into the *car* field of a record. The expression *x.car* is translated into a dataflow node labeled *car* with input x. When this operator receives a pointer to a record, it executes and sends a read request for the *car* field of x. If the *car* field has not been written into, the structure memory delays this request until the write happens [4]. The translation of operations on the other fields of a record are similar.

We are currently working on a denotational semantics for this language. This semantics uses the viewpoint of constraints described earlier to give abstract meanings to programs.

4.2 Demand propagation

To understand the relevance of monotonic records to demand propagation, it is useful to consider the following analogy. A graduate student has a committee of three professors and to get his degree, he has to fulfill the demands that all three professors make on him. How does he go about figuring out what the professors demand of him? One way is to let each professor make up a list of his demands. Some activities may be demanded by more than one professor (presumably, 'write your thesis' is in all three lists) while others may be in only one list ('teach CS999 while I'm way'). To figure out the requirements for graduation, the student must 'merge' all the three lists together, eliminating duplicate demands. This is analogous to the functional solution of Figure 4. How can the merging of demand lists be avoided? One way is for the student to make out a single 'check-list' of all the activities he can perform, with a box next to each item on this list. This list is circulated to all three professors who make their demands known by placing an 'X' in the appropriate boxes. To figure out what is expected of him, the student can simply read this check-list - by eliminating separate demand lists, we eliminate the need for merging these lists together.

How can we use this analogy for the problem of propagating demands in F programs? Instead of requiring that each consumer of a data structure generate a separate demand structure, there must be a single demand structure which is defined co-operatively by all the consumers. In fact, we can go further and combine the data structure and the demand structure into a single, composite structure that has both data values and demand values in it. This is achieved by implementing a cons cell as a monotonic record of four fields *d-car, car, d-cdr* and *cdr* (for *d*emand for car, *car*, etc.). This record should be thought of as a 'mail-box' shared by the producer and the consumers. The mail-box is allocated by the producer and a pointer to this mailbox is passed to any consumer that wishes to make a demand. The *d-car* and *d-cdr* fields of the mail-box are read by the producer to monitor demands from the consumers. A consumer that wants the value of the *car* field stores the value 'd' in the *d-car* field and reads the *car* field of the record. The 'd' value stored in the *d-car* field is read by the producer which propagates this demand to the computation responsible for producing the value of the *car* field. When this value is computed, the

[4]There is a close connection between read operations on monotonic records and unification of read-only variables in Concurrent Prolog [20].

producer stores it in the *car* field of the mail-box from where it is read by the consumer. Multiple demands for the *car* or *cdr* of a cons-cell show up as multiple writes of the value 'd' into the *d-car* or d-cdr field. This can be handled by a trivial case of unification which checks that these constants are indeed the same.

With this intuitive picture in mind, let us examine the new transformation shown in Figure 6 which shows how demand propagation code is introduced into F programs. As before, if x is an atomic value, a demand token on line dx represents a demand for its value. If x is the output of a *cons* operation, a demand token on line dx represents a demand for allocation of the 'mail-box' through which demands and data are to communicated. There are no demand structures - so the problem of unifying them does not arise. Not surprisingly, the differences between this transformation and the earlier one shown in Figure 4 show up only in the demand propagation code for data structures. Let us look at the code for the *cons*. The mail-box is allocated when a demand arrives for the output of the *cons* and a pointer to this mail-box is produced as output. Read requests are sent to the *d-car* and *d-cdr* fields to monitor demands from the consumers; if and when these fields are written into by the consumers, these demands are propagated to y and z respectively. Once y or z is computed, its value is stored in the appropriate field of the mail-box. Note that the *car* and *cdr* fields are written into at most once. Let us now examine the code for the *car* operation. A demand for the output of the *car* operation is propagated as a demand for the input of the *car* - *i.e.*, a demand is made for the allocation of the mail-box. When a pointer to this mail-box is received, a d value is stored into the *d-car* field and a read request is sent for the *car* field. If the *d-car* field was empty (*i.e.*, this is the first demand for the *car* field), the *d-car* field gets set to the value d, the producer reads this value and the computation for the *car* field is begun. If the *d-car* field already had the value d in it, the write request is discarded. The value of the *car* field is returned when it is written into by the producer.

Figure 7 shows a fragment of an F function and the dataflow graph for the transformed program. The *allocate* operator executes when the function is invoked producing a pointer to a record in structure memory. Let us refer to this record as R. The token carrying this pointer is held at the *gate* operator until a demand is made either for p or for q. Suppose a demand is made for p. The demand token on line dp is propagated by the *lub-2* operator to the *gate* operator, which causes a token carrying a pointer to R to appear on line x and then on line $x1$. A 'd' value is stored in the d-car field of R, thereby posting a read request for the *car* field. This value is read by the *d-car* operator and a demand token is produced on line da. As a result, a token carrying the value '5' is sent to the *store-car* operator. This operator executes, storing the value 5 in the car field of R. This value is returned to the *car* operator which produces a token carrying the value 5 on line p. Subsequently, if a demand token is sent on line q, a token carrying a pointer to R is produced on line $x2$. The value d is written into the d-car field of R, but this write is ignored. The *car* operator

Definition in Source	Definition in Transformed Program
x = constant	x = gate(constant,dx)
x = y	x = y dy = dx
x = y op z	x = y op z dy = dx dz = dx
x = if a then b else c	x = if a then b else c da = dx db,dc = switch(a,dx)
x = cons(y,z)	x = gate(L-cons(), dx) x.car = y x.cdr = z dy = x.d-car dz = x.d-cdr
x = car(y)	x = y.car dy = dx y.d-car = d
x = cdr(y)	x = y.cdr dy = dx y.d-cdr = d
x1,..xn = fork-*n*(y)	t1,...,tn = fork-*n*(y) x1 = gate(t1,dx1) ... xn = gate(tn,dxn) dy = lub-*n*(dx1,...,dxn)
x1,...,xn = fi(y1,...,ym)	x1,...,xn,dy1,...,dym = Tfi(y1,...,ym,dx1,...,dxn)

Figure 6: Using Logic variables for Demand Propagation

executes, producing the value 5 on line q.

The use of logic variables for demand propagation provides a nice example of what Lindstrom has called the 'action-at-a-distance' feature of logic variables[16].

4.3 Discussion

Transformed programs do not need the full power of unification. Since the source language is a functional language, multiple writes cannot occur in the *car* and *cdr* fields of monotonic records in transformed programs. *d-car* and *d-cdr* fields may be written into more than once; since the values being written must be the atomic constant d, this can be handled by ignoring multiple writes. Although transformed programs do not need the full power of unification, we do not see any way to mimic this style of demand propagation in a purely functional language. Monotonic records acquire their values by the co-operative effort of a number of statements in the program and we have put this feature to use to eliminate demand structures from the demand propagation algorithm. However, one can ask whether a compiler might be able to analyze a program produced by the functional transformation and 'optimize' it to produce the program that would have resulted from the new transformation. We leave this as an open question.

It is straight-forward to extend the transformation of Figure 6 to other constructs such as a *nil* constructor and higher-order functions. Let us enrich the source language with a constructor *nil* and an *is-nil?* function which returns *true* if its argument is *nil* and *false* otherwise. Suppose x = *is-nil?(y)* is a definition in the source program and y is the output of a *cons* operator. How would a demand for x be propagated to a demand for y? The functional demand propagation algorithm runs into difficulties because demands can be made only for the elements of y, but these elements are of no interest to the *is-nil?* function. Using the new approach, demand propagation is straight-forward - the demand propagation code is simply $dy = dx$. Propagating a demand token on line dy will cause the address of a mail-box to be produced on line y and the *is-nil?* operator executes, producing *false* as the output.

Higher-order functions pose no problems since they can be treated like the *cons* operation. We assume that all functions are curried and that lambda-lifting has been done [12]. If *(f a)* is a partial application, a demand for its output is propagated as a demand for *f*. When the value of *f* is obtained, its arity is checked. If the arity is one, the function application is carried out; otherwise, a mail-box is allocated (this is like an *apply* node in the G-machine) and the value of *f* is written into the *car* field of the mail-box. The *d-cdr* and *cdr* fields are used to communicate data and demand for the argument a.

. . . .

a = 5

x = cons(a,b)

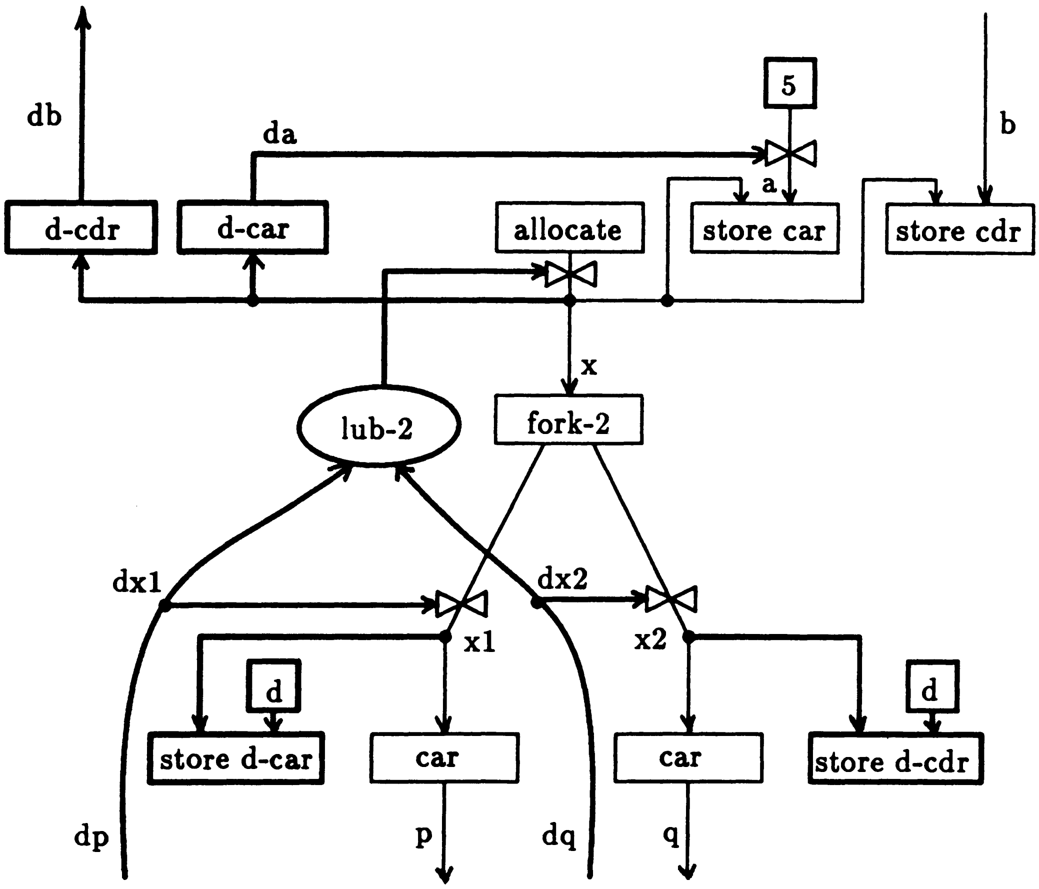
x1,x2 = fork-2(x)

p = car(x1)

q = car(x2)

. . . .

(a) Fragment of Source Program



(b) Program Transformed According to Figure 6

Figure 7: Propagating Demands Using the New Algorithm

5 Applications

In this section, we describe two applications of the transformation of Figure 6. First, we show how this transformation can be used to implement lazy evaluation on a real dataflow machine. Second, we discuss some connections with algorithms for strictness analysis.

5.1 Implementing lazy evaluation on a dataflow machine

Apart from two minor complications, it is straight-forward to implement the transformation described in the previous section on the tagged-token dataflow machine[2]. I-structure memory as originally defined did not permit multiple writes into the fields of a record. The transformation described above requires multiple writes into the *d-car* and *d-cdr* fields to be handled by a trivial case of unification (unification of constants). This involves a minor change to I-structure memory. Secondly, dataflow instructions in this machine do not have any local 'state' associated with them that carries information from one firing to the next. Examining the transformation of Figure 6, we see that there are two operators which have such state: *cond* and *lub*. As we have described it, *cond* fires twice, once to produce the output and once to absorb the input that was not needed. This means that the value of the predicate must be stored somewhere from one firing to the next. Notice though that under lazy evaluation, the *cond* operator is not required since a token will arrive only the input that is needed to produce the output of the *cond* operator. Thus, in the implementation, the *cond* operator is eliminated and its two data inputs are 'tied together' (i.e., they become the same line in the dataflow graph). Similarly, the *lub* operator cannot be implemented as a primitive since information that a *d* token has been produced must be carried from one firing to the next. The *lub* operator is implemented using the *d-car* field of a record in memory. A definition of the form x = *lub(a,b)* is implemented as the scheme

```
r = L-cons();
r.d-car = a;
r.d-car = b;
x = r.d-car;
```

The tagged token dataflow machine permits different invocations of a function to share the same code by tagging tokens to distinguish tokens belonging to one activation from those belonging to another. The execution of programs produced by the transformation of Figure 6 is compatible with tag manipulation on the dataflow machine - notice that during function invocation, at most one token flows down an arc in the dataflow graph for the function.

Garbage collection poses an interesting problem in this implementation. Dataflow graphs for data-driven execution have the self-cleaning property [4] - once computation has terminated in some function activation, no tokens are left behind. This is useful because tokens are held in an associative store (called the waiting-matching section) and space in this store is limited. Unfortunately, dataflow graphs generated by our transformation are not self-cleaning - for example, if a demand is made only by one output of a fork but not the other, a token will sit at the *gate* operator unless it is 'garbage collected'. While it possible to come up with schemes for collecting such tokens, recent developments in the architecture have made the problem moot. To circumvent the problem of building a large associative store, the tagged token dataflow machine has been redesigned so that a waiting-matching section is not required. Instead, an *activation record* in ordinary memory is assigned to each function activation. This is very much like a stack frame in conventional implementations of sequential languages, and holds operand values during the execution of the function activation. In this architecture, activation records can be garbage collected with structure memory.

5.2 Comparison with Graph Reduction

The seminal work on parallel, lazy graph reduction interpreters was done by Keller and Lindstrom in their AMPS and Rediflow projects [15,14]. This work has inspired much of the current effort in parallel implementations of lazy evaluation. In their system, the program is stored as a directed graph of instructions which gets over-written by data as the program executes. Each instruction has an opcode, pointers to other instructions that produce its operands and two bits called the 'demand bit' and the 'ready' bit. The output of an instruction is demanded by putting its address on a global 'demand list' (task queue). A free processor picks up this instruction, sets its demand bit and attempts to execute it. If the operands have not been computed, their addresses are placed on the demand list and the instruction is discarded. When an instruction is evaluated, it is over-written by the result and the 'ready bit' of the instruction is set. Addresses of instructions that are waiting for this result are placed on a 'result list'. These instructions are then rescheduled for execution. The demand bit is used to ensure that an instruction is not queued multiply on the demand queue.

This approach differs from the one described in this paper in several ways. In our approach, instructions are not over-written by data. This permits different invocations of a function to share the same copy of code, provided tokens are 'tagged' to distinguish tokens belonging to one activation from those that belong to other activations. It may be possible to achieve this in the AMPS model by separating operand storage from instruction storage by giving each function invocation its own 'activation record' for storing operands. This would represent a step away from graph reduction. In our approach, data and demands are treated uniformly - these values are carried on tokens and an instruction is scheduled when it receives a token, regardless of whether the token represents data or demand. In essence, we have merged the 'demand list' and 'result list', which simplifies the design of the processor. The *d-car* and *d-cdr* fields in our approach play a role similar to demand bits in AMPS. However, we introduce them only at *forks* and in data structures; *i.e.*, where data is shared, rather than in every instruction. Checking the demand bit does not have to part of the instruction execution cycle. In some sense, ours is a 'RISC' approach to lazy evaluation! In a processor designed to implement lazy graph reduction, the interpretation of instructions is complicated by the fact that an instruction must be processed in two different ways: one set of actions must be taken if the instruction is being processed in response to a demand, while a different set of actions must be taken if the instruction is being processed because its inputs have arrived. In our approach, the processor is freed from these concerns since demand propagation code is exposed by the compiler.

5.3 Connections with Strictness Analysis

One problem with completely lazy evaluation is that parallelism is limited [6]. We emphasize that this is a problem in our implementation as much as it is in an implementation based on graph reduction. On the other hand, the advantage of exposing demand propagation code is that this code becomes subject to standard program optimization which can take the place of strictness analysis. Examining the program in Figure 3, we see that there is considerable scope for optimization. At run-time, a 'd' token must flow on line *do* to indicate demand for the output of the function. Using constant propagation, we see that the *lub-2* operator and the *gate* operator on line *y1* can be removed, and that the demand on line *do* can be propagated directly to *dx* and *dy*. A similar conclusion can be drawn using 'strictness analysis' [6,10] of this program. Traditionally, strictness analysis algorithms have been based on abstract interpretation, but their complexity can be exponential in the size of the program. We believe that it should be possible to view strictness analysis as an optimization of the demand propagation graph, using standard techniques such as constant propagation and value subsumption [1]. Results in this direction are reported by Amamiya et al who have used global dataflow analysis to optimize the demand propagation code generated by our functional transformation [17].

Interestingly enough, support for this thesis comes also from a new strictness analysis algorithm called 'backward strictness analysis' [11]. This algorithm exhibits much better complexity than the traditional algorithms based on abstract interpretation and uses a technique similar to our functional transformation. Backward analysis is presented using 'contexts' which correspond roughly to what we have called demands. Strictness analysis is performed by associating a 'context transformer' with each function; context transformers correspond to demand propagation code. The formalization of contexts and context transformation appears to require a fair amount of mathematical machinery such as lifted domains and projections. It is possible that viewing backward analysis in terms of optimization of demand propagation code may simplify the presentation. Moreover, the improved demand propagation algorithm shown in Figure 6 suggests that this analysis might profitably be performed in a language with logic variables. We are currently investigating this problem.

6 Conclusions

We have argued that logic variables are required to explicate the process of demand propagation in lazy evaluation of pure functional languages. This result provides a new approach to the old problem of efficient analysis and implementation of laziness. It offers a simple, 'RISC'-like parallel implementation of lazy evaluation in which data and demands are handled uniformly by the processor. In addition, it suggests that strictness analysis could be viewed fruitfully as conventional optimization of demand propagation code.

References

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

[2] Arvind and David Culler. *Dataflow Architectures.* Technical Report C.S.G. memo TM-294, M.I.T. Laboratory for Computer Science, February 1986.

[3] Arvind and K. Gostelow. The U-interpreter. *Computer*, February 1982.

[4] Arvind, K. P. Gostelow, and W. Plouffe. *An Asynchronous Programming Language and Computing Machine.* Technical Report 114a, University of California, Irvine, December 1978.

[5] Arvind, R. Nikhil, and K. Pingali. *I-structures: data structures for parallel computing.* Technical Report 87-810, Cornell University, Februrary 1987.

[6] C. Clack and S. Peyton-Jones. Strictness analysis - a practical approach. In *Proceedings of the Functional Languages and Architectures Conference*, 1985.

[7] J. B. Dennis. First version of a data flow procedure language. In *Proceedings of the Colloque sur la Programmation, Vol. 19, Lecture Notes in Computer Science*, pages 362–376, 1974.

[8] D. Friedman and D. Wise. Cons should not evaluate its arguments. In *Automata, Languages and programming*, 1976.

[9] B. Goldberg and Paul Hudak. Alfalfa: distributed graph reduction on a hypercube multiprocessor. In *Proceedings of the Los Alamos Workshop on Graph Reduction*, 1986.

[10] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, 1986.

[11] J. Hughes and P. Wadler. Projections for strictness analysis. In *Proceedings of the Functional Languages and Architectures Conference*, 1987.

[12] T. Johnsson. Efficient compilation of lazy languages. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, 1984.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, 1974.

[14] R. Keller. Rediflow multiprocessing. In *Proceedings of Compcon 84*, 1984.

[15] R. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. In *Proceedings of the National Computer Conference*, 1979.

[16] G. Lindstrom. Functional programming and the logical variable. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, 1985.

[17] S. Ono, N. Takahashi, and M. Amamiya. Optimized demand-driven evaluation of functional programs on dataflow machines. In *Proceedings of the 1986 ICPP*, pages 421–428, 1986.

[18] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1986.

[19] K. Pingali and Arvind. Efficient demand-driven evaluation. *ACM TOPLAS*, April 1985.

[20] E. Shapiro. *A subset of concurrent Prolog and its interpreter*. Technical Report TR-003, Institute for new generation computer technology, 1983.

[21] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of the 1985 Functional Languages and Architectures Conference*, 1985.

[22] D. Turner. The semantic elegance of functional languages. In *Proceedings of the 1981 Functional Languages and Architectures Conference*, 1981.

[23] D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 3(9):31–49, 1979.