International Conference on Fifth
Generation Computer Systems 1988
Nov. 28 - Dec. 2, 1988 Tokyo, Japan

LAZY EVALUATION OF FP PROGRAMS: A DATA-FLOW APPROACH[1]

Yi-Hsiu Wei

Computer Science Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Jean-Luc Gaudiot

Computer Research Institute
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California

ABSTRACT

This paper presents a lazy evaluation system for the list-based functional language, Backus' FP in data-driven environment. A superset language of FP, called DFP (Demand-driven FP), is introduced. FP eager programs are transformed into DFP lazy programs which contain the notions of demands. The data-driven execution of DFP programs has the same effects of lazy evaluation. DFP lazy programs have the property of always evaluating a sufficient and necessary result. The infinite sequence generator is used to demonstrate the eager-lazy program transformation and the execution of the lazy programs.

1 INTRODUCTION

Lazy evaluation and *eager evaluation* are two methods for the execution of functional programs (Vegdahl 1984). The execution in *lazy evaluation* is driven by the need for function values (*call by need*). This has also been termed *demand-driven*. On the other hand, the execution in *eager evaluation* is driven by the availability of the function arguments (*call by value*). It is often termed *data-driven* and has spurred the development of several *data-flow* architecture models (Veen 1986). Lazy evaluation can handle unbound data structures such as infinite lists and makes interactive input/output possible (Johnson 1984)(Friedman and Wise 1976). Besides, it may support a better execution efficiency since only the necessary computation is performed in the execution. These characteristics are not shared by eager evaluation.

Lazy evaluation can be implemented by two methods: (a) *The normal order reduction* treats a program as a syntactic object. The execution is a sequence of "normal order" rewrite processes to the program that successively replaces the outermost functions by the function definitions at each step of execution until the result value is obtained (Kennaway and Sleep 1984); (b) The *data-flow execution with demand-driven semantics* treats a program as a data-flow graph with backward demand arcs as well as forward data arcs at function nodes. The execution relies on both demand and data propagations through function nodes on this mixed demand/data-flow graph (the lazy graph). With demand propagations, the effects of the evaluation "by need of the function value" is obtained.

[1]This material is based upon work supported in part by the U.S. Department of Energy under Grant No. DE-FG03-87ER25043

Several important results in the second approach have been obtained in previous work (Pingali and Arvind 1985) (Amamiya and Hasegawa 1984). The lazy graph schemas and the graph properties for a stream language L have been studied extensively. However, the efforts have mainly been on the graph level analysis. Due to the lack of a formal description of lazy programs, the precise meaning of the lazy programs dealing with more general data-structures other than streams such as nested lists is not clear.

In this paper, we deal with the complex data structures in the second approach. The study involves the defining of the lazy version of languages, the eager-lazy program transformation, and the lazy graph generation. We use the Backus' FP (Backus 1978) as a source language since it is a nested-list based language with a simple syntax and semantics. The methodology developed here will be applicable to the development of lazy evaluation systems in data-driven environment for the other first order functional languages.

In section 2, a basic FP system is briefly reviewed. In section 3, the lazy counterpart of FP, DFP (Demand-driven FP) and the eager-lazy (FP-DFP) program transformation are defined. In section 4, an example of program transformation and execution is given. In section 5, the lazy graph schemas and an example of lazy graph generation are described. Conclusions are made in section 6.

2 FP PRELIMINARY

An FP system has: 1) a set of objects O_{FP}, 2) a set of primitive functions, and 3) a set of functional forms. Every FP function is monadic (requires one object). An FP function maps an object into an object. The function application of a function f to an object in O_{FP}, denoted by $f : x$, is the basic operation.

A. Objects: The set of objects is defined recursively with a given set A of atoms: (1) All atoms, the $\perp$ (an undefined object), and the empty sequence <> are objects; (2) $< x_1, \ldots, x_n >$ is an object provided that x_i (for all $i = 1, \ldots, n$) is an object. If any $x_i = \perp$, $< x_1, \ldots, x_n >= \perp$.

B. Primitive functions: Either the function input or the function value is a single object. Every primitive function is *bottom preserving*. That is $f : \perp = \perp$.

Object reformat functions:
$\{apndl, apndr, trans, reverse, rotl, rotr\}$

DISCLAIMER

$apndl : x \equiv x =< x_1, <> > \rightarrow < x_1 >;$
$x =< x_1, < y_1, \ldots, y_n > > \rightarrow < x_1, y_1, \ldots, y_n >; \perp$
$reverse : x \equiv x =<> \rightarrow <>;$
$x =< x_1, \ldots, x_n > \rightarrow < x_n, \ldots, x_1 >; \perp$

Select functions: $\{n, nr, tl, tlr\}$
$n : x \equiv x =< x_1, \ldots, x_m > \& m \geq n \geq 1 \rightarrow x_n; \perp$
$tl : x \equiv x =< x_1 > \rightarrow <>;$
$x =< x_1, \ldots, x_n > \& n \geq 2 \rightarrow < x_2, \ldots, x_n >; \perp$

Broadcast functions: $\{distl, distr\}$
$distl : x \equiv x =< x_1, <> > \rightarrow <>;$
$x =< x_1, < y_1, \ldots, y_n > >$
$\rightarrow << x_1, y_1 >, \ldots, < x_1, y_n > >$

Arithmetic, Logic, Predicate, and other functions:

$\{+, -, \times, \div, gt, le, and, or, not, eq, atom, null, length, id\}$
$+ : x \equiv x =< x_1, x_2 > \& \; x_1, x_2$ are numbers
$\rightarrow x_1 + x_2; \perp$
$id : x \equiv x$

C. Functional forms: Functional form is the mechanism for combining known functions to obtain more complex functions.

Composition: $(f \circ g) : x \equiv f : (g : x)$
Construction: $[f_1, \ldots, f_n] : x \equiv < f_1 : x, \ldots, f_n : x >$
Conditional: $(p \rightarrow f; g) : x \equiv (p : x) = T \rightarrow f : x;$
$(p : x) = F \rightarrow g : x; \perp$
Apply-to-all:
$\alpha f : x \equiv x =<> \rightarrow <>;$
$x =< x_1, \ldots, x_n > \rightarrow < f : x_1, \ldots, f : x_n >; \perp$
Insert:
$/f : x \equiv x =< x_1 > \rightarrow x_1;$
$x =< x_1, \ldots, x_n > \& \; n \geq 2$
$\rightarrow f :< x_1, /f :< x_2, \ldots, x_n >>; \perp$
Constant: $\overline{y} : x \equiv x = \perp \rightarrow \perp; y$

3 DFP: DEMAND-DRIVEN FP

The principle of lazy evaluation is: only the 'necessary and sufficient' arguments be requested and used to compute only the 'necessary and sufficient' values in response to the request of the consumer. Since every function in an FP system is monadic, either the function argument or the function value is a single object. In order to follow the principles of lazy evaluation, functions in FP should be allowed to be 'non-strict' and the function argument and value objects should be able to carry a *partial* information. This partial information would be "necessary and sufficient' for the proper evaluation of the program. This leads to semantics for the *partial* request and production of function values.

An extension of the language is required to support these operational semantics. The domain of objects in the original FP systems have to be extended to include an extra atom ϵ in the set A of atoms. An ϵ represents an unknown object that has not yet been evaluated. The extended object domain will then contain sequences in which ϵ occurred in one or more places. These sequence objects are *partial objects*. The sequence objects which contain no ϵ atom are *total objects*. Each total object, an original FP object, and its associated partial objects make up a partially ordered object set. An example is shown in Fig. 1 which shows the partially ordered object set for the total object $< 1,2 >$. The partial objects $< 1,\epsilon >$ and $< \epsilon,2 >$ contain different partial information of $< 1,2 >$. A single ϵ denotes the totally unevaluated $< 1,2 >$. The $\perp$ below ϵ is the *undefined* object that may represents the erroneous condition, namely the undefined computation. Note that ϵ is different from the empty sequence $<>$ (or ϕ) in FP where ϵ is a notation representing the *unevaluated* object which could be any data object.

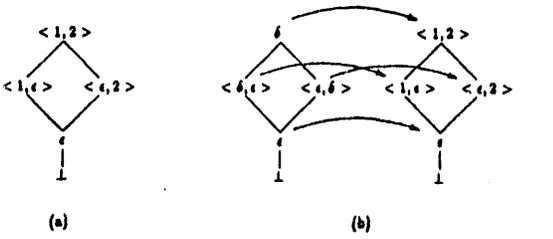

Figure 1: (a) Partially ordered set for $< 1,2 >$; and (b) The object function for $< 1,2 >$.

If a function f with an argument $< 1,2 >$ evaluates to a value $< 2,4 >$ in a data-driven fashion, the function value could however be partially requested and evaluated in a demand-driven (lazy) fashion. When only the second element of the function value is of interest, a demand on the partial object could initiate a partial object evaluation and cause the production of partial object $< \epsilon,4 >$ instead of the entire $< 2,4 >$. The portion(s) of the program which do not contribute to the computation of this partial object are deemed not necessary to the program and should not be evaluated. Furthermore, only the second element of function argument must be requested by a demand to the parent function if the first element is irrelevant to the computation.

The addition of the 'ϵ' atom to the language enables the formalization of the demand-driven execution of FP programs. This formalism is defined through a superset language of FP, Demand-driven FP (DFP). An eager-lazy program transformation will create demand-annotated DFP lazy programs from FP programs. This DFP program could be executed directly by a data-driven interpreter for DFP.

In addition to partial objects, the DFP object domain has a set of *demand objects* with which the 'demand to the partial object' can be represented. The set of DFP primitive functions include data functions (FP primitives) and *demand functions*. A demand function determines the demand propagation for the corresponding FP primitive function. A demand function takes a demand object and evaluates to a demand object of a proper form. A data function takes a data object and evaluates to a data object. An object function is a function that takes a demand object and evaluates to a data object. A DFP lazy program represents an object function. A object function is constructed by data functions, demand functions, and object functions. The FP-DFP transformation rules for eager-lazy program transformation define the DFP function forms for FP primitive functions, and various FP functional forms.

3.1 DFP objects

The set O_D of objects in DFP is the union of the set O_a of data objects and of the set O_d of demand objects, $O_D = O_a \cup O_d$, where O_a and O_d are recursively defined: $O_a = \{atoms, \varepsilon\} \cup \{\perp\} \cup \{< x_1, \ldots, x_n > | n \in N, x_i \in O_a - \{\perp\}, i = 1, \ldots, n\}$; and $O_d = \{\delta, \varepsilon\} \cup \{< d_1, \ldots, d_n > | n \in N, d_i \in O_d, i = 1, \ldots, n\}$ Where N is the set of natural numbers.

Objects in O_a are data objects (for examples: $< 1,2 >, < 1, \varepsilon >$) The data objects which contain no ε atom are *total data objects* (e.g. $< 1,2 >$), otherwise they are *partial data objects* (e.g. $< 1, \varepsilon >$). Objects in O_d are demand objects (for example: $< \delta, \varepsilon >$) [1]. Let $O_t = \{\delta\} \cup \{< d_1, \ldots, d_n > | n \in N, d_i \in O_t, i = 1, \ldots, n\}$ and $O_e = \{\varepsilon\} \cup \{< d_1, \ldots, d_n > | n \in N, d_i \in O_e, i = 1, \ldots, n\}$ The demand objects in O_t are called *total demand objects*(for examples: δ, $< \delta, \delta >$). The demand objects in O_e are *empty demand objects* (for examples: ε, $< \varepsilon, \varepsilon >$). The demand objects in $O_d - O_t - O_e$ are *partial demand objects* (e.g. $< \delta, \varepsilon >$). $d \in O_t$ is replaceable by δ (for example: $< \delta, \delta > \rightarrow \delta$. Therefore, δ may represent any total demand object. Similarly, any d in O_e is replaceable by ε and thus ε may represent any empty demand object.

For notational convenience, $d \in O_d$, $< d^n >$ denotes $< \underbrace{d, \ldots, d}_{n} >$ and $< d^+ >$ denotes $< \underbrace{d, \ldots}_{m} >$. Where m is an undetermined natural number. For examples:

1. $< \delta^3 > = < \delta, \delta, \delta >$
2. $< \varepsilon, \delta^2, \varepsilon^+ > = < \varepsilon, \delta, \delta, \varepsilon, \ldots >$

The demand object in the first example represents a demand vector for a triplet of data where all three data elements are requested. In the third example, only the $2nd$, and $3rd$, elements of a list of indefinite length are requested.

3.2 DFP primitive functions

The set P_D of primitive functions in DFP is the union of:

1. The set P_d, $P_d \subset [O_d \rightarrow O_d]$, of *demand functions*;
2. The *object function* $\tilde{x}$, $\tilde{x} \in [O_d \rightarrow O_a]$;
3. A *mask function*, $mask \in [O_a \times O_d \rightarrow O_a]$; and
4. The set P_a, $P_a \subset [O_a \rightarrow O_a]$, of *data functions* (primitive functions in FP).

All functions f in DFP verify the *bottom-preserving* property. The extended definitions of some of FP primitive functions will also be described.

A. Object function $[O_d \rightarrow O_a]$: An object function defines a one-to-one mapping from a set of demand objects into a set of data objects. For example, an FP object $x = < 1,2 >$ is transformed into the DFP object function $\tilde{x}$ which defines a set of mappings (Fig. 1b):

$\{(\delta \rightarrow < 1,2 >), (< \delta, \varepsilon > \rightarrow < 1, \varepsilon >), (< \varepsilon, \delta > \rightarrow < \varepsilon, 2 >), (\varepsilon \rightarrow \varepsilon)\}$

The first δ in the first mapping is interpreted as $< \delta, \delta >$. The range set is the partially ordered data object set for $< 1,2 >$, while the domain set is the corresponding partially ordered demand object set. When an object function is applied to a demand object, a unique data object will be mapped. The object function is therefore considered as a producer of data objects driven by demand objects. For example, assuming $x = < 1,2 >$, if demand d is *total*, δ, the application $\tilde{x} : d$ evaluates to *total data object* $< 1,2 >$. Otherwise, if demand d is *partial*, such as $< \varepsilon, \delta >$, $\tilde{x} : d$ evaluates to *partial data object* $< \varepsilon, 2 >$. If demand d is *empty*, ε, $\tilde{x} : d$ evaluates to a totally unevaluated object ε.

Definition: An FP data object x is transformed by operator ρ into a DFP object function $\tilde{x}$. This object function is defined by:

$$\rho(x) : d \equiv \tilde{x} : d \equiv d = \delta \rightarrow x; d = \varepsilon \rightarrow \varepsilon; \\ d = < d_1, \ldots, d_n > \& x = < x_1, \ldots, x_n > \\ \rightarrow < \tilde{x_1} : d_1, \ldots, \tilde{x_n} : d_n >; \\ \perp$$

A demand object *matchs* with a data object when the demand object is: 1) a total demand or an empty demand, or 2) of the same length as the data object and every component demand object matches with a corresponding component data object. If a object function is applied to a mismatched demand, it evaluates to a $\perp$. For examples, if $x = < 1,2 >$, $\tilde{x} : \delta = < 1,2 >$ (matched demand), $\tilde{x} : < \varepsilon, \delta > = < \varepsilon, 2 >$ (matched demand), and $\tilde{x} : < \delta, \delta, \varepsilon > = \perp$ (mismatched demand, since the demand object has three elements while the data object in $\tilde{x}$ has only two elements 1 and 2).

B. Mask function:$[O_a \times O_d \rightarrow O_a]$ This function is used together with broadcast functions, such as *distl* and *distr*, to remove extraneous duplicated data (refer to FP-DFP transformation for *distl* and *distr* in the following subsection). It is defined by:

$$mask : < x, d > \equiv d = \delta \rightarrow x; d = \varepsilon \rightarrow \varepsilon; \\ x = < x_1, \ldots, x_n > \& d = < d_1, \ldots, d_n > \\ \rightarrow < mask : < x_1, d_1 >, \ldots, mask : < x_n, d_n >>; \\ \perp$$

Examples:

1. $mask : << 1,2,\varepsilon >, < \varepsilon, \delta, \varepsilon >> = < \varepsilon, 2, \varepsilon >$
2. $mask : << 1,2,3 >, < \varepsilon, \delta, \delta >> = < \varepsilon, 2, 3 >$

C. Demand Functions $[O_d \rightarrow O_d]$: A demand function corresponding to a data function evaluates the demand propagation for the data function. For example, a total demand δ to the scalar value of function $+$ is converted into a demand $< \delta, \delta >$. This demand is made to a two-element data object for function $+$.

1. **Arithmetic, Predicate, and Logic demand functions:** For all $f, f \in \{+, -, \times, \div, gt, le, eq, and, or, not, eq0, id, length, atom, null\}$

$$f^d : d \equiv d = \delta \rightarrow \delta; \\ d = \varepsilon \rightarrow \varepsilon; \perp$$

Since the function value of f is an atom, a demand object to f^d should be either δ or ε. An identity function *id* may serve as a weak definition of f^d. The difference between f^d and *id* is that *id* does not detect the mismatched demand object. This weak definition is useful since *id* is removable when it is either pre-composed or post-composed with *any* other function. While the detection for mismatched demands can be postponed.

2. **Reformat demand functions:** The functions *apndl*, *trans*, *rotl*, *reverse*, etc. are used to rearrange the object elements. The same as that of the nonstrict constructor *cons* in *LISP*, reformat functions should not force the evaluation of their arguments. Reformat demand functions should rearrange object elements in a reverse man-

[1] In the following, d (with or without subscripts) denotes a demand variable, while x denotes data variable.

ner to reflect the proper locations of the requested elements.

$apndl^d : d \equiv d = \delta \rightarrow \delta; d = \varepsilon \rightarrow \varepsilon;$
$d =< d_1, d_2, \ldots, d_n > \rightarrow < d_1, < d_2, \ldots, d_n >>;$
$\perp$

$trans^d \equiv trans$
$reverse^d \equiv reverse$
$rotl^d \equiv rotr$

Example:
$apndl^d :< \delta, \varepsilon, \delta >=< \delta, < \varepsilon, \delta >>$
(Note that $apndl :< a, < \varepsilon, b >>=< a, \varepsilon, b >$.)

3. **Select demand functions:** Select functions, such as n, tl, etc., are nonstrict since they do not make use of certain portions of the input object. They are similar to the selectors *car* and *cdr* in LISP. Select demand functions should create partial demands to request required partial objects.

$n^d : d \equiv d = \varepsilon \rightarrow \varepsilon; d \in O_d \rightarrow < \varepsilon^{n-1}, d, \varepsilon^+ >; \perp$
$tl^d : d \equiv d = \delta \rightarrow < \varepsilon, \delta^+ >;$
$d = \varepsilon \rightarrow \varepsilon;$
$d =< d_1, \ldots, d_n > \rightarrow < \varepsilon, d_1, \ldots, d_n >;$
$\perp$

Examples:
(a) $n^d : \delta =< \varepsilon^{n-1}, \delta, \varepsilon^+ >$
(b) $n^d :< \delta, \varepsilon >=< \varepsilon^{n-1}, < \delta, \varepsilon >, \varepsilon^+ >$
(c) $tl^d : \delta =< \varepsilon, \delta^+ >$
(d) $tl^d :< \delta, \varepsilon >=< \varepsilon, \delta, \varepsilon >$

In the first example, a demand function for the selector n converts a total demand δ into a demand object of undetermined length with a δ at the nth element and εs elsewhere.

4. **Broadcast demand functions:** *distl* and *distr* are considered as data broadcast functions.

$distl^d : d \equiv d = \delta \rightarrow \delta; d = \varepsilon \rightarrow \varepsilon;$
$d =< d_1, \ldots, d_n > \rightarrow apndl \circ [join \circ \alpha 1, \alpha 2] : d;$
$\perp$

$distl^d$ performs the reverse operation as the *distl* does. However, the demand for the first element, which is the one to be copied, is the *join* of all demands for the individual copies.

Examples:
(a) $distl^d :< \delta, \varepsilon, \delta >=< \delta, < \delta, \varepsilon, \delta >>$
(b) $distl^d :<<< \delta, \varepsilon, \varepsilon >, \delta >, << \varepsilon, \varepsilon, \delta >, \delta >$
$, \varepsilon >=<< \delta, \varepsilon, \delta >, < \delta, \delta, \varepsilon >>$

Where the *join* function performs the union of all demands for copies of every subelement of the demand object. It works like an bitwise OR operation on two sequences of the same length where δ is a boolean *True* and ε is a boolean *False*. In the second example, $< \delta, \varepsilon, \delta >$ is the *join* of $< \delta, \varepsilon, \varepsilon >$ and $< \varepsilon, \varepsilon, \delta >$.

D. Extended definitions of FP functions: The definition of several data functions is extended to allow handling demand objects. For examples:

1. *trans* and *reverse*.
2. $n : d \equiv d = \delta \rightarrow \delta; d = \varepsilon \rightarrow \varepsilon;$
$d =< d_1, \ldots, d_m > \& \; m \geq n \geq 1 \rightarrow d_n; \perp$
3. $tl : d \equiv d = \delta \rightarrow \delta; d = \varepsilon \rightarrow \varepsilon;$
$d =< d_1, \ldots, d_n > \& n \geq 2 \rightarrow < d_2, \ldots, d_n >; \perp$
4. For any FP function f, $f : \varepsilon = \varepsilon$
5. $apndl :< x, \varepsilon >=< x, \varepsilon^+ >$

3.3 FP-DFP Transformations

The transformation of FP objects to DFP object functions is defined by transformation operator ρ, $\rho : O_{FP} \rightarrow [O_d \rightarrow O_a]$. Each FP object can be transformed into a DFP object function. The FP object to which the FP function application $f : x$ evaluates is also transformed to its corresponding DFP object function $\rho(f : x)$. This object function represents, in fact, the DFP lazy program that corresponds to $f : x$ in FP. When this DFP program has a total/partial demand object as its input, it generates a total/partial data object. Note that the total data object is $f : x$. ρ is defined as follows:

- For $x \in O_{FP}$, $\rho(x) \equiv \tilde{x}$. ρ maps an FP object x into DFP object function $\tilde{x}$. (Refer to the example of the object function for $< 1, 2 >$ in section 3.2.)
- For an FP function f, its argument object $x \in O_{FP}$, and function application $f : x$ (Fig. 2a), $\rho(f : x) \equiv (f \circ \rho(x))^* \equiv (f \circ \tilde{x})^*$ (Fig. 2b).

The expressions marked with asterisks are recursively defined as follows:

$(f \circ \tilde{x})^* \equiv$
$f = f_1 \circ f_2 \rightarrow (f_1 \circ (f_2 \circ \tilde{x})^*)^*;$
$f = (f_1 \rightarrow f_2; f_3) \rightarrow ((f_1 \circ \tilde{x})^* \rightarrow (f_2 \circ \tilde{x})^*; (f_3 \circ \tilde{x})^*);$
$f = [f_1, \ldots, f_n] \rightarrow [(f_1 \circ \tilde{x})^* \circ 1, \ldots, (f_n \circ \tilde{x})^* \circ n];$
$f = \alpha f' \rightarrow ((apndl \circ [f' \circ 1, \alpha f' \circ tl]) \circ \tilde{x})^*;$
$f = /f' \rightarrow ((f' \circ [1, /f' \circ tl]) \circ \tilde{x})^*;$
$f = \overline{y} \rightarrow \overline{y};$
$f \in \{distl, distr\} \rightarrow mask \circ [(f \circ \tilde{x})^*, id];$
$f \in P_a - \{distl, distr\} \rightarrow f \circ \tilde{x} \circ f^d;$
$\perp$

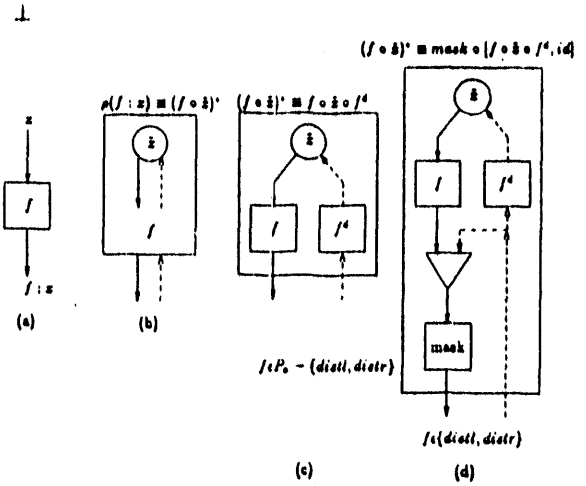

Figure 2: (a) FP function application $f : x$; (b) DFP function for $f : x$; (c) Transformation for $f \in P_a - \{distl, distr\}$; and (d) Transformation for $f \in \{distl, distr\}$.

The FP function application $f : x$ in Fig. 2a is transformed into $(f \circ \tilde{x})^*$ as shown in Fig. 2b in which the solid lines are data paths and the dashed lines are demand paths. The transformation rules involving expressions marked with an asterisk are determined according to the following observations:

1. Primitive functions except *distl* and *distr*: As shown in Fig. 2c, the DFP program of $f : x$ is a composition of demand function f^d, object function $\tilde{x}$, and data function f. The demand object is first converted by f^d into a proper demand which requests the appropriate data object at the object function. According to the demand, the object function produces a data object for data function f to produce the result. When f is an arithmetic or logic function, f^d can be replaced by *id*, and *id* is removable in the function composition. Therefore, the input demand object can be directly forwarded to the object function.

2. *distr* and *distl*: The transformation for the broadcast functions can be observed in Fig. 2d. An example will explain the reason why a mask function is needed:

   Since $distl :< 1, < 2,3 >>=<< 1,2 >, < 1,3 >>$, $\rho(distl :< 1, < 2,3 >>) :< \delta, \varepsilon >$ should produce a $<< 1,2 >, \varepsilon >$. However, $\rho(distl :< 1, < 2,3 >>) :< \delta, \varepsilon >$
   $= distl \circ \tilde{x} \circ distl^d :< \delta, \varepsilon >$
   $= distl \circ \tilde{x} :< \delta, < \delta, \varepsilon >>$
   $= distl :< 1, < 2, \varepsilon >>$
   $= << 1,2 >, < 1, \varepsilon >>$
   Which is incorrect since the second '1' is not requested and should not be produced.

   The mask function in Fig. 2d is used to remove the extra elements which are not requested. The triangle denotes the combining of the data object from $f \circ \tilde{x} \circ f^d$ and the input demand, which is implicitly defined by the construction form. The result of the above example is corrected by the mask function: $mask :<<< 1,2 >, < 1, \varepsilon >>, < \delta, \varepsilon >>=<< 1,2 >, \varepsilon >$

3. *Composition form*, $f = f_1 \circ f_2$: From the definition of the transformation operator ρ,
   $$\rho(f_1 \circ f_2 : x) \equiv \rho(f_1 : (f_2 : x)) \equiv (f_1 \circ \rho(f_2 : x))^* \equiv (f_1 \circ (f_2 \circ \tilde{x})^*)^*$$
   In Fig. 3, objects $x, f_2 : x$, and $f_1 \circ f_2 : x$ are transformed into corresponding object functions. Object function $\tilde{x}$ is activated by a demand from block f_2. Similarly, object function $(f_2 \circ \tilde{x})^*$ is activated by a demand from block f_1. Object function $(f_1 \circ (f_2 \circ \tilde{x})^*)^*$ is activated by an input demand to this form.

4. *Construction form*, $f = |f_1, f_2|$: In Fig. 4a, input data object x is copied for both functions f_1 and f_2 and the two computed function values are combined to form a single output object. In Fig. 4b, the input demand to the form will request a two-element data object. Select functions 1 and 2 are used to obtain the component demands of the input demand. The component demands to the object functions of $f_1 : x$ and $f_2 : x$ will result in the production of required function values.

5. *Conditional form*, $f = f_1 \rightarrow f_2; f_3$: In Fig. 5a, the boolean value of predicate function f_1 determines either branch function f_2 or f_3 to execute. This branch function value will then become the output of the conditional form. Accordingly, in Fig. 5b, the input demand should first requests the boolean value of the object function of $f_1 : x$. The boolean value then determines to which branch object function the input demand should be sent.

6. *Apply-to-all form*, $f = \alpha f'$: The apply-to-all form can be recursively redefined as: $\alpha f' \equiv apndl \circ |f' \circ 1, \alpha f' \circ tl|$. The transformation simply make use of the transformation rules for primitive functions, composition form, and construction form.

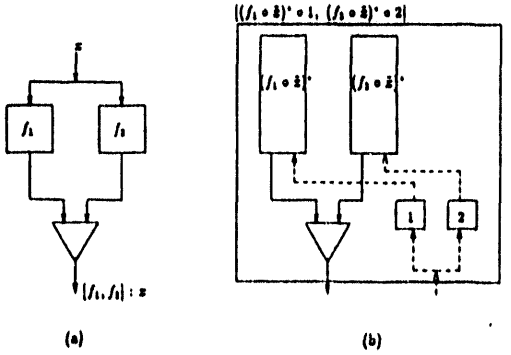

Figure 3: Transformation for the composition form.

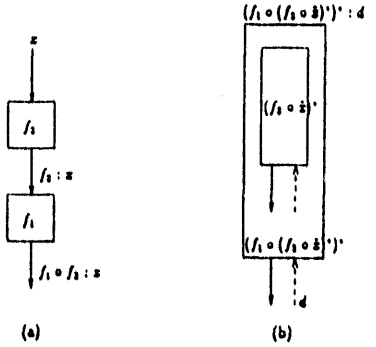

Figure 4: Transformation for the construction form.

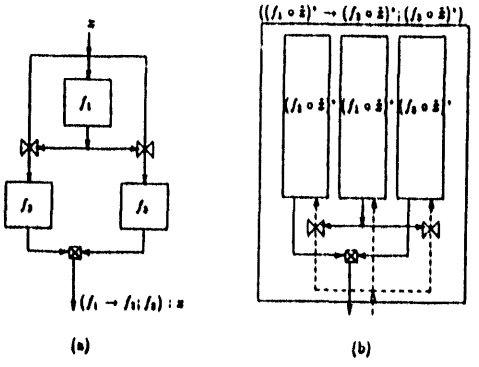

Figure 5: Transformation for the conditional form.

7. *Insert form*, $f = /f'$: Similarly, the insert form can also be recursively redefined as: $/f' \equiv f' \circ [1, /f' \circ tl]$. The transformation makes use of the other transformation rules.

8. *Constant form*, $f = \bar{y}$: The demand object replaces the data object x as a trigger to $\bar{y}$. The data object which is not ϵ will cause the production of the constant object y.

According to the transformation rules, the data-driven execution of any DFP function application will always evaluates to a data object with a *necessary and sufficient* information in respond to the input demand. In other words, not only all information requested is produced but also no information which is not requested will be produced. This evaluation is said to be least evaluation and the result produced is a least solution. This least evaluation property holds at every point of computation of DFP programs (Wei and Gaudiot 1988). If the does not hold at every point of computation, extra computation may be performed somewhere in the program and the execution may not be terminated when the extra computation is unbound.

3.4 DEMAND REDUCTION

DFP programs contain demand functions for runtime demand propagations. The purpose of demand propagations is to determine the execution paths and the appropriate execution order that lead to a least solution. Since it does not directly contribute to the production of the results, demand propagation is considered as an execution overhead. The demand propagations in many cases, for example in a network of arithmetic and logic functions, are irrelevant to the program dynamic behaviors and therefore will always result in a same request pattern. A demand reduction process is to remove these demand propagations at compile time to reduce the execution overhead.

A few useful rewrite rules for removing 'obvious' unnecessary demand functions from DPF programs are listed below.

(R1). For $f^d \in P_d - \{n^d, tl^d\}$, $f^d \circ cl \rightarrow cl$
(R2). For $f^d \in P_d$, $cl \circ f^d \rightarrow cl$
(R3). $n \circ cl \rightarrow cl$
(R4). f is an arithmetic, logic, or predicate function. $f^d \rightarrow id$
(R5). $cl \circ cl \rightarrow cl$
(R6). $\tilde{x} \circ id \rightarrow \tilde{x}$, $id \circ \tilde{x} \rightarrow \tilde{x}$, $(f \circ \tilde{x})^* \circ id \rightarrow (f \circ \tilde{x})^*, id \circ (f \circ \tilde{x})^* \rightarrow (f \circ \tilde{x})^*$
(R7). $f^d \circ id \rightarrow f^d$
(R8). $id \circ f^d \rightarrow f^d$
(R9). $(\tilde{x})^* \rightarrow \tilde{x}$
(R10). f is an arithmetic/logic function. $f \circ [\tilde{x} \circ 1, \tilde{x} \circ 2] \rightarrow f \circ [id, id] \circ \tilde{x}$

4 EXAMPLES OF TRANSFORMATION

The transformation of an infinite sequence generator (ISG) written in FP into its corresponding DFP program is presented. The FP ISG is not terminated in data-driven execution, The data-driven execution of the DPF ISG program is shown.

4.1 Square

FP SQuare can be defined by: $SQ \equiv \times \circ [id, id]$. The DFP SQ will be:

$$
\begin{aligned}
\rho(SQ : x) &\equiv (SQ \circ \tilde{x})^* && \text{(TR)}\\
&\equiv ((\times \circ [id, id]) \circ \tilde{x})^* && \text{(SQ)}\\
&\equiv (\times \circ ([id, id] \circ \tilde{x})^*)^* && \text{(TR)}\\
&\equiv \times \circ ([id, id] \circ \tilde{x})^* \circ \times^d && \text{(TR)}\\
&\equiv \times \circ ([id, id] \circ \tilde{x})^* && \text{(TR)}\\
&\equiv \times \circ [(id \circ \tilde{x})^* \circ 1, (id \circ \tilde{x})^* \circ 2] && \text{(TR)}\\
&\equiv \times \circ [\tilde{x} \circ 1, \tilde{x} \circ 2] && \text{(R4,R6)}\\
&\equiv \times \circ [id, id] \circ \tilde{x} && \text{(R10)}\\
&\equiv SQ \circ \tilde{x} && \text{(SQ)}
\end{aligned}
$$

where (TR), (SQ), and (R1) mean that the relations are according to transformation rules, the definition of SQ, and the rewrite rule R1 respectively. DFP SQ program contains no demand function.

4.2 Infinite Sequence Generator (ISG)

The infinite sequence generator (ISG) of the SQ of integers can be defined in FP by: $ISG = apndl \circ [SQ, ISG \circ add1]$. DFP ISG becomes:

$$
\begin{aligned}
&\rho(ISG : x) \equiv (ISG \circ \tilde{x})^* && \text{(TR)}\\
&\equiv ((apndl \circ [SQ, ISG \circ add1] \circ \tilde{x})^* && \text{(ISG)}\\
&\equiv (apndl \circ ([SQ, ISG \circ add1] \circ \tilde{x})^*)^* && \text{(TR)}\\
&\equiv apndl \circ ([SQ, ISG \circ add1] \circ \tilde{x})^*) \circ apndl^d && \text{(TR)}\\
&\equiv apndl \circ [(SQ \circ \tilde{x})^* \circ 1, ((ISG \circ add1) \circ \tilde{x})^* \circ 2] \circ apndl^d && \text{(TR)}\\
&\equiv apndl \circ [SQ \circ \tilde{x} \circ 1, ((ISG \circ add1) \circ \tilde{x})^* \circ 2] \circ apndl^d && \text{(SQ)}\\
&\equiv apndl \circ [SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x})^*)^* \circ 2] \circ apndl^d && \text{(TR)}\\
&\equiv apndl \circ [SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x} \circ add1^d))^* \circ 2] \circ apndl^d && \text{(TR)}\\
&\equiv apndl \circ [SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x}))^* \circ 2] \circ apndl^d && \text{(R4,R6)}
\end{aligned}
$$

4.3 Execution of DFP ISG

The data-driven execution of DFP ISG is shown. If $x = 3$ (therefore, $\tilde{x} : \delta = 3$), the sequence $ISG : 3$ will be $< 9, 16, 25, 36, \ldots >$. Let $d =< \epsilon, \delta, \epsilon^+ >$ to request the second element (which is 16) of the sequence.

$$
\begin{aligned}
&(ISG \circ \tilde{x})^* :< \epsilon, \delta, \epsilon^+ >\\
&= apndl \circ [SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x}))^* \circ 2] \circ apndl^d :< \epsilon, \delta, \epsilon^+ >\\
&= apndl \circ [SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x}))^* \circ 2] :< \epsilon, < \delta, \epsilon^+ >>\\
&= apndl :< SQ \circ \tilde{x} : \epsilon, (ISG \circ (add1 \circ \tilde{x}))^* :< \delta, \epsilon^+ >>\\
&= apndl :< \epsilon, apndl \circ [SQ \circ (add1 \circ \tilde{x}) \circ 1,\\
&\quad (ISG \circ (add1 \circ (add1 \circ \tilde{x})))^* \circ 2] \circ apndl^d :< \delta, \epsilon^+ >>\\
&= apndl :< \epsilon, apndl \circ [SQ \circ add1 \circ \tilde{x} \circ 1,\\
&\quad (ISG \circ (add1 \circ add1 \circ \tilde{x}))^* \circ 2] :< \delta, < \epsilon^+ >>>\\
&= apndl :< \epsilon, apndl :< SQ \circ add1 \circ \tilde{x} : \delta,\\
&\quad (ISG \circ (add1 \circ add1 \circ \tilde{x}))^* : \epsilon >> && \text{(note1)}\\
&= apndl :< \epsilon, apndl :< SQ \circ add1 : 3, \epsilon >>\\
&= apndl :< \epsilon, apndl :< SQ : 4, \epsilon >>\\
&= apndl :< \epsilon, apndl :< 16, \epsilon >>\\
&= apndl :< \epsilon, < 16, \epsilon^+ >> && \text{(note2)}\\
&= < \epsilon, 16, \epsilon^+ >
\end{aligned}
$$

Notes: (1) ϵ may represent $< \epsilon^+ >$; (2) According to the definition of function *apndl*, the second ϵ of the argument object of *apndl* is a sequence object. Since the

length of this sequence is not known, the result sequence will contain the first element followed by an uncertain number of unevaluated elements.

The in-line expansions of ISG is implemented by a higher order function call which will be explained in the next section. According to $f : \varepsilon = \varepsilon$, without performing an actual invocation, function application $(ISG \circ (add1 \circ add1 \circ \tilde{x}))^* : \varepsilon$ is directly replaced by an ε. In this example, only the second element of the infinite sequence is computed and its value is returned. The computation for the first element and the elements following the second element are not performed.

5 DFP DATA-FLOW GRAPH

Data-flow graph schemas for DFP programs are presented here. The construction of the DFP ISG data-flow graph is given.

5.1 Higher Order Function Application

A higher order function application may either take a function as its input argument or produce a function as its output. From the composition form, every DFP function $\rho(f : x)$ contains a function call to the parent object function $\tilde{x}$ to request a data object as its argument. The parent function would be dynamically determined when FP function f involves recursive definition as shown in ISG example. In order to handle this condition, the parent object function has to be parameterized.

The most general function application (the *Apply* actor in Fig. 6) in DFP consists of applying a DFP functional form Cf (for f) to a parent object function f' and a demand object d. $Cf, Cf \equiv (f \circ df?)^*$, is a DFP function with an unspecified parent object function. $df?$ is the parameter to be bound to parent object function argument f'. The data-flow graphs of DFP programs may contain:

1. *Apply* actor with all three input arcs: Functional form $(f \circ df?)^*$ at Cf uses f' as a parent object function to create an object function $(f \circ f')^*$. This object function then takes demand object d and produces a data object as its output.
2. *Apply* with only Cf and d arcs: An ordinary function application is performed.
3. *Apply* with only Cf and f' arcs: Functional form $(f \circ df?)^*$ at Cf uses f' as a parent object function to create an object function $(f \circ f')^*$ as its output.

When $d = \varepsilon$, there will be no actual function application actually activated at the *Apply*. Instead, the *Apply* actor simply produces an ε as output. With this definition, the removal of unnecessary computation is implemented.

5.2 Graph Schemas

According to the transformation rules, FP primitive functions and functional forms correspond to certain DFP program structures. For generality, $df?$ instead of $\tilde{x}$ is used in the graphs to denote that the parameter may be bounded to *any* DFP object functions in addition to object functions for FP objects.

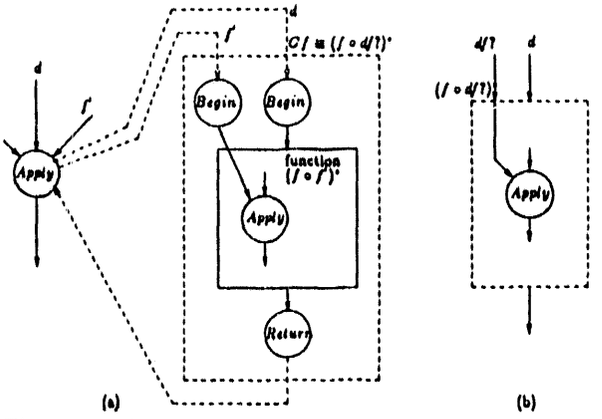

Figure 6: DFP graph schema for function applications.

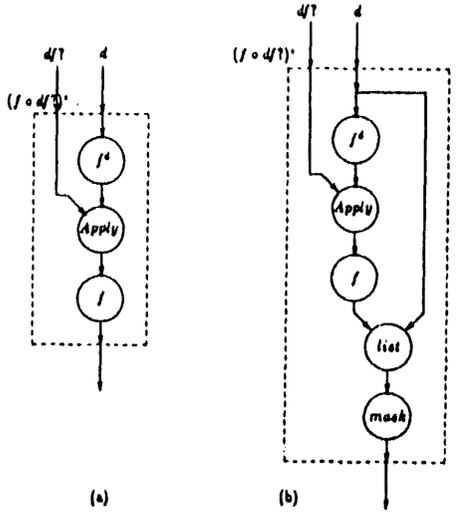

Figure 7: DFP graph schemas for:(a) Primitive functions which are not *distl* or *distr*; and (b) *distl* or *distr*.

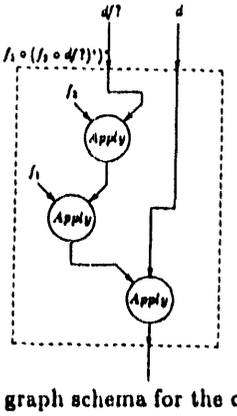

Figure 8: DFP graph schema for the composition form.

1. *f is a primitive FP function:* If $f \notin \{distl, distr\}$, the graph for $(f \circ df?)^* \equiv f \circ df? \circ f^d$ is shown in Fig. 7a. If $f \in \{distl, distr\}$, the graph for $(f \circ df?)^* \equiv mask \circ [f \circ df? \circ f^d, id]$ is shown in Fig. 7b. The *list* actor is to create an output object $< a, b >$ from two objects a and b.

2. *f is a composition Form:* $f \equiv f_1 \circ f_2$. The graph for $(f \circ df?)^* \equiv (f_1 \circ (f_2 \circ df?)^*)^*$ is shown in Fig. 8. The first *Apply* actor creates object function $(f_2 \circ df?)^*$ and sends it to the second *Apply* actor. The second *Apply* actor then creates object function $(f_1 \circ (f_2 \circ df?)^*)^*$. Ths object function takes demand object d and produces an output data object.

3. *f is a construction Form:* $f \equiv [f_1, f_2]$. The graph for $(f \circ df?)^* \equiv [(f_1 \circ df?)^* \circ 1, (f_2 \circ df?)^* \circ 2]$ is shown in Fig. 9a. The first and second elements of the demand object d are retrieved by selectors 1 and 2 to request respectively the function values of f_1 and f_2. Two branch objects are then combined into a single output object at the actor *list*.

4. *f is a conditional Form:* $f \equiv f_1 \rightarrow f_2; f_3$. The graph for $(f \circ df?)^* \equiv (f_1 \circ df?)^* \rightarrow (f_2 \circ df?)^*; (f_3 \circ df?)^*$ is shown in Fig. 9b. The boolean object evaluated by $(f_1 \circ df?)^* : d$ determines the branch object function to which the demand object d should be sent. Accordingly, the branch function will be invoked to produce output.

5. *f is a constant Form:* A *constant* actor is used. Any demand object token except *VEPS* to this actor will trigger the production of a predefined constant token. If the demand is an ϵ, an ϵ will be produced.

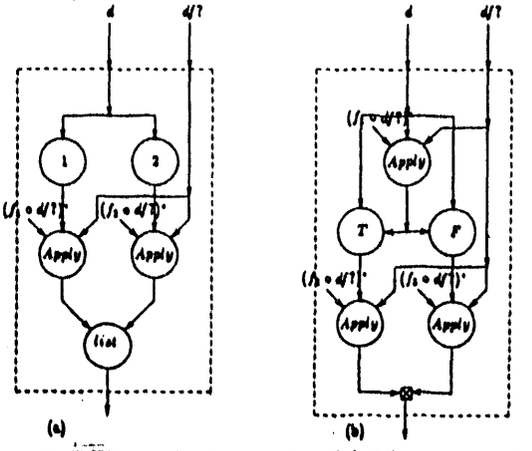

Figure 9: DFP graph schemas for: (a) The construction form; and (b) The conditional form.

5.3 DFP ISG Graph

DFP ISG data-flow graph is obtained by applying the above graph schemas to the DFP ISG program: $\rho(ISG : x) \equiv apndl \circ [SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x}))^* \circ 2] \circ apndl^d$

Figure 10a is the graph for $(ISG \circ (add1 \circ \tilde{x}))^*$ where function form $(add1 \circ df?)$ (detailed in Fig. 10b) takes $\tilde{x}$ to produce object function $(add1 \circ \tilde{x})$. The object function is taken by function form $(ISG \circ df?)^*$ to produce object function $(ISG \circ (add1 \circ \tilde{x}))^*$. This object function replaces the $(f_2 \circ \tilde{x})^*$ of DFP construction-form graph construct in Fig. 9a. The DFP ISG data-flow graph is obtained by the compositions of $apndl^d$, $[SQ \circ \tilde{x} \circ 1, (ISG \circ (add1 \circ \tilde{x}))^* \circ 2]$, and $apndl$ as shown in Fig. 10c.

6 CONCLUSIONS

The lazy evaluation of complex data structure functional programs in data-driven environment is presented. The basic FP system is used as a source language in the study. A lazy version of FP, DFP (Demand-driven FP) is first defined by including the concepts of partial data/demand objects. FP-DFP transformation is used to convert FP eager programs into DFP lazy programs with demand propagations. With DFP graph schemas, DFP lazy data-flow graphs can be generated from DFP lazy programs. The data-driven execution of these DFP data-flow graphs has the same effects of lazy evaluation. The methodologies presented are applicable to the development of lazy evaluation systems for other functional languages.

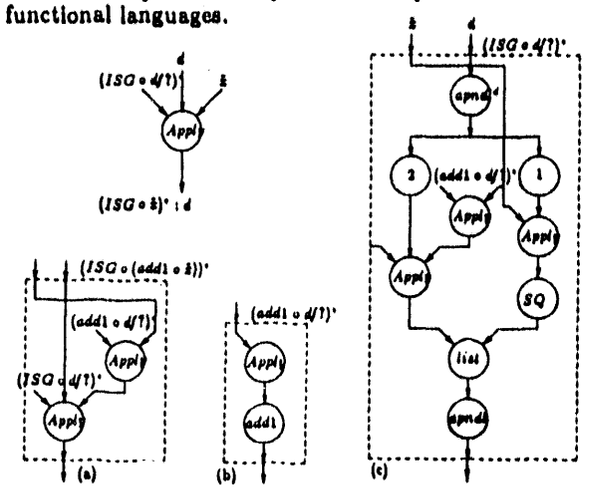

Figure 10: DFP ISG data-flow graph.

REFERENCE

[AH84] M. Amamiya and R. Hasegawa. Dataflow computing and eager and lazy evaluations. *New Generation Computing*, pp. 105-129, 1984.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, pp. 613-641, August 1978.

[CP88] C. Clack and S.L. Peyton Jones. Strictness analysis - a practical approach. *In Springer-Verlag:LNCS*, pp. 35-49, 1988.

[FW76] D.P. Friedman and D.S. Wise. Cons should not evaluate its arguments. *Automata, Languages and Programming*, pp. 257-284, 1976.

[HW85] J.Y. Halpern, J.H. Williams, E.L. Wimmers, and T.C. Winkler. Denotational semantics and rewrite rules for FP. Proc. of the 12th ACM Conf. on Principles of Prog. Lang, pp. 108-120, 1985.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. *In ACM SIGPLAN Notices*, pp. 58-69, June 1984.

[KS84] J.R. Kennaway and M.R. Sleep. The 'language first' approach. *Distributed Computing*, pp. 111-124, 1984.

[PA85] K. Pingali and Arvind. Efficient demand-driven evaluation. part 1. *ACM Transactions on Programming Language and Systems*, pp. 311-333, April 1985.

[Vee86] A.H. Veen. Dataflow machine architecture. *ACM Computing Survey*, 18(4):pp. 365-396, December 1986.

[Veg84] S.R. Vegdahl. A survey of proposed architecture for the execution of functional languages. *IEEE Transactions on Computers*, c-33(12):pp. 1050-1071, December 1984.

[WG88] Y.H. Wei and J.L. Gaudiot. Demand driven interpretation of fp programs on a data-flow multiprocessor. *IEEE Transactions on Computers*, August 1988.

END

DATE
FILMED
11/16/93