Formal Aspects of Computing (2001) 3: 1–000

Stability, Sequentiality and Demand Driven Evaluation in Dataflow

Arnon Avron and Nada Sasson

Tel Aviv University, Israel

Keywords: Dataflow; Stability; Sequentiality.

Abstract. We show that a given dataflow language l has the property that for any program P and any demand for outputs D (which can be satisfied) there exists a least partial computation of P which satisfies D, iff all the operators of l are stable. This minimal computation is the demand-driven evaluation of P. We also argue that in order to actually implement this mode of evaluation, the operators of l should be further restricted to be effectively sequential ones.

1. Introduction

Dataflow machines are a method of implementing functional languages, which is especially suited to exploit parallelism in programs (see [Iee82]). For this they usually incorporate streams, which are infinite data objects. On the other hand, they are designed for *data-driven evaluation* of programs: operations on elements of streams can execute as soon as their inputs are available. It was observed (e.g. in [PiA85]) that the combination of these two properties might generate an infinite amount of useless computations. Hence a *demand-driven* (or "lazy") paradigm is preferable.

The papers [PiA85, Pin86] are devoted to a thorough study of demand-driven evaluation in dataflow computations. Still, Arvind and Pingali (A&P) have left several theoretical gaps in their work:

- The central notion of a demand-driven evaluation of a program P has never been defined in complete generality. More precisely, A&P defined it to be

Correspondence and offprint requests to: Arnon Avron and Nada Sasson, Raymond and Beverly Sackler Faculty of Exact Sciences, Computer Science Department, Tel Aviv University, Ramat-Aviv 69978 Israel. Email: aa@taurus.bitnet.

the minimal element (if such exists) in the set of "legal valuations for P" which satisfy the demand. A&P gave, however, no general definition of a "legal valuation for P" (which intuitively is a description of a state which can be reached during a data-driven evaluation of P). Instead they gave two conditions that any such valuation should obviously satisfy, but they gave themselves an example in which these conditions are intuitively not *sufficient* for "legality". A&P restricted themselves then to the case in which all the operations involved in a program are sequential. Obviously they were assuming that in this special case their two conditions *are* sufficient. No attempt to justify this belief was made, though.

- For the sequential case A&P were able to show (granting the above assumption) that a minimal "legal" valuation meeting a given demand always exists (unless the demand is unsatisfiable). Here however, it is not clear whether the condition of sequentiality is necessary, and A&P made no attempt to determine the exact range of programs for which one can meaningfully talk about demand-driven evaluation.
- From a practical point of view, A&P suggested a method of implementing the demand-driven evaluation of a program within a certain restricted class of operations. It is clear that the method is applicable to a much larger class, but no attempt to determine its range of applicability was made.

Our goal in this paper is to fill in the above gaps and to put the subject in a more general framework. Our main result is that from an abstract point of view, the largest class of operations which allows demand-driven evaluation in dataflow is that of *stable* operations (which contain the sequential ones). It is rather interesting that the class of stable functions which was originally introduced by G. Berry (see [Ber76, Ber78a, Ber78b, Ber79]), has an important role in our work. What we actually show is that the following two properties are equivalent for a given set S of operations on streams of data tokens from flat domains:

1. Given any program P which is based on S and any demand D which P can satisfy, there exists a least legal valuation for P which satisfies D.
2. All the operations in S are stable.

The notions of a "legal valuation" and a "demand" in the above characterization are defined in rather general terms, with no extra assumptions about the operations in S (except that they are continuous). We show, however, that for *stable* operations the two conditions of A&P are indeed sufficient and necessary for legality.

When it comes to practical implementation of demand-driven evaluation, the set of stable operations is apparently too large. We illustrate this by an example. We argue then that for this goal even the set of sequential operations [1] is too large, and that one should limit oneself to the subset of *effectively sequential* operations. We outline the obvious method of doing the implementation in such a case (which is essentialy A&P's method, put in a general form), and conclude that at least for one especially important type of demands (in fact, the only ones disscussed in [PiA85]), the effective sequentiality condition is indeed sufficient for the applicability of the method.

[1] These are the only operations studied by A&P.

The structure of the rest of the paper is as follows. In section 2 we present the general syntax of basic stream languages and describe denotational and operational semantics for data-driven evaluations of their programs. In section 3 we present a denotational semantics for *demand-driven* evaluations of such programs and show that stability of the language's operators is a necessary and sufficient condition for its applicability. Finally, in Section 4 we describe a corresponding operational semantics and explain why the stronger condition of effective sequentiality is needed to make it work.

Two important remarks: First, although the specific target of the present paper is to improve theoretical results of Arvind and Pingali ([PiA85, Pin86], the topic of demand-driven dataflow should be framed in the wider context of interpreting applicative languages ([Tur79]). The idea has indeed been tested by Richmond's implementation ([Ric82]) of Turner's SASL language, running on the Manchester dataflow hardware. Second, despite the fact that the paper heavily draws on these works of A&P, we have made a great effort to make it self-contained, and so no acquaintance with A&P's work is needed in order to read it.

2. Basic Stream Languages

In this section, we will present the syntax and semantics of basic stream languages for stream processing. We will call these languages ST languages.

2.1. Syntax of an ST Language

2.1.1. Alphabet and Types

There are four sets of symbols:

- T – The set of *simple* Types.
- C – The set of Constants.
- VR – The set of Variables.
- F – The set of Function Symbols.

If $\sigma_1, \ldots, \sigma_{n+1}$ are elements of T then $\sigma_1\_stream * \ldots * \sigma_n\_stream \rightarrow \sigma_{n+1}\_stream$ is a *complex* type (this includes the case $n = 0$). Together the simple and the complex types form the category of *types*. Any other construct of the language has a unique type. In particular, each element of C and VR has a complex type of the form σ_stream for some $\sigma \epsilon T$, while each $f \epsilon F$ has a complex type of the form $\sigma_1\_stream * \ldots * \sigma_n\_stream \rightarrow \sigma_{n+1}\_stream$ where $n > 0$ and $\sigma_1, \ldots, \sigma_{n+1} \epsilon T$.

Note: In the intended semantics elements of simple types are what is called in [PiA85] *scalar values*. Note that we do not include in the language constants for such values. On the other hand we put no restriction on the form of the constants in C. Thus a typical constant of type $integer\_stream$ (where $integer \epsilon T$) which we use below is $[1, 2, 3, \ldots]$. This form reflects the obvious intended meaning, but its "components" (like "1") are *not* part of the language. Of course, in a concrete implementation such constants of C will be represented by finite terms of some auxiliary language, which most probably *will* contain (among other things) constants for scalar values.

2.1.2. Terms and their Types

The set of terms over (T,C,VR,F) is divided into *atoms* and *applications*. Each term is associated with a type.

- **Atoms:** constants and variables.
- **Applications:**
  If $f \epsilon F$ is of type $\sigma_1\_stream * \ldots * \sigma_n\_stream \rightarrow \sigma_{n+1}\_stream$ and $X_1, \ldots, X_n$ are *atoms* of types $\sigma_1\_stream, \ldots, \sigma_n\_stream$ respectively, then $f(X_1, \ldots, X_n)$ is a term of type σ_{n+1}_stream.

2.1.3. The Syntax of Programs in ST Languages over (T,C,VR,F)

A program in an ST language consists of two parts: *Graph*, a set of $n \geq 1$ equations of the form $X_i = t_i$, and *Input*, a set of $m \geq 0$ equations of the form $I_j = c_j$, such that:

- $X_1, \ldots, X_n, I_1, \ldots, I_m$ are $n+m$ different variables from VR.
- $c_1, \ldots, c_m \ \epsilon C$.
- The t_i's $(i = 1, \ldots, n)$ are applications of the form $f_i(Y_{i_1}, \ldots, Y_{i_{k(i)}})$ where $Y_{i_j} \epsilon \{X_1, \ldots, X_n, I_1, \ldots, I_m\}$ for $j = 1, \ldots, k(i)$.
- The types in an equation should match.
- Every variable occurs exactly once on the left hand side of one of the equations.

$I_1, \ldots, I_m$ are called *Input Variables*. All the other variables are *internal Variables*. Some of them are determined in advance as *Output Variables* (either by declaring them as such or by using some convention of the kind we use below).[2]

We shall now introduce conventions regarding the names of the variables of a program in an ST language:

- The input variables and only them will begin with the letters I or i.
- The output variables and only them will begin with the letters O or o.

2.1.4. Representing a Program in an ST Language as a Directed Graph

An alternative way to represent a program in an ST language, which is extensively used in [PiA85, Pin86], is as a directed graph (see example 2.1 below). There is an obvious correspondence between the two representations: each node of the directed graph represents one of the applications appearing in the *Graph* part of the textual program. The incoming arcs of the node represent the inputs of the application, and the outgoing arc represents the output of the application.

[2] Note that output variables *may* occur on the r.h.s. of an equation, and so they cannot be distinguished from other variables by just inspecting the program.

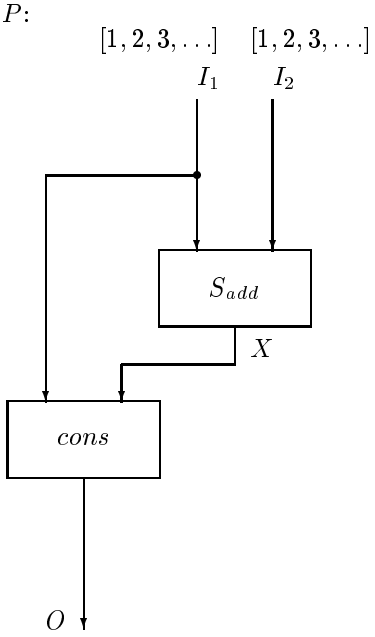
Example 2.1.

Program P:

$$\begin{aligned} Graph: X &= S_{add}(I_1, I_2) \\ O &= cons(I_1, X) \\ Input: I_1 &= [1,2,3,\ldots] \\ I_2 &= [1,2,3,\ldots] \end{aligned}$$

Remark: As we have already explained above, the constant $[1,2,3,\ldots]$ is considered to be a single data object. Later on an obvious semantics will be given to it.

The graph representation of P:



2.2. Denotational Semantics

We assume that the reader is acquainted with the *basic theory of Complete Partial Order Sets* (see [LoS84, MiS76, ScB69, Sco70, Sto77]).

In order to give denotational semantics for ST languages, we first need some definitions: Let D be a *flat domain*. (I.e. D is a set which is equipped with a partial order $\subseteq$, so that it contains a least element $\perp$, and all its other elements are incomparable). Denote by Ω_D the set of all infinite sequences of elements from D, and by Φ_D the subset of Ω_D of *proper sequences*. (A sequence is called *proper* if whenever one of its elements is equal to $\perp$, then so are all its successors).

Define, for $X_1, X_2 \epsilon \Omega_D$, $X_1 \subseteq X_2$ iff $\forall i > 0$, $X_1[i] \subseteq X_2[i]$ in D. ($X[i]$ denotes the i'th element of X). This makes both Φ_D and Ω_D *cpos* with a least element

[ ] (a sequence in which every element is $\perp$). [3]

Notation: When writing down a sequence from Φ_D or Ω_D which has $\perp$ from one place onwards, we will omit its rightmost $\perp$ elements (as we just have done in the case of [ ]).

We follow [Pin86] in presenting two types of denotational semantics for ST languages: semantics of type L and semantics of type L-tagged.

Definition 2.1.

Let l be an ST language. A function IN is called an *interpretation* for l of type L / L-tagged if:

1. for $\sigma \epsilon T$, $IN(\sigma) = D^\sigma$ where D^σ is a flat domain. [4]
2. for $\sigma \epsilon T$, $IN(\sigma\_stream) = E^\sigma$, where:
   (a) For type L: $E^\sigma = \Phi_{D^\sigma}$.
   (b) For type L-tagged: $E^\sigma = \Omega_{D^\sigma}$.
3. $IN(c^\sigma) \epsilon E^\sigma$ in case $\sigma \epsilon T$ and $c^\sigma \epsilon C$ is a constant of type σ_stream.
4. For $\sigma_1, \ldots, \sigma_{n+1} \epsilon T$, $IN(\sigma_1\_stream * \ldots * \sigma_n\_stream \rightarrow \sigma_{n+1}\_stream)$ is the domain of continuous functions from $E^{\sigma_1} * \ldots * E^{\sigma_n}$ to $E^{\sigma_{n+1}}$.
5. For $f \epsilon F$, $IN(f) \epsilon IN(\alpha)$ where α is the type of f.

Definition 2.2.

Let l and IN be as above. Let P be a program in l. Define:

$$\mathcal{V} = \bigcup_{\sigma \epsilon T} E^\sigma$$

- A *valuation* v for P (relative to IN) is a type-respecting assignment of an element of $\mathcal{V}$ for each of the variables of P.

  Notation: We denote by vX the element of $\mathcal{V}$ which v assigns to X.
- If v_1 and v_2 are valuations for P, then $v_1 \subseteq v_2$ iff for any variable X in P, $v_1 X \subseteq v_2 X$. (Notice that the valuations of P are elements of a *cpo* which is a product domain of *cpos*).

Definition 2.3.

Let l, IN and P be as above. Define a function τ_P from valuations to valuations (for P) as follows:

1. For any input variable I: $\tau_P(v)I = vI$.

[3] We prefer to follow [Pin86] in using these cpos rather than Kahn's *cpo* D^ω (see [Kah74]) which was adopted in [PiA85]. The reason is that it makes the expression $X[i]$ meaningful for every X and i, and so allows a more unified treatment of the two types of semantics given below.

[4] By convention, the flat domains $D^{integer}, D^{boolean}$ etc. will be given their usual interpretations. (e.g. : $D^{integer} = \{\perp, 1, 2, \ldots\}$).

2. For any internal variable X, which is the output of a function symbol f with inputs $Y, Z, \ldots$: $\tau_P(v)X = IN(f)(vY, vZ, \ldots)$.

Note: For convenience, we shall use from now on the same symbol for denoting a function symbol in the language and its associated function (unless there is a danger of confusion).

Given l, IN and P, it is easy to see that τ_P is a monotonic and continuous function from a product of sequence domains to the same product domain. Now P can be viewed as a set of equations over *cpos*. It is well known that such a set of equations has a least solution, which is the least upper bound (lub) of the chain $\{\tau_P^k(v_0)\}_{k=0}^{\infty}$ where:

- for any input variable I with the equation $I = c$ in P: $v_0 I = c$.
- for any internal variable X: $v_0 X = [\,]$.

For more information see [Sto77].

This least solution is called in [PiA85, Pin86] the *meaning of* P (relative to IN), and is denoted by H_P.

Example 2.2.

1. Standard interpretations of type L for the function symbols given in example 2.1:
   (a) $cons : \Phi_{D^{integer}} * \Phi_{D^{integer}} \to \Phi_{D^{integer}}$
   $cons([\,], [x_1, x_2, \ldots]) = [\,]$
   $cons([a_1, \ldots], [x_1, x_2, \ldots]) = [a_1, x_1, x_2, \ldots])(a_1 \neq \perp)$
   (b) $S_{add} : \Phi_{D^{integer}} * \Phi_{D^{integer}} \to \Phi_{D^{integer}}$
   $S_{add}(X, [\,]) = S_{add}([\,], Y) = [\,]$
   $S_{add}([x_1, x_2, \ldots], [y_1, y_2, \ldots]) = cons([x_1 + y_1], S_{add}([x_2, \ldots], [y_2, \ldots]))$
2. It is easy to see, that under the interpretations of part (1), for the program given in example 2.1:

$$\begin{array}{cccccc} & & I_1 & & I_2 & & X & & O \\ H_P = & ([1,2,3,\ldots], & [1,2,3,\ldots], & [2,4,6,\ldots], & [1,2,4,6,\ldots]) \end{array}$$

2.3. Computational Models (Operational Semantics) for ST Languages

We will only briefly present this subject. For detailed information refer to [ArG77]. We can refer to the graphical representation of a program in an ST language as a dataflow graph. Its arcs represent data lines with data streams flowing along them, and its nodes represent dataflow operators which process the data streams. The streams are implemented as sequences of data tokens, holding data values of simple types. The behaviour of the dataflow operators is determined by *firing rules* which describe how the operators consume their input tokens and how they produce output tokens.

There is a difference between the computational models of semantics of types L and L-tagged. For semantics of type L, a *queued / piped* dataflow model is used. In this case, the data lines of the graph represent unbounded FIFO buffers

in which the tokens are kept. For semantics of type L-tagged, the data streams can have "holes" in them, so in this case a *tagged* dataflow model is used [5]. In addition to its data value, each token has a tag which identifies its position number on the data line. This is necessary because a token at position i can be produced on a data line even if a token at position $j < i$ had never been produced. The firing rules of the dataflow operators relate to the tags of the tokens as well as their data.

If a "fair" algorithm for scheduling operators for execution is used (i.e. every operator that can fire will eventually be scheduled), the behaviour of the graph is independent of the scheduling policy, and for any program (in any ST language with semantics of type L or L-tagged) the streams produced on the lines of the dataflow graph of the program and the values of the appropriate variables in the least solution of the program are identical. This fact is known as *the Kahn Principle* and is usually expressed by saying that the operational semantics is *congruent* with the denotational semantics. (Proofs of the Kahn Principle in similar setups are given in [Fau82, LiS89]).

3. Demand Driven Evaluation for ST Languages

3.1. Data Driven Evaluation Versus Demand Driven Evaluation

The valuation H_P represents a *data driven evaluation* for P. This comes from the fact that in the dataflow graph of P, operators can execute as soon as the data tokens needed for their operation are available on their input lines. The problem with this method of evaluation is that in many cases it is very wasteful. Frequently, programers are only interested in the outputs of the program or even in part of the outputs. However, in a data driven evaluation, any data token that can be produced on the lines of the program's graph will eventually be produced, whether or not its value is actually needed for producing the wanted outputs. Therefore, on the lines of the graph, a large or even an unbounded (in the case of infinite data streams) amount of unnecessary data tokens might be produced.

Example 3.1.

In the program given in example 2.1, an unbounded number of data tokens are computed on lines X and O, even if the programer is only interested in the values of the first N data tokens.

Since a data driven evaluation for programs is not sufficiently efficient, a need arises for a *demand driven* evaluation in which only data tokens which are *needed* for computing the demanded outputs of a program are produced on its dataflow graph.

[5] It should be noted that, although Ω_D provides an elegant basis for describing streams of *tagged*-tokens (in the style of the UI interpreter of [ArG77]), the semantics of actual tagged-dataflow systems are inherently more complex (cf., e.g., the Manchester machine [Oli84, Jon87]).

Example 3.2.

Intuitively, a demand driven evaluation for producing the first 3 data tokens on the output line of program P from example 2.1, is represented by the following valuation v:

$$v = \begin{array}{cccc} I_1 & I_2 & X & O \\ ([1,2], & [1,2], & [2,4], & [1,2,4]) \end{array}$$

It is easy to see that the computation represented by v is sufficient to produce the demanded outputs, and that any attempt to produce these outputs by performing less computations will fail.

3.2. Computations and Legal Valuations

The previous example raises the question, whether it is possible to give a precise definition of the concept "a demand driven evaluation" for any program P in any ST language and any demands for outputs of P. Intuitively, if such an evaluation does exist, it has to be a *minimal* computation performed on the dataflow graph of P, that can still satisfy the demand for outputs.

First we should define the notion of a "computation performed on the dataflow graph of P". Obviously, from an external point of view, a computation is represented by the data streams produced on the dataflow graph while performing that computation. In other words, we can represent a computation performed on the dataflow graph of P, as a valuation for P (as we did in example 3.2). However, it is clear that not every valuation for P represents such a computation. Intuitively, a valuation representing a computation for P must satisfy the following two conditions:

1. The streams it assigns to the input lines of P are partial to the actual input streams of P, as determined by the interpretation.
2. There are no "guesses" in the valuation: data items are produced only if this is dictated by P.

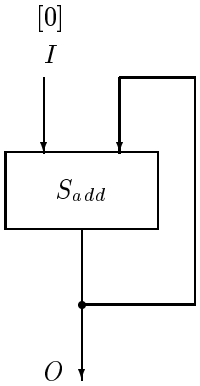
We will call a valuation for P, which intuitively satisfies these two conditions, *a legal valuation for* P and identify the concept of "computation performed on the dataflow graph of P" with the concept of "legal valuation for P". Our first task is therefore to provide a *formal* definition of the concept of legal valuations for programs.

Let v be a valuation for a program P. It is easy to formally express condition (1): for any input line I of P, with the equation $I = c$ appearing in the input part of P, the following condition should hold: $vI \subseteq c$.

It is more difficult to define the absence of guesses in v (condition (2)). On the face of it, it seems as if we should only check that for any operator f of P with inputs $X_1, \ldots, X_n$ and output Y, $vY \subseteq f(vX_1, \ldots, vX_n)$. This is undoubtfully a necessary condition for the legality of v, but the next example (taken from [Pin86]) shows that this condition is not sufficient.

Example 3.3.

Let us look at the following program P:
(The definition of the S_{add} operator is given in example 2.2).



No data tokens can be produced on line O and the least solution of P is $H_P = ([0], [\,])$. In the valuation $v = ([0], [1])$, the value 1 is clearly a guess but the condition $[1] = vO \subseteq S_{add}(vI, vO) = [1]$ still holds!

The conclusion from this example is that we cannot determine whether there are guesses in a valuation v or not, merely by checking the values of the lines of P in v. We must perform a more serious check. What we should do is to start with a state in which vI appears on each input line I of the dataflow graph of P, and on the other lines there are no data tokens at all. We should then check if we can produce on each line X the stream vX, without guessing the value of any of the data tokens.

Given a valuation v, let us define a valuation v_1 that represents the initial state: for any input line I, $v_1 I = vI$ and for any other line X, $v_1 X = [\,]$. Now let us execute each of the operators of the dataflow graph that can fire. As a result, on each line of the graph, a new stream of data tokens will appear. For each line X, let us *remove* from the new stream all the data tokens which do not appear in vX. We do this because v represents a computation in which those data tokens are not produced. As a result of our actions, we receive a new valuation – v_2.

Now, let us perform on v_2 the same actions we performed on v_1 and call the resulting new valuation v_3. We can go on performing the same actions on v_3 etc..

It is easy to see that $\{v_i\}_{i=1}^{\infty}$ is an increasing chain of valuations and that $\forall i,\ v_i \subseteq v$. This chain of valuations represents a series of steps in a computation that can be performed on the dataflow graph of P. The way we create the valuations ensures that only data tokens which can actually be produced by that computation appear in these valuations – so there are no guesses in any of them. The lub of the chain is the result of the computation, and intuitively it contains all the data tokens of v that can be created without guessing their values. (Each of them can be computed from the input streams and therefore will appear somewhere along the chain). Hence, there are no guesses in v iff it is equal to the lub of the chain. It is very reasonable, therefore, to formally define the concept of *a legal valuation* using this chain of valuations.

From now on, l will be some fixed ST language and IN some fixed interpretation for l of type L or L-tagged. All the definitions below are done relative to l and IN.

Definition 3.1.

Let P be an l-program and let v be a valuation for P.

1. The *projected chain of* P *and* v is the chain $v_1 \subseteq v_2 \subseteq \ldots \subseteq$ where:
   (a) For any input variable I: $v_1 I = vI$
   (b) For any internal variable X: $v_1 X = [\,]$
   (c) For $i > 0$: $v_{i+1} = glb\{\tau_P(v_i), v)\}$.
2. v is *a legal valuation for* P iff the following two conditions hold:
   (a) For any equation $I = c$ appearing in P, $vI \subseteq IN(c)$.
   (b) The *lub* of the projected chain of P and v is v.

Remark: it is easy to see that H_P is a legal valuation for P.

3.3. On Demands and Legal Valuations

We now give several other definitions which are needed in order to achieve our goal of defining the concept of *a demand driven evaluation* for a program in precise terms.

Definition 3.2.

Let P be an l-program.

1. *A demand D for outputs of* P is a function D: $\mathcal{O} \to 2^{\mathbf{N}}$ where $\mathcal{O}$ denotes the set of output variables of P.
2. Given a valuation v and a demand D, we say that token $vO[i]$ is *demanded* by D iff $i \epsilon D(O)$.
3. A valuation v for P is *output-complete* with respect to D iff $\forall O \epsilon \mathcal{O}$, $i \epsilon D(O) \Rightarrow vO[i] \neq \perp$.

Example 3.4.

1. $D(O) = \{i |\ H_P O[i] \neq \perp\}$. [6]
2. $D(O) = \{i |\ 1 \leq i \leq n\}$ [7].
3. Let $H_P O[i]$ be the first element of $H_P O$ whose value is $=, <, \leq, >$ or $\geq$ than some fixed value *val*. Take $D(O) = \{j |\ j < i \ \wedge \ H_P O[j] \neq \perp\}$.

[6] H_P is the only legal valuation which satisfies this demand. Satisfying it amounts therefore to data-driven evaluation.

[7] For languages of type L only demands of this particular form are considered in [PiA85, Pin86]. For languages of type L-tagged the possibility of "holes" is also allowed in [Pin86].

Definition 3.3.

Let P be an l-program and let D be a demand for outputs of P.

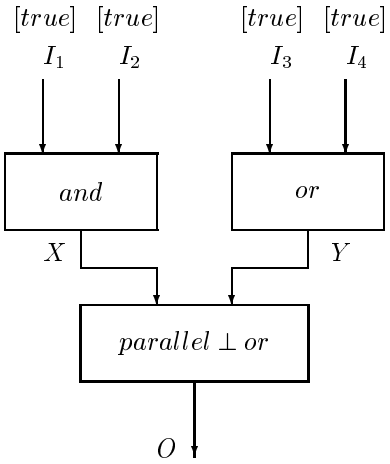
1. The set $S_{P,D}$ denotes the set of all the *legal* valuations for P, which are *output-complete* w.r.t to D.
2. The *D-driven evaluation of P* is the least element of $S_{P,D}$ (if such exists).

The set $S_{P,D}$ contains all the valuations which represent computations that can be produced on the dataflow graph of P and contain the demanded outputs. (Obviously, a demand driven computation should be minimal among them).

A crucial question now is whether for any IN, P and D, $S_{P,D}$ actually has a least element. A positive answer to this question would have meant that the concept of the *demand driven evaluation* is always meaningful. Unfortunately, the answer to this question is in general *negative*, as the next example from [Pin86] shows.

Example 3.5.

Let program P (represented as a directed graph) be:



and consider the following demand D for P: $D(O) = \{1\}$. Examine now the following two valuations for P:

	I_1	I_2	I_3	I_4	X	Y	O
$v_1 =$	([*true*],	[*true*],	[ ],	[ ],	[*true*],	[ ]	[*true*])
$v_2 =$	([ ],	[ ],	[*true*],	[*true*],	[ ]	[*true*],	[*true*])

The output of P can be produced either by computing the left input of the parallel-or (v_1), or by computing its right input (v_2). Since we can compute the required output by following two different pathes in the graph, we cannot compare the two computations. Indeed, it is easy to see that v_1 and v_2 are both legal valuations of P and output-complete w.r.t D, but the only legal valuation that is smaller than both of them (the valuation that matches the value [ ] to every line), is *not* output-complete w.r.t D. Hence, for this program P, $S_{P,D}$ does not contain a least element.

Note: The operators *and, or* and *parallel-or* perform the usual operations (with the same names) on pairs of data tokens with identical position numbers. For the reader convenience we include here the truth table of the parallel-or operation:

	$\perp$	$false$	$true$
$\perp$	$\perp$	$\perp$	$true$
$false$	$\perp$	$false$	$true$
$true$	$true$	$true$	$true$

In spite of the previous example, there are many cases in which $S_{P,D}$ does indeed contain a least element. Our next goal is to identify the set of interpretations of types L and L-tagged, for which for *any* program P and *any* demand D for outputs of P (or at least a "reasonable" class of such demands) the set $S_{P,D}$ is either empty or contains a least element.

3.4. Stable Functions and Demand Driven Computations

The failure in example 3.5 is obviously due to the use of a truely parallel function (parallel-or). It seems reasonable, therefore, to conjecture that if we employ only *sequential* functions (see [Cur85, KaP, Mil77, Pin86, Vui73]) then such a phenomenon will not happen.

Formally, a function $f : \Omega_{D^{\sigma_1}} * \ldots * \Omega_{D^{\sigma_n}} \rightarrow \Omega_{D^{\sigma_{n+1}}}$ is *sequential* iff for every $X = (X_1, \ldots, X_n)$ and every k such that $f(X)[k] = \perp$, there exist some $1 \leq i \leq n$ and j such that $X_i[j] = \perp$ and

$$\forall Y = (Y_1, \ldots Y_n) \supseteq X, f(Y)[k] \neq \perp \Rightarrow Y_i[j] \neq \perp \ ^{8}$$

(The intuition behind this definition is that a function f from streams to streams is sequential at X if for each empty position k at $f(X)$ we can identify some unfilled position in the input X which is "critical" – i.e. unless this position in the input is filled, position k in the output cannot be filled.)

Basically, the above conjecture concerning sequential functions is proved in [Pin86] [9]. It turns out, however, that the condition of sequentiality is only *sufficient* for the existence of a least element in $S_{P,D}$. A larger class of functions will in fact do: that of *stable* functions ([Ber76, Ber78a, Ber78b, Ber79]).

Definition 3.4.

Let D_1 and D_2 be *cpos* and let $f : D_1 \rightarrow D_2$ be a continuous function. We call f *stable* iff for every $x \epsilon D_1$ and $y \epsilon D_2$ where $y \subseteq f(x)$, there exists $M(f, x, y) \epsilon D_1$ such that $\forall z \subseteq x,\ \ y \subseteq f(z) \ \Leftrightarrow M(f, x, y) \subseteq z$.

It is easy to show that sequentiality implies stability in the cpos we consider here. The converse is false, however (see references above).

---

[8] For Φ_{D^σ} the definiton is similar except that k and j must be, respectively, the first unfilled positions in $f(X)$ and X_i.

[9] Except that no general definition of a legal valuation, which does not a priori assume that only sequential functions are used, is given.

Theorem 3.1. The Characterization Theorem

The following two properties are equivalent for any interpretation IN of type L or L-tagged (for a given ST language l):

1. For any l-program P and any demand D for outputs of P, the set $S_{P,D}$ is either empty or contains a least element.
2. IN uses only stable functions.

In order to prove this theorem we first need several lemmas.

Lemma 3.1. If IN uses only stable functions, then for any l-program P the corresponding function τ_P is stable.

Proof. Since the f_i's are stable functions by assumption, and Pr_k is always a stable function for any k, the definition of τ_P implies that all its projections are stable. Hence τ_P is also stable (See [Ber79]). □

Lemma 3.2. Let P be an l-program and let v be a legal valuation for P. Then, $v \subseteq H_P$ and $v \subseteq \tau_P(v)$.

Proof. Let us look at the projected chain of P and v: $\{v_i\}_{i=1}^{\infty}$.

1. By definition 3.1, $\forall i \geq 1\ v_{i+1} = glb\{\tau_P(v_i), v\}$. Therefore, $\forall i \geq 1\ v_{i+1} \subseteq \tau_P(v_i)$. From this and from the fact that $\forall i \geq 1\ v_i \subseteq v$ (property of the chain), we can derive that $\forall i \geq 0\ v_{i+1} \subseteq \tau_P(v)$ (τ_P is monotonic). Since $v = lub\{v_i\}_{i=1}^{\infty}$ (legality of v), it follows that $v \subseteq \tau_P(v)$.
2. Let $\{w_i\}_{i=0}^{\infty}$ be the chain defined by:

$$w_1 = \begin{cases} H_P X & \text{if } X \text{ is an input variable of } P \\ [\,] & \text{otherwise} \end{cases}$$
$$w_{i+1} = \tau_P^i(w_1)(= \tau_P(w_i)) \quad (i \geq 1)$$

   $lub\{w_i\} = H_P$ by definition of H_P. Also, $\forall i \geq 1\ v_i \subseteq w_i$ (by induction on i). Therefore, $v = lub\{v_i\} \subseteq lub\{w_i\} = H_P$.

□

Lemma 3.3. If IN uses only stable functions, then for any l-program P and any valuation v for P: (i) $v \subseteq H_P$ and (ii) $v \subseteq \tau_P(v) \Rightarrow v$ is a legal valuation for P.

Proof. Assume otherwise. Let $\{v_i\}_{i=1}^{\infty}$ be the projected chain of P and v and define $v' = lub\{v_i\}_{i=1}^{\infty}$. Then $v' \subseteq v$ but $v' \neq v$. Let C be the set of cells in which there is a value $\neq \perp$ in v but not in v' (Recall that the basic values are taken from flat domains). Intuitively, the cells in C are those whose values were determined in v by a guess. They have the following two properties:

(1) $\forall c \epsilon C\ \tau_P(v)(c) = v(c)$
(2) For every $c \epsilon C\ \tau_P(v')(c) = \perp$.

The first property is because $v \subseteq \tau_P(v)$. The second – because $v'(c) = \perp \Rightarrow \forall v_i\ v_i(c) = \perp \Rightarrow \forall v_i\ \tau_P(v_i)(c) = \perp \Rightarrow \tau_P(v')(c) = \perp$ (since τ_P is continuous).

Let us now define the chain $\{w_i\}_{i=1}^{\infty}$ as in lemma 3.2 (part 2). The lub of this chain is H_P. In w_1 the cells in C contain $\perp$, but since $v \subseteq H_P$, they all

contain values $\neq\perp$ in H_P. Hence, there exists a *last* valuation v'' in $\{w_i\}_{i=1}^{\infty}$ for which all the cells in C contain $\perp$. Then there exists $c_0 \epsilon C$ s.t $\tau_P(v'')(c_0) = v(c_0)$ $(\neq\perp)$. Define a valuation $x = lub\{v, v''\}$ (x is well defined since both v and v'' are bounded by H_P). Since τ_P is monotonic, we can easily conclude that $\tau_P(x)(c_0) = \tau_P(v'')(c_0)$.

Define next another valuation y by: $y(c_0) = v(c_0)$, $y(c) = \perp$ otherwise. Obviously, $y \subseteq \tau_P(x)$ and so by stability of τ_P (see lemma 3.1) we conclude:

(3) Let $m = M(\tau_P, x, y)$. Then for all $z \subseteq x$: $y \subseteq \tau_P(z) \Leftrightarrow m \subseteq z$.

From (1) it follows that $y \subseteq \tau_P(v)$. Hence $m \subseteq v$. On the other hand $y \subseteq \tau_P(v'')$ implies that $m \subseteq v''$. Hence $m \subseteq glb(v, v'')$ and so $\forall c \epsilon C$ $m(c) = \perp$. Therefore $m \subseteq v'$ (v' is exactly as v except that the cells of C contain $\perp$ in it). From this, from (2) and from the fact that τ_P is monotonic it follows that $\forall c \epsilon C$ $\tau_P(m) = \perp$. On the other hand $y \subseteq \tau_P(m)$ (from (3)) and so $\tau_P(m)(c_0) \neq \perp$. A contradiction. □

Corollary 3.1.

Suppose IN uses only stable functions and let P be an l-program. Then a valuation v for P is legal iff (i) $v \subseteq H_P$ and (ii) $v \subseteq \tau_P(v)$.

3.5. Proof of the Characterization Theorem (theorem 3.1)

($\Rightarrow$) Suppose IN uses only stable functions.
Let P be any l-program and let D be any demand for outputs of P.

Assume first that there is an output line O and a demand for data token i on O, but $H_P O[i] = \perp$. For any legal valuation v of P, $v \subseteq H_P$ (see lemma 3.2). In particular, $vO \subseteq H_P O$. Hence $vO[i] = \perp$. Therefore, none of the legal valuations for P is output-complete w.r.t. D. Hence $S_{P,D}$ is empty.

Next, consider the case that for all the output lines of P, all the demanded data tokens appear in H_P. The set $S_{P,D}$ is then not empty, because H_P belongs to it. Obviously, $S_{P,D}$ is a subset of a domain C which is a product of sequence domains. It is easy to see that any nonempty subset of C has a glb in C. Let us denote the glb of $S_{P,D}$ by v'. We will finish by showing that $v' \epsilon S_{P,D}$.

Now, the fact that $v \subseteq H_P$ and v is output-complete $\forall v \epsilon S_{P,D}$, implies that v' has the same properties. By lemma 3.3, it remains to show that $v' \subseteq \tau_P(v')$. Assume otherwise. Since $v' \subseteq H_P$, we have $\tau_P(v') \subseteq \tau_P(H_P) \subseteq H_P$ (τ_P is monotonic). Therefore, both v' and $\tau_P(v')$ are bounded by H_P, and so our assumption that $v' \not\subseteq \tau_P(v')$ entails:

(1) there is a cell c in C that contains a value $\neq\perp$ in v' and $\perp$ in $\tau_P(v')$.

By lemma 3.2, $\forall v \epsilon S_{P,D}$ $v' \subseteq v \subseteq \tau_P(v)$. Hence:

(2) $\forall v \epsilon S_{P,D}$, the cell c contains value $\neq\perp$ in $\tau_P(v)$ (equal to its value in v').

Define now y as follows: $y(c) = v'(c)$ and $y(a) = \perp$ otherwise. Since $y \subseteq \tau_P(H_P) \subseteq H_P$ and τ_P is stable by lemma 3.1, we can conclude:

(3) Let $m = M(\tau_P, H_P, y)$. Then $\forall z \subseteq H_P :\ y \subseteq \tau_P(z) \Leftrightarrow m \subseteq z$.

From (2) it follows that $\forall v \epsilon S_{P,D}, v \subseteq H_P$ and $y \subseteq \tau_P(v)$. This fact together with (3) imply that m is a lower bound of $S_{P,D}$. Hence $m \subseteq v'$. From this and from (3) we conclude that $y \subseteq \tau_P(v')$. Hence the definition of y implies that cell c contains a value $\neq \perp$ in $\tau_P(v')$. This contradicts (1).

($\Leftarrow$) Let us assume that for any l-program P and any demand D for outputs, $S_{P,D}$ is either empty or contains a least element. We show that IN uses only stable functions.

Assume otherwise. Then for some f of type $\sigma_1\_stream \to \sigma_2\_stream$, $IN(f)$ is an unstable function of type $E^{\sigma_1} \to E^{\sigma_2}$. (For $IN(f) : E^{\sigma_1} * \ldots * E^{\sigma_n} \to E^{\sigma_{n+1}}$ the proof is similar). For convenience we shall denote $IN(f)$ simply by f. From definition 3.4, it follows that there exist $A \epsilon E^{\sigma_1}$ and $B \epsilon E^{\sigma_2}$ such that $B \subseteq f(A)$, but the set $U = \{z|\ z \subseteq A \text{ and } B \subseteq f(z)\}$ does not contain a least element.
Now, let us look at the following l-program P :

$$\begin{array}{rrcl} Graph: & O & = & f(I) \\ Input: & I & = & A \end{array}$$

Obviously, $H_P = (A, f(A))$.
Next, define $D(O) = \{i|\ B[i] \neq \perp\}$. The fact that $B \subseteq f(A)$ entails that H_P is output-complete w.r.t. D, and so $S_{P,D}$ is not empty. Hence, by assumption, $S_{P,D}$ contains a least element.
Now, (vI, vO) is a legal valuation of P iff :

1. $vI \subseteq A$
2. The lub of the projected chain $\{v_i\}_{i=1}^{\infty}$ of P and (vI, vO) is (vI, vO).
   In the present case, this sequence is defined as follows:
   $v_1 = (vI, [\,])$,
   for $j > 1 :\ v_j = glb\{\tau_P(v_{j-1}), (vI, vO)\} = glb\{(vI, f(vI)), (vI, vO)\}$ Hence $lub\{v_i\}_{i=1}^{\infty} = (vI, vO)$ iff $vO \subseteq f(vI)$.

On the other hand, a legal valuation (vI, vO) for P is output-complete w.r.t D iff: $\forall i > 0,\ \ B[i] \neq \perp \Rightarrow vO[i] \neq \perp$. Since B and vO are bounded by $f(A)$ (from 1, 2 and the fact that f is monotonic), this is equivalent to $B \subseteq vO$. It follows that:

$$S_{P,D} = \{(vI, vO)|\ vI \subseteq A \text{ and } B \subseteq vO \subseteq f(vI)\}.$$

Let (vI', vO') denote the least element of $S_{P,D}$. We have:

- $(vI', vO') \epsilon S_{P,D} \Rightarrow vI' \subseteq A$ and $B \subseteq vO' \subseteq f(vI') \Rightarrow vI' \epsilon U$.
- $\forall z \epsilon U,\ z \subseteq A$ and $B \subseteq f(z) \Rightarrow \forall z \epsilon U,\ (z, f(z)) \epsilon S_{P,D} \Rightarrow \forall z \epsilon U\ vI' \subseteq z$.

It follows that vI' is the least element of U. This contradicts our assumption that U does not contain a least element.

3.6. On Pingali Conditions for Legality of Valuations

Corollary 3.1 above, enables us to check legality of valuations by using conditions (i) and (ii). This is much easier than checking it directly by definition 3.1.

For the *sequential* case legality is actually defined in [Pin86] by these two conditions. No attempt to justify the definition is made, though. Corollary 3.1 formalizes Pingali's intuition: in the sequential case a valuation satisfies the two conditions iff it is a legal valuation. In fact, the corollary shows this for a larger class of functions – the class of the stable ones. It should be noted that Pingali himself has given in [Pin86] an example which shows that conditions (i) and (ii) are *not* sufficient for legality in the general case. His example contains indeed an unstable function – parallel-or.

4. Effectively Sequential Functions and Effective Demand Driven Evaluation

In the previous section, we gave a denotational semantics for demand driven evaluations of programs in ST languages, relative to interpretations of types L or L-tagged which use only stable functions. From now on all the interpretations we discuss are assumed to have this property, unless we say otherwise.

Our next goal is to look for an operational semantics for demand driven evaluations. Namely, the question is how can a demand driven evaluation of an ST-program *actually* be performed.

Such an operational semantics was given by A&P in [PiA85, Pin86] for a small subset of the allowed interpretations. As we explain below, their method is the most natural one, and we want to determine the largest possible set of interpretations to which it applies.

In section 4.1 below we motivate, explain and generalize the approach of A&P. In 4.2 we determine the scope of its applicability. Here the notion of *effective sequentiality* will take the role of stability in the previous sections.

4.1. The Basic Idea of Pingali and Arvind for Performing Demand Driven Evaluations Presented in a General Setting

Let l, IN, P and D be as above. We want that only the data tokens the values of which are needed for producing the requested outputs of P be created on the lines of its dataflow graph. To accomplish this goal, we need some way of informing the system about which data tokens are required to appear on the output lines of P. Then, each operator which produces data tokens on an output line of P, should produce the requested data tokens and only them. To do that, such an operator needs to have certain data tokens on its input lines. Therefore, the demands for outputs of an operator should be propagated in some way into demands for the outputs of the operators which produce the inputs of that operator. Such demands, in turn, will be satisfied if appropriate demands are propagated for outputs of other operators (the ones which produce data tokens on those lines), and so on. This means that there should be a propagation of demands in the *direction opposite* to the flow of data tokens in the dataflow graph of P. This propagation of demands can most naturally be implemented by using a dataflow graph in which there is a flow of data values of a *new* type, representing *demands*, in the direction opposite to the flow of the data tokens required for producing the wanted outputs.

These considerations led A&P in [PiA85, Pin86] to the idea of defining an operational semantics for the D-driven evaluation of P, by the operational semantics for the *data* driven evaluation of another program P', which is obtained from P and D by expansion of P with lines for demand propagation. There is a one to one correspondence between the operators of P and some of the operators of P', and between the lines of P and some of the lines in P'. The connections between the operators and the lines in P remain for the corresponding operators and lines in P'. The demand D is translated into input streams of P', which are propagated through the "demand lines" of P' into demands for the operators of P'. The *data* driven evaluation of P' creates, on the data lines of P' that correspond to the lines of P, the *D-driven* evaluation of P.

We next briefly and informally describe how to transform the directed graph of P, in order to get the directed graph of P'. For examples refer to [PiA85, Pin86].

First, we define a *demand token* as a data token with a special value of a new type *demand*. We define a *demand line* as a line on which there is a flow of demand tokens, and call the series of their values - a *demand stream*.

For each data line X in P, we create in P' a data line X and a demand line DX. On DX there is a flow of demand tokens that describe which data tokens should be produced on X: the occurrence of a demand token number i on DX, means that there is a demand for data token number i on line X. To each output line O in P, line O is an output line of P' and line DO is an input line of P'. The input to line DO is a demand stream that describes the request D from line O.

The essence of the transformation is to propagate the demands for the outputs of the operators of P' into demands for the data tokens the values of which are needed for producing those outputs. To do that, a code for demand propagation is added in P' to each of the operators of P. This code gets demands for the outputs of the operator, and propagates them into demands for its inputs. Therefore, if in P there is an operator F with inputs $X_1, \ldots, X_n$ and outputs $Y_1, \ldots, Y_m$ then in P' there is an operator FD with inputs $X_1, \ldots, X_n, DY_1, \ldots, DY_m$, and outputs $DX_1, \ldots, DX_n, Y_1, \ldots, Y_m$.

In view of the names of the input and output lines of the various FD operators, it is clear that if in P line X is an output line of an operator F and an input line of an operator G, then in P' line X is an output line of FD and an input line of GD, and line DX is an output line of GD and an input line of FD. In this way, GD can produce on DX demands for the data tokens it needs on X in order to produce its demanded outputs.

It must be ensured that only input tokens that are requested by the operators of P', will be allowed to flow on the lines of P'. For that purpose, a new operator called $Gate$ is used. This operator receives as inputs a data stream X and a demand stream D, and creates an output data stream in which for all $i > 0$:

$$O[i] = \begin{cases} X[i] & \text{if } X[i] \neq \perp \text{ and } D[i] \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

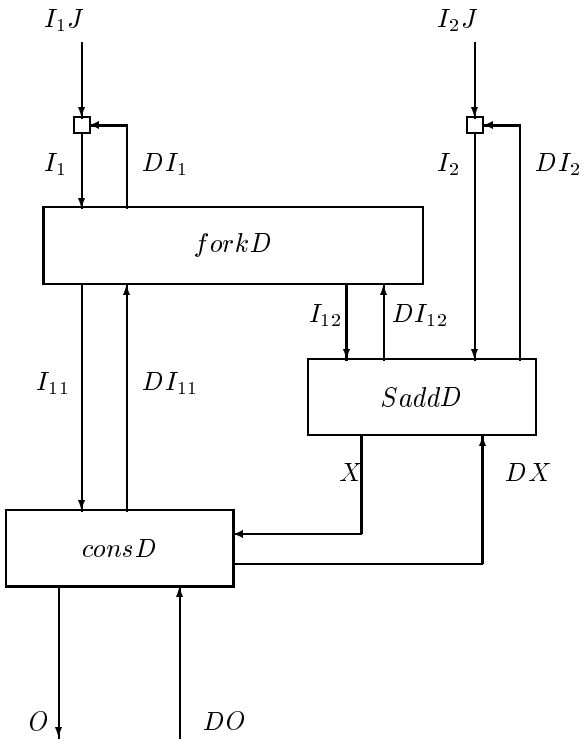
The $Gate$ operators are used to limit the flow of input data tokens on the lines of P' in the following manner: if I is an input line of P, then a new line IJ is an input line of P' and an input of a $Gate$ operator. Line DI is the second input of the $Gate$. Line I is the output of the $Gate$. The $Gate$ operator transfers to line

I only data tokens which are needed by the operators of P', and so unnecessary input data tokens are not used in P'.

Remark: Sometimes, a line in the dataflow graph of P is an input of several operators. In that case, a $fork_j$ operator is used: it replicates the input line to j output lines. $Fork_j$ operators are represented in dataflow graphs as thick dots (see example 2.1). In the dataflow graph of P, the inputs and outputs of $fork_j$ operators have the same names, but before transforming P into P', a different name should be given to each of the lines.

Example 4.1.

The graph representation of the graph part of program P' which corresponds to the program P of example 2.1:
(*Gate* operators are marked as small empty squares).

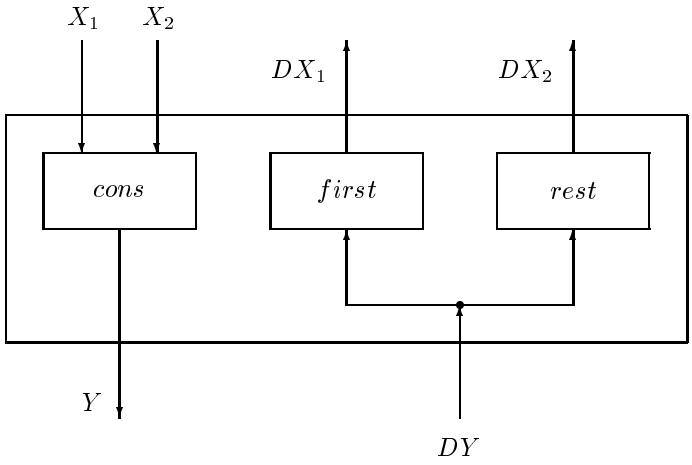


Remarks on the transformed programs:

1. In the original programs, any operator (except $fork_j$) has only one output line. This can be a property of the transformed programs as well. Any FD operator can be divided into several operators, where each has only one output line. Therefore, from now on we will call FD a *generalized operator*.
2. The method described here for performing demand driven evaluations treats only one way of specifying a demand for output: that which specifies exactly which are the needed data tokens (and not, e.g., that which requires that data tokens will be produced until a certain condition is satisfied). From now on we will only treat this case.

Example 4.2.

This example from [Pin86] describes a generalized operator $consD$ for the *cons* operator defined in example 2.2.



The definitions of the $first$ and $rest$ operators are as follows:
$first([x_1, x_2, \ldots]) = [x_1]$
$rest([x_1, x_2, \ldots]) = [x_2, \ldots]$

Remark: Recall that the dataflow operators in the operational semantics of ST languages are stable (namely correspond to stable functions). That is not necessary for the generalized operatos. They should only be continuous (that is, correspond to continuous functions). This is due to the fact that for STD programs we are only interested in data driven evaluations and not in demand driven ones.

4.2. The Need to Use only Effectively Sequential Functions when Performing Demand Driven Evaluations

The question which naturally arises now is whether every stable dataflow operator F has a compatible continuous generalized operator FD. If so, then the denotational semantics for demand driven evaluations given above is equivalent to the operational semantics described in 4.1. We give now an example that intuitively shows that this is not the case, i.e. that such a generalized operator *does not* always exist.

Example 4.3.

Define a stable operator F with input lines I_1, I_2, I_3 and output line O by:

$$F([x_1, x_2, \ldots], [y_1, y_2, \ldots], [z_1, z_2, \ldots]) = [f(x_1, y_1, z_1), f(x_2, y_2, z_2), \ldots]$$

where f is Berry's stable function ([Ber79]), defined as the minimal monotonic function satisfying:

$$f(\bot, true, false) = f(false, \bot, true) = f(true, false, \bot) = true$$

If a corresponding generalized operator FD exists it has inputs I_1, I_2, I_3 and DO, and outputs DI_1, DI_2, DI_3 and O. When there is a demand for an output data token number i, FD should *identify* which of the three input data tokens numbered i are needed in order to produce the demanded output. Since FD has no way of knowing *a priori* the values of those input data tokens, it will *always* have to propagate demands for all three of them. That is due to the fact that there are cases in which the value of the output data token cannot be calculated if only two input data tokens are demanded, no matter from which two input lines those tokens were demanded. (E.g. assume FD propagates demands for data tokens numbered i from lines I_1 and I_2. If the value of those two tokens is $true$, FD will not be able to produce the output data token without knowing the value of data token number i on line I_3, since $f(true, true, false) = true$ and $f(true, true, true) = \bot$).

On the other hand, if FD does propagate the demand for output into demands for each of the three input lines I_1, I_2, I_3, then there are cases in which it demands *unnecessary* data tokens. (E.g. if the values of data tokens numbered i on I_1 and I_2 are $true$ and $false$ respectively, then knowing their values is sufficient for determining the value of the output data token, no matter what the value of data token numbere i on I_3 is. Hence, in a demand driven evaluation there is no need to produce it!)

An analysis of the previous example indicates that in order to do its job, the generalized FD operators should not, under any circumstances, propagate demands for unnecessary data tokens. Therefore each output data token *must* have *critical* input data tokens, namely data tokens that must be produced in order to calculate the value of the demanded output. This implies that the original F operators should be sequential (namely, correspond to sequential functions). Moreover, it is not enough that critical input data tokens exist for every output data token. There should also be an algorithm that can *identify* those critical data tokens! In other words: the only functions to be used are functions $f : \Omega_{D^{\sigma_1}} * \ldots * \Omega_{D^{\sigma_n}} \to \Omega_{D^{\sigma_{n+1}}}$ for which there exists an algorithm which given $X = (X_1, \ldots, X_n)$ and k such that $f(X)[k] = \bot$, computes some $1 \leq i \leq n$ and

j for which $X_i[j] = \perp$ and $\forall Y \supseteq X\ \forall k\ f(Y)[k] \neq \perp \Rightarrow Y_i[j] \neq \perp$ [10]. Such functions are known as *effectively sequential* ([Tra75, Tra85]). Obviously, every effectively sequential function is sequential. The converse, however, fails (a counterexample is given in [Tra85, pp. 17–19]).

In each of the examples of A&P given in [PiA85, Pin86] of operators which have corresponding compatible generalized operators, only effectively sequential operators have indeed been used. On the other hand, the operator F from example 4.3 is not sequential (See [Ber79]). We conjecture that this is not an accident, and that the effectively sequential operators are exactly those opearators which have corresponding generalized continuous operators.

Let us informally describe how to create generalized operators for effectively sequential operators: If F is such an operator with inputs $I_1, \ldots, I_n$ and output line O, then a correspondig generalized operator FD for F includes:

- The operator F itself with input lines $I_1, \ldots, I_n$ and output line O'.
- A gate operator with input lines O' and DO and output line O.
- A generalized operator FD', with input line DO and some inputs from $I_1, \ldots, I_n$, and output lines $DI_1, \ldots, DI_n$. This operator reads demands for output data tokens and implements the algorithm for identifying the critical inputs data tokens, in order to propagate the demands for outputs into demands for inputs.

The *Gate* operator in FD on the output of F is needed because there are cases in which data tokens that were created in order to satisfy demands for certain outputs, cause the creation of other unneccessary ones. (For example the effectively sequential operator $F([a_1, a_2, \ldots]) = [a_1, a_1, a_2, a_2, \ldots]$ will always produce outputs i and $i+1$ even if only one of them is needed).

An example of a generalized operator which has been constracted in this way is given in example 4.2 – the *consD* operator (but note that in this particular case the *Gate* operator on the output of the *cons* operator is unnecessary and was therefore ommited).

5. Conclusion and Suggested Further Research

Our conclusion is that although the denotational definition of demand driven evaluation (given in definition 3.3) for ST programs (with semantics of types L or L-tagged) exists whenever the semantics use only stable functions, the natural way for actually performing such evaluation can only be used when we employ a subclass of the stable functions – the effectively sequential ones.

What remains to be done is to show this formally, by defining, given an effectively sequential operator F, the associated generalized operator FD in precise terms, and then to prove that it is actually compatible with F.

Another line of investigation is to see how much of the above theory is applicable to *nondeterministic* dataflow graphs, the semantics of which has received considerable attention in the dataflow literature (see, e.g. [Kon78, Kel78, BrB84]). Our first impression is that it *is* applicable, but only further study can confirm it.

[10] For Φ_{D^σ} the definiton is similar except that k and j must be, respectively, the first unfilled positions in $f(X)$ and X_i.

Acknowledgement

We would like to thank Boris A. Trakhtenbrot and Alexander Rabinovitz for introducing us to the subject and for their helpful comments on preliminary versions of this paper.

References

[ArG77] Arvind and Gostelow K.P.: *Some Relationships Between Asynchronous Interpreters of a Dataflow Language.* Proceedings of the IFIP WG2.2 Conference on Formal Description of Programming Languages, St. Andrews, Canada, 1977.

[Ber76] Berry G.: *Bottom-up Computation of Recursive Programs.* R.A.I.R.O. Informatique Theorique vol. 10 no. 3, pp. 47–82 mars 1976,.

[Ber78a] Berry G.: *Sequentialité de l'Evaluation Formelle des λ-expressions.* Proc. 3rd International Colloquium on Programming, Paris, March 28–30, 1978, DUNOD.

[Ber78b] Berry G.: *Stable Models of typed lambda - calculi.* Proc. 5th Coll. on Automata, Languages and Programming, July 1978, pp. 72–89.

[Ber79] Berry G.: *Modèles complètement adéquateet Stable des λ-calculus typés.* Thèse de Doctorat d'État, Université Paris VII, 1979.

[BrB84] Broy M. and Bauer F.L.: *A Systematic Approach to Language Constructs for Concurrent Programs.* Science of Computer Programming 4, pp. 103–139, North-Holland Publishing Company, 1984.

[Cur85] Curien P.: **Categorical Combinators, Sequential Algorithms and Functional Programming.** Ph.D Thesis, Univesite Paris 7, Paris, 1985.

[Fau82] Faustini A.: *The equivalence of an Operational and Denotational Semantics for Pure Dataflow.* Ph.D Thesis, University of Warwick, 1982.

[Iee82] IEEE. **Special issue on Dataflow Systems.** IEEE Comput. 15, 2 Feb 1982.

[Jon87] Jones K.: *A Formal Semantics for a Dataflow Machine — Using VDM.* Lecture Notes in Computer Science 254, pp. 331–335, 1987 (reprinted in Shaw R.C., Jones C.B.: *Case Studies in Systematic Software Development.* Prentice-Hall, 1990).

[Kah74] Kahn G.: *The Semantics of a Simple Language for a Parallel Programming.* Proceeding of the IPIF Congress 74, pp. 471–475, 1974.

[KaP] Kahn G. and Plotkin G.: *Structures de données concrètes.* Rapport IRIA - LABORIA 336 (D).

[Kel78] Keller R.M.: *Denotatinal Models for Parallel Programs with Indeterminate Operators.* Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.), pp. 337–366, North Holland, 1978.

[Kon78] Kosinski P.R.: *A Straightforward Denotational Semantics for Non-determinate Data Flow Programs.* Conf. Record of the 5^{th} Annual ACM Symposium on Principles of Programming Languages, pp. 214–221, 1978.

[LiS89] Linch Nancy A. and Stark E.W.: *A Proof of the Kahn Principle for I/O Automata.* Information and Computation 82, pp. 81–92, 1989.

[LoS84] Loeckx Jacques and Sieber Kurt.: *The Foundations of program Verification.* Teubner, 1984.

[MiS76] Milne R. and Strachey C.: *A Theory of Programming Language Semantics.* Chaprmand and Hall, 1976.

[Mil77] Milner R.: *Fully Abstract Models of Typed λ-calculi.* Theoret. Comput. Sci., vol 4, no 1, pp. 1–23, Feb 1977.

[Oli84] Oliveira J.N.: *The Formal Semantics of Deterministic Dataflow Programs.* Ph.D. Thesis, Department of Computer Science, University of Manchester, February 1984.

[PiA85] Pingali K. and Arvind: *Efficient Demand-Driven Evaluation part 1.* ACM Trans. on Prog. Lang. and Sys. vol 7, no 2, pp. 311–333, April 1985.

[Pin86] Pingali K.: *Demand Driven Evaluation on Dataflow Machines.* Ph.D Thesis, 1986.

[Ric82] Richmond G.: *A Dataflow Implementation of S_{ASL}.* M.Sc. thesis, Department of Computer Science, University of Manchester, 1982.

[ScB69] Scott D. and de Bakkar J.W.: *A Theory of Programs.* IBM Seminar unpublished notes, 1969.

[Sco70] Scott D.: *Outlines of Mathematical Theory of Computation.* 4th Annual Princeton Conf. Inform. Sc. and Sys. pp. 169–176, 1970.

[Sto77] Stoy J.: **Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.** M.I.T press, 1977.

[Tra75] Trakhtenbrot A.: *On Representation of Sequential and Parallel Functions.* Proceeding of 4th Symposium on Mathematical Foundations of Comp. Sci., vol 32, pp. 411–417, 1975.

[Tra85] Trakhtenbrot B.: *Topics in Typed Programming Languages, Lecture Notes.* CMU, Dep. of Comp. Sci., Fall Term, 1985.

[Tur79] Turner D.: *A New Implementation Technique for Applicative Languages.* Software — Practice & Experience, 9(1), pp. 31–39, 1979.

[Vui73] Vuillemin J.: **Proof Techniques for Recursive Programs.** Ph.D Thesis, Stanford University, 1973.