Pomset Interpretations of Parallel Functional Programs*

Paul Hudak
Steve Anderson

Yale University
Department of Computer Science
New Haven, CT

Abstract

A new framework is presented, based on the notion of a *partially ordered multiset* (or *pomset*), which is able to provide not only a precise operational semantics of parallel functional program evaluation, but also a handle through which to control such behavior. As an operational semantics, pomsets are able to distinguish between call-by-value, call-by-name, call-by-need, and call-by-speculation evaluation strategies (even though all but the first of these have the same standard semantics); and as a "handle" from which to control operational behavior, pomsets can express most of the behaviors achieved by previously proposed annotations that control not only evaluation order but also the spatial mapping of program to machine.

1 Introduction

The use of functional languages in writing parallel programs is hardly new, having its roots in research on dataflow machines which began almost 20 years ago. The claimed advantages of using functional languages include the facts that "parallelism is implicit" (that is, derived solely from data dependencies) and "results are determinate" (meaning one is free to choose a variety of, including parallel, execution orders). Based on these advantages, reseachers have concentrated on completely *automatic* techniques for parallel execution of functional programs – their goal has been for the user to remain completely unaware of the underlying parallelism. Dataflow and reduction machines [15], hybrid machines [3,6,11], and fancy compilation strategies [7] have all contributed to the success of this line of research.

On the other hand, there are times when the programmer would like to understand, and ultimately control, lower-level operational behavior. For example, if one wants to know what kind of parallelism to expect from the expression $f(x)$, one needs to know a great deal about the implementation, independent of the "standard" semantics (which we assume models normal-order reduction). In this example, one needs to know first of all whether or not the system is able to infer that the function f is strict. If so, is the argument then evaluated in parallel with the call? And if f is not strict, is the argument evaluated "eagerly" anyway? As we shall soon see,

*This research was supported in part by grants from the National Science Foundation (DCR-8451415) and the Department of Energy (FG02-86ER25012).

there are many variations on even these simple questions, and they are all reasonable ones to ask when trying to reason about the efficiency of a program. In fact, such questions are not restricted to *parallel* operational behavior – the subtle difference between lazy evaluation and "fully lazy evaluation" is an example of *sequential* behavior that is implementation dependent, and that can have a significant impact on performance. How can we provide this level of *understanding* to the user?

Even if one understands a particular implementation well, the resultant behavior is not always what the programmer desires. Indeed, there are often specific operational details that a programmer would like to control – details that one can never expect an automatic system to infer – but for which conventional functional languages have no means to express. Controlling evaluation order and mapping a program onto a particular machine topology are our canonical examples of such operational behaviors. How can we provide this level of *expressiveness* to the user?

One solution to both of these problems, of course, is to give up on functional languages altogether, and resort to a much lower-level language such as OCCAM, Concurrent Pascal, Ada, or some other parallel imperative language. But this would mean giving up all of the things that we *like* about functional languages, and would amount to throwing out the baby with the bath water! It is our thesis that most of the time a programmer need not be concerned with operational behavior, and thus functional languages are indeed an effective vehicle for parallel computation. On the other hand, we feel that the situations, however few, in which one needs to reason about and control such behavior cannot be ignored.

In this paper we present a new framework, based on the notion of a *partially ordered multiset* (or *pomset*), which is able to provide not only a precise operational semantics of parallel functional program evaluation, but also a handle through which to control such behavior. As an operational semantics, pomsets are able to distinguish between call-by-value, call-by-name, call-by-need, and call-by-speculation evaluation strategies (even though all but the first of these have the same standard semantics), including the subtle distinction between "lazy evaluation" and "fully lazy evaluation" (which are refined versions of call-by-need). Furthermore, as a "handle" from which to *control* operational behavior, pomsets allow us to express most of the behaviors achieved by previously proposed annotations that control not only evaluation order [1,8,14], but also the spatial mapping of program to machine [5,8,12]. In addition, we have used pomset-based annotations to express several other non-trivial operational behaviors, including the routing of data through a multiprocessor network (dually, the path taken in accessing a value), and the synchronized, lock-step execution of two (unrelated) recursive function calls.

There are those who believe that the need for meta-linguistic devices to refine operational behavior is a sign of weakness in functional languages. Thus there has been a tendency in the functional programming community to ignore such techniques as being too "ad hoc" and "impure." We do not feel this way, and one of our goals is to bring some respectability to this area by not only providing a formal semantics for such mechanisms, but also to show that the resulting programs can be very elegant, and are still a considerable improvement over their counterparts written in parallel imperative languages. The overall approach suggests what we call a "para-functional programming" methodology in which a programmer may refine operational behavior *without* restructuring the whole program or completely rewriting it in some other language. The separation of operational and functional behavior, both semantically and in the language itself, is what makes the methodology attractive; and by concentrating on functional behavior first, the methodology is consistent with the software engineering notions of "rapid prototyping" and "get it right first."

1.1 Why Pomsets?

The utility of pomsets as a tool for modelling concurrency is well demonstrated by Pratt [13], who is able to express a surprising diversity of concurrent behaviors clearly and concisely. Our contribution here is to show how they can be used to reason about and control parallel functional program evaluation. But why choose pomsets? In our search for a vehicle to not only express, but control, operational behavior, we considered and ultimately rejected several other alternatives. The most prominent of these were *dataflow graphs*, *temporal logic*, and *execution trees*, each of which we discuss briefly below.

Dataflow graphs actually share much of the appeal of pomsets, since they display naturally the partial-order of expression evaluation. However, they are inadequate for our purposes for several reasons: First, the most common versions of dataflow graphs are first-order and call-by-value. The extensions necessary to make them work with higher-order functions and lazy evaluation are cumbersome at best. Second, even if the appropriate extensions are made, they do not capture all of the detail that we are after, such as the fact that multiple demands for the same value do not cause recomputation. To capture such behavior requires a further interpretation of the dataflow graphs themselves (perhaps using pomsets!). Finally, it is not clear how dataflow graphs could be used as a basis from which to refine operational behavior.

Among other possible mechanisms for exressing control over evaluation order is temporal logic. However, temporal logic is unable to express the spatial relationships that we also wish control of, and seems to be inadequate, or at best cumbersome, in expressing the operational semantics of a language. Furthermore, Pratt has shown that pomsets in some sense *subsume* temporal logic in that they can be used as a model of temporal logic semantics (we do not know if the converse is true). For this reason we view temporal logic as a possible *meta-language* for refining operational behavior, but with pomsets still forming the semantic foundation.

In earlier work on the formal semantics of para-functional programming languages we introduced a notion of *execution trees* to capture the evaluation history of a program [4]. However, execution trees, being trees, do not capture any notion of *sharing*, which is crucial to the semantics of, for example, lazy evaluation. Extending the trees to graphs results in a form of "unwound" dataflow graph, which has all of the attendant problems mentioned above.

1.2 Overview of Paper

We begin our technical discussion in the next section with a definition of pomsets, processes, and various operations on them. To demonstrate their use as a foundation for operational semantics, we use them to describe four common parallel evaluation strategies in Section 3. Then in Section 4 we demonstrate their use as a handle from which to refine operational behavior. Taken together, these results suggest the possibility of a formal semantical framework with which to reason about operational behavior, which we discuss in Section 5. Finally, we discuss problems with our approach and point to future research in Section 6.

2 An Introduction to Pomsets

A *pomset* is a natural generalization of a *string*, in which the string's total, or linear order is replaced with a *partial order*. *Multisets* are needed instead of sets, because there may be multiple occurrences of the same entity – just as there may be multiple occurrences of the same character in a string. In fact, a string may be thought of as a totally-ordered multiset (or *tomset*). The following definitions are taken mostly from [13]:

Definition: A *partially ordered multiset*, or pomset, is a 4-tuple $\langle V, \Sigma, \preceq, \mu \rangle$, where:[1]

1. V is the *vertex set*, representing *events*.
2. Σ is the *alphabet*, representing *actions*.
3. $\preceq$ is a *partial order*, representing an *ordering between events*, and represented as a binary relation in $V \times V$.
4. μ is a *labelling function* with functionality $V \rightarrow \Sigma$.

Recall that as a partial order, $\preceq$ is reflexive, transitive, and anti-symmetric. When the distinction is necessary, we use the symbol $\prec$ as the irreflexive version of $\preceq$. Notationally, for a pomset p we write V_p, Σ_p, $\preceq_p$, and μ_p to denote its four components, and $e \in p$ as shorthand for $e \in V_p$. We denote the "empty pomset" by ϵ.

Pomsets should be thought of as modelling the concurrency (temporal or otherwise) of events, the events being instances of particular actions. In our context, actions and events will be tied to the evaluation of expressions; the details of this are forthcoming.

Definition: We say that p is an *augment* of q, iff $V_p = V_q$, $\Sigma_p = \Sigma_q$, $\mu_p = \mu_q$, and $\preceq_p \supseteq \preceq_q$. In other words, p augments q by being as constrained or more constrained than q. The opposite of augment is *subsume*.

Definition: We say that p is a *prefix* of q, iff p is obtained from q by deleting some of the events of q, under the constraint that if u is deleted and $u \preceq v$, then v is also deleted. This corresponds to the notion of a string prefix. The opposite of prefix is *suffix*.

2.1 Operations on Pomsets

Given the notion of a pomset, one can imagine a variety of operations on them, operations that collectively form an *algebra* of pomsets. The choice of operations depends primarily on one's application. A fairly general set of operations is described in [13], from which we derive the following set suitable for our purposes:

Definition: The *concurrence* of pomsets p and q, denoted $(p \| q)$, is defined as the pomset $\langle V_p \cup V_q, \Sigma_p \cup \Sigma_q, \preceq_p \cup \preceq_q, \mu_p \cup \mu_q \rangle$. Concurrence corresponds to the notion of two pomsets happening concurrently – there is no ordering relationship between events chosen pairwise from p and q.

Definition: The *concatenation* of two pomsets p and q, denoted $(p \cdot q)$, is defined as the pomset $\langle V_p \cup V_q, \Sigma_p \cup \Sigma_q, \preceq_p \cup \preceq_q \cup (V_p \times V_q), \mu_p \cup \mu_q \rangle$. Concatenation corresponds to the notion of two pomsets happening sequentially – every event in p is forced to occur before every event in q.

Definition: The *orthocurrence* of two pomsets p and q, denoted $(p \otimes q)$, is defined as the pomset $\langle V_p \times V_q, \Sigma_p \times \Sigma_q, \preceq_p \times \preceq_q, \mu_p \times \mu_q \rangle$. Letting $\preceq = (\preceq_p \times \preceq_q)$, we thus have that $\langle a, a' \rangle \preceq \langle b, b' \rangle$ just when $a \preceq_p b$ and $a' \preceq_q b'$. Orthocurrence corrresponds to the conventional notion of cartesian product – the pairing of unrelated objects – but in addition preserves the internal structure (i.e. elemental ordering) of the objects.

Figure 1 shows a graphical representation of the concurrence, concatenation, and orthocurrence of the pomsets $p = 0 \cdot 1$ and $q = a \cdot b$.

[1] In [13] this definition is actually for a *labelled partial order*, or *lpo*, and a pomset is defined as the isomorphism class of an lpo. However this technical distinction is unnecessary in our context.

$$p \equiv 0 \to 1$$

$$q \equiv a \to b$$

$$p \| q \equiv \begin{pmatrix} 0 \to 1 \\ a \to b \end{pmatrix}$$

$$p \cdot q \equiv 0 \to 1 \to a \to b$$

$$p \otimes q \equiv \begin{pmatrix} \langle 0,a \rangle & \to & \langle 1,a \rangle \\ \downarrow & & \downarrow \\ \langle 0,b \rangle & \to & \langle 1,b \rangle \end{pmatrix}$$

Figure 1: Graphical Representations of Concurrence, Concatenation, and Orthocurrence

The observant reader will note that concurrence and concatentation are at opposite ends of a spectrum – concurrence represents minimal order, concatentation maximal (i.e. total) order. There are clearly a number of orders that fall between the two extremes. We can express such orderings through the notion of *restricted concatenation*, in which concatenation is restricted to those events satisfying a particular predicate. Informally, $p \cdot_{pred} q$ is a pomset in which every event in p occurs before every event in q that satisfies *pred*. Formally, $p \cdot_{pred} q = \langle V_p \cup V_q, \Sigma_p \cup \Sigma_q, \preceq_p \cup \preceq_q \cup (V_p \times \{e \in V_q \mid pred(e)\}), \mu_p \cup \mu_q \rangle$. A similar definition applies for $p_{pred} \cdot q$. The most common use of restricted concatenation is where *pred* simply tests for a particular label, in which case we replace the predicate with that label. For example, $p \cdot_x q$ specifies that every event in p must precede every event e in q such that $\mu(e) = x$.

One should note that $p \cdot q$ is an augment of $p \cdot_{pred} q$, which is in turn an augment of $p \| q$. Indeed, $p \| q$ is essentially the same as $p \cdot_{false} q$, where *false* is the empty predicate. Conceptually, the easiest way to create an augment of an existing pomset is to somehow "add arrows" to the pomset's partial order. We occasionally have a need to do this, in which case we do so explicitly.

Finally, we introduce the notion of *pomset homomorphism*. The technical definition of pomset homomorphism follows exactly that of string homomorphism, and thus we omit the details. Notationally we write $p[x \mapsto q]$ for the result of applying to p the homomorphism that maps events with label x to the pomset q; i.e., "substitute q for x in p." For example:

$$((a \cdot b) \| (c \cdot a \cdot d))[a \mapsto (u \| v)] = ((u \| v) \cdot b) \| (c \cdot (u \| v) \cdot d)$$

2.2 Processes

Sometimes a single pomset is not sufficient to model a certain behavior, just as a single string is not always sufficient to characterize an entire language.

Definition: A *process* is a set of pomsets.

Intuitively, a process models something that may exhibit any one of a set of possible concurrent behaviors. From a mathematical perspective, a process is to a pomset as a language is to a string. The most general (i.e. least constrained) process over alphabet Σ is just the set of all possible pomsets constructed over Σ, which we denote $\Sigma\ddagger$, in analogy to Σ^* being the set of all strings over alphabet Σ.

We extend the previously defined operations on pomsets to processes, in the obvious pointwise manner. Furthermore, the following operations are useful:

Definition: The *augment closure* of a pomset p, written $\alpha(p)$, is the set of all augments of p.

Definition: The *prefix closure* of a pomset p, written $\pi(p)$, is the set of all prefixes of p.

Definition: The set of *linearizations* of a pomset p, written $\lambda(p)$, is the set of all linear (i.e. totally ordered) augments of p.

3 Pomset Interpretations of Functional Programs

3.1 Preliminaries

For purposes of exposition we use an extremely simple functional language which we will call *PFL*, whose abstract syntax is given by:

$$\begin{array}{rcll} c & \in & Con, & \text{constants, including primitive functions.} \\ x & \in & Id, & \text{identifiers.} \\ e & \in & Exp, & \text{expressions, defined by:} \\ & & e ::= & c \mid x \mid b \mid \lambda x.e \mid e_1\ e_2 \mid fix\ x.e \end{array}$$

PFL can be viewed as the unrestricted lambda calculus with constants. Constructs such as *letrec* and *whererec* can be transformed easily into a *fix* expression if need be. *Con* normally embodies all primitive functions, but for clarity we give specific examples involving the conditional and strict arithmetic operators, using the syntax $e_1 \rightarrow e_2, e_3$ and $e_1\ op\ e_2$, respectively.

Without loss of generality, we make two simplifying assumptions about PFL programs: First, all bound variables are unique. This is convenient when dealing with scoping rules. Second, every expression has associated with it a unique *label*. Labels are needed to distinguish syntactic expressions that otherwise would appear identical – for example, multiple occurrences of the same bound variable. When necessary, we write ${}^{l}e$ for an expression e whose label is l.

We assume the existence of two semantic functions, $\mathcal{E}_a$ and $\mathcal{E}_n$, that compute the standard applicative-order and normal-order denotational semantics, respectively, of PFL. More specifically, their functionality is given by:

$$\begin{array}{l} \mathcal{E}_a\colon Exp \rightarrow Env \rightarrow D \\ \mathcal{E}_n\colon Exp \rightarrow Env \rightarrow D \\ d \in D = Bas + (D \rightarrow D) \\ env \in Env = Id \rightarrow D \end{array}$$

The base domain *Bas* is left unspecified, but is presumed to contain the necessary objects to capture the meaning of elements of *Con*.

Henceforth the phrase "standard semantics" shall mean the semantics computed by one of these two semantic functions, depending on whether one wishes applicative-order or normal-order semantics.

3.2 Actions and Events in Expression Evaluation

A good starting point in making the connection between pomsets and expression evaluation is to decide what, exactly, we wish our pomsets to model – in other words, what is the meaning to be attached to *actions* and *events*? It should be clear that our concerns are rooted in very operational issues, and the meaning that we seek is not something typically captured in a language's standard denotational semantics.

Although there are many complicating details in any particular evaluation strategy, we have found that it is sufficient to consider as our set of underlying actions simply the *demand* for and

return of each syntactic expression's value. Thus for every labelled expression ^{l}e we associate two events, D_{l_e} and R_{l_e}, corresponding to the demand and return, respectively, of e's value. When the context is clear we often omit either the label or expression part, and simply write D_l or D_e. In addition, if used in a context requiring a pomset, we interpret D_l, R_l, etc. as singleton pomsets.

A syntactic expression, of course, may be evaluated in many different contexts (i.e. environments). Furthermore, depending on the semantics being captured, there can be more than one demand for an expression in a particular context, and for each such demand there is typically a corresponding return of value. Each of these occurrences of a particular action is, of course, simply an *event* in our model. The pomset interpretation of a PFL program is thus a pomset (or possibly set of pomsets) that captures the demand/return behavior resulting from program evaluation.

Now normally there are other events (actually, a pomset of events) that intervene between the demand and return of an expression e, corresponding naturally to the evaluation of other things needed to compute e's value. Of particular interest is the evaluation of an *identifier* – if the value was previously computed there may be *no* intervening events, otherwise there may be an entire pomset of events, corresponding to the evaluation of the actual parameter to which the identifier was bound.

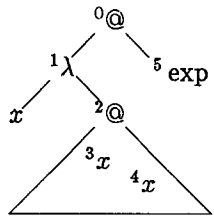
To make these and other ideas more concrete, we shall take the reader on a tour of pomset interpretations of four common evaluation strategies, shown graphically in Figure 2, and described intuitively below:

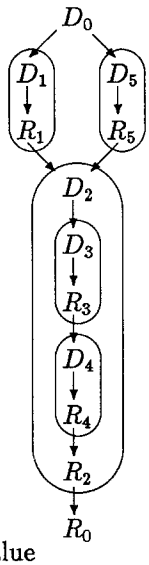
- In a *call-by-value* semantics there is *never* an intervening pomset, because the expression corresponding to the actual parameter is completely evaluated *before* the call.
- Conversely, in a *call-by-name* semantics there is *always* an intervening pomset.
- Alternatively, in a *call-by-need* (i.e. lazy) semantics there is either *zero* (if the argument is never needed) or *exactly one* occurrence of a bound variable with an intervening pomset.
- Finally, *call-by-speculation* is a blend between call-by-value and call-by-need, in which the evaluation of the arguments is begun at the time of the call, but the evaluation of the body proceeds as in call-by-need, blocking only if an argument is needed that hasn't been completely evaluated yet.

One of the complicating factors in all of these semantics is the proper treatment of *higher-order functions*. Such treatment becomes especially critical in call-by-need evaluation, where in fact we shall discuss *two* versions of the semantics, one that we call *lazy evaluation* and one that we call *fully lazy evaluation* – the difference lies in the way higher-order functions share free variables.

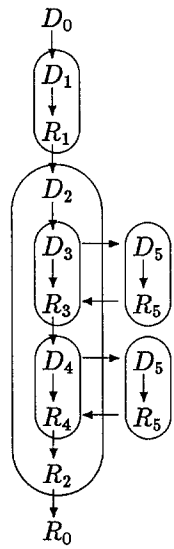
Note, as mentioned earlier, that the standard semantics will differentiate call-by-value from the other three, but will not differentiate call-by-need (where the result is "cached") from call-by-name (where the value is recomputed on each demand) or call-by-speculation. Nor will it differentiate lazy from fully lazy evaluation.

Also note that, although the standard semantics is deterministic, there are some aspects of operational behavior that are non-deterministic. For example, we may allow two expressions to be evaluated in parallel, but which actually begins (or ends) first is non-deterministic. The Church-Rosser property, of course, normally allows us to completely ignore this issue, since a deterministic result is guaranteed in either case. As we shall soon see, however, when *sharing* is manifest (such as in call-by-need evaluation) we must specify the non-deterministic behavior

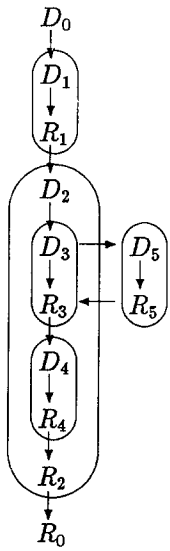




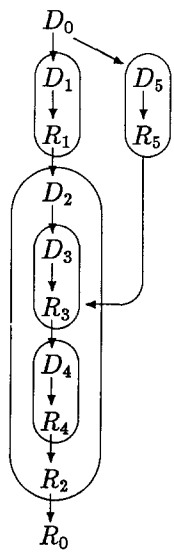
(a) Call-by-value



(b) Call-by-name



(c) Call-by-need



(d) Call-by-speculation

Figure 2: Four Common Evaluation Strategies

quite carefully, and we will find that a single pomset is inadequate – we must use a *set* of pomsets, or *process*.

In the remainder of this section we assume that our universe of pomsets is given by:

$$\Sigma = Id \cup \{R_l \mid l \in Lab\} \cup \{D_l \mid l \in Lab\} \cup \{error\}$$
$$p \in Pom = \Sigma\ddagger$$
$$ps \in Proc = \mathcal{P}(Pom)$$

Thus Pom is the domain of pomsets, and $Proc$ the domain of processes.

3.3 Pomset Interpretation of Parallel Call-by-Value

It is perhaps not surprising that call-by-value has the simplest operational semantics to capture. On the other hand, PFL has higher-order functions, and we must provide an adequate treatment of them immediately, which is not always an easy task. The key observation to be made is that the *evaluation* of an expression e can be described by a pomset, but the *value* of e may be a higher-order function which has embedded in it the "delayed" evaluation of *other* expressions, each also described by a pomset. We must provide a way to embody these delayed pomsets. Our solution is to pair with the pomset of an expression a higher-order object that, when applied later, may yield another pomset.[2] Of course, each result may in turn be applied again, and thus our embodiment must be a recursive object. We call this object a *behavior*, and its domain is defined formally by:

$$b \in Beh = Pom \times (D \to Beh \to Beh)$$

Note that the second component, which we call a *pomset abstraction*, is a function from a standard value d and a behavior b to another behavior b'. d and b represent the standard value and behavior, respectively, of the argument to which the pomset abstraction will eventually be applied. The result of that application is then the behavior b'.

We now define the domain of *behavior environments* that map identifiers to pomset abstractions:

$$benv \in Benv = Id \to (D \to Beh \to Beh)$$

and we define a special "error" pomset abstraction as follows:

$$err = \lambda d\ b.\langle error, err\rangle$$

Recall that $error \in \Sigma$, and is interpreted here as a singleton pomset.

Finally, this leads us to the definition of two semantic functions, $\mathcal{B}'$ and $\mathcal{B}$, that give pomset interpretations of PFL programs:

$$\mathcal{B}'\colon Exp \to Env \to Benv \to Beh$$
$$\mathcal{B}\colon Exp \to Env \to Benv \to Beh$$

We should point out that the presence of D in Beh and Env in $\mathcal{B}$ and $\mathcal{B}'$ reflects our integration of the standard semantics with the pomset semantics. This is necessary because we wish to provide an *exact* pomset interpretation, and thus we need to know the standard values of expressions in order to give proper meaning to, for example, conditional expressions.

[2]This solution is similar to that used in various other semantics, including strictness analysis [9], execution-tree semantics [4], and sharing analysis [2].

$\mathcal{B}'$ simply wraps a "demand/return event pair" around an expression's pomset, which is in turn derived using $\mathcal{B}$:

$$\mathcal{B}'[\![^l e]\!]\ env\ benv = \begin{array}{l}\text{let } \langle p, f\rangle = \mathcal{B}[\![e]\!]\ env\ benv \\ \text{in } \langle D_l{\cdot}p{\cdot}R_l, f\rangle\end{array}$$

It should be clear, then, that the "meat" of the pomset semantics is captured in $\mathcal{B}$, which is defined piecemeal below.

There is no pomset associated with evaluating a constant:

$$\mathcal{B}[\![c]\!]\ env\ benv = \langle \epsilon, K[\![c]\!]\rangle$$

where we assume that $K[\![c]\!]$ returns *err* for atoms, and a suitable pomset abstraction for primitive functions. Recall that ϵ is the empty pomset.

There is also no pomset associated with evaluating an identifier (recall that in call-by-value arguments are evaluated before the call):

$$\mathcal{B}[\![x]\!]\ env\ benv = \langle \epsilon, benv[\![x]\!]\rangle$$

Lambda abstraction and function application are (not surprisingly) the two most interesting cases, and to properly understand them they should be considered together:

$$\mathcal{B}[\![\lambda x.e]\!]\ env\ benv = \langle \epsilon, \lambda d\ \langle p, f\rangle.\mathcal{B}'[\![e]\!]\ env[d/x]\ benv[f/x]\rangle$$

$$\mathcal{B}[\![e_1\ e_2]\!]\ env\ benv = \begin{array}{l}\text{let } \langle p_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1,2 \\ \quad \langle p, f\rangle = f_1\ (\mathcal{E}_a[\![e_2]\!]env)\ \langle p_2, f_2\rangle \\ \text{in } \langle (p_1 \| p_2){\cdot}p, f\rangle\end{array}$$

There is no pomset associated with evaluating a function (i.e. lambda abstraction). However, the function's pomset abstraction has embedded in it the behavior of the function's body, which is computed in a standard environment and behavior environment that are updated accordingly. Note that during application, the function and argument are evaluated *in parallel* followed sequentially by the evaluation of the function body; all this is expressed by the pomset $(p_1 \| p_2){\cdot}p$.

Evaluating a conditional is straightforward:

$$\mathcal{B}[\![e_1 \rightarrow e_2, e_3]\!]\ env\ benv = \begin{array}{l}\text{let } \langle p_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1,2,3 \\ \text{in if } \mathcal{E}_a[\![e_1]\!]env \text{ then } \langle p_1{\cdot}p_2, f_2\rangle \\ \quad \text{else } \langle p_1{\cdot}p_3, f_3\rangle\end{array}$$

Note that the predicate is completely evaluated before either the consequence or alternative, as represented by the pomsets $p_1{\cdot}p_2$ and $p_1{\cdot}p_3$.

Strict binary operators are also straightforward:

$$\mathcal{B}[\![e_1\ op\ e_2]\!]\ env\ benv = \begin{array}{l}\text{let } \langle p_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1,2 \\ \text{in } \langle p_1 \| p_2, err\rangle\end{array}$$

Note that the two arguments are evaluated *in parallel*, as expressed by the concurrence $p_1 \| p_2$.

Finally, recursive functions require a recursive definition of the pomset abstraction:

$$\mathcal{B}[\![fix\ x.e]\!]\ env\ benv = \begin{array}{l}\text{let } x' = \mathcal{E}_a[\![fix\ x.e]\!]env \\ \quad \langle p', f'\rangle = \mathcal{B}'[\![e]\!]\ env[x'/x]\ benv[f'/x] \\ \text{in } \langle p', f'\rangle\end{array}$$

An Example. As an example of call-by-value pomset semantics, consider the simple expression:

$$\begin{array}{l}\text{let } f = \lambda x.\lambda y.\ x = 0 \rightarrow y,\ f\ (x-1)\ (y+1)\\ \text{in } f\ a\ b\end{array}$$

which ultimately just adds a to b. When put into proper PFL syntax, including labels on everything but constants, this becomes:

$$^{0}(^{1}(fix\ f.^{4}(\lambda x.^{5}(\lambda y.^{6}(^{7}(^{8}x = 0) \rightarrow ^{10}y,\ ^{11}(^{12}(^{14}f\ ^{15}(^{9}x - 1))\ ^{13}(^{16}y + 1))))))\ ^{2}a)\ ^{3}b$$

The corresponding call-by-value pomset, ignoring the trivial parallelism in the arithmetic expressions, is:

$$\begin{array}{c}((D_0(D_1 \cdot D_4 \cdot R_4 \cdot R_1 \| D_2 \cdot R_2) \cdot D_5 \cdot R_5 \cdot R_0) \| (D_3 \cdot R_3)) \cdot \\ [D_6 \cdot D_7 \cdot D_8 \cdot R_8 \cdot R_7 \cdot D_{11} \cdot ((D_{12} \cdot (D_{14} \cdot R_{14} \| D_{15} \cdot D_9 \cdot R_9 \cdot R_{15}) \cdot D_5 \cdot R_5 \cdot R_{12}) \| (D_{13} \cdot D_{16} \cdot R_{16} \cdot R_{13}))]^a \cdot \\ D_6 \cdot D_7 \cdot D_8 \cdot R_8 \cdot R_7 \cdot D_{10} \cdot R_{10} \cdot R_6 \cdot [R_{11} \cdot R_6]^a\end{array}$$

where the notation $[...]^a$ means a concatenations of the pomset "...." Although tedious, the reader should study this pomset carefully – it exposes very nicely the parallel evaluation of function and argument, and demonstrates well the nature of recursive function calls. Later we will contrast it with the corresponding pomset for call-by-name evaluation, and will find the differences striking.

Eager Evaluation. Before proceeding, there is an interesting variation of call-by-value operational semantics that is worth mentioning. Instead of requiring that the evaluation of the argument to a function complete before the body of the function begins execution, we could simply require that it complete before the entire call returns. This results in more parallelism, while retaining applicative-order standard semantics, and we refer to the resulting evaluation strategy as *eager evaluation.*

The necessary changes to the existing semantics to achieve this new behavior are minor, and essentially amount to replacing the pomset $((p_1 \| p_2) \cdot p)$ in the semantics of function application with $(p_2 \cdot_{R_x} (p_1 \cdot p))$, where x is the bound variable in the pomset abstraction associated with the function. This pomset expresses the fact that evaluation of the argument must complete only before the argument's value is used in the body of the function. The details are left to the reader.

3.4 Pomset Interpretation of Call-by-Name

Recall that in call-by-name evaluation an argument is not evaluated until it is needed – however, if it is needed more than once it is recomputed each time. Making this change to the call-by-value semantics is not difficult. In fact, the functionality of all domains remain the same, and other than substituting $\mathcal{E}_n$ for $\mathcal{E}_a$ only three equations for $\mathcal{B}$ change, those for (not surprisingly) identifiers, lambda abstractions, and applications.

For an identifier, we leave a "marker" which will eventually be replaced by the pomset to which that identifier is bound:

$$\mathcal{B}[\![x]\!]\ env\ benv = \langle [\![x]\!], benv[\![x]\!] \rangle$$

Note that we simply use the identifier itself as the marker; i.e., $[\![x]\!]$ (as the first component of the result) is to be interpreted as a singleton pomset whose single event is labelled with the identifier x.

As before, we treat lambda abstraction and function application together:

$$\begin{aligned}\mathcal{B}[\![\lambda x.e]\!]\ env\ benv = \langle \epsilon, \lambda d\ \langle p, f\rangle.\ \text{let } b = \mathcal{B}'[\![e]\!]\ env[d/x]\ benv[f/x] \\ F\langle p', f'\rangle = \langle p'[x{\mapsto}p], \lambda d\ b.F(f'\ d\ b)\rangle \\ \text{in } F\ b\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{B}[\![e_1\ e_2]\!]\ env\ benv = \ & \text{let } \langle p_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1,2 \\ & \langle p, f\rangle = f_1\ (\mathcal{E}_n[\![e_2]\!]env)\ \langle p_2, f_2\rangle \\ & \text{in } \langle p_1{\cdot}p, f\rangle\end{aligned}$$

Note now that in function application the pomset of the argument (p_2) is *not* incorporated directly into the result. Rather, it is passed to the pomset abstraction for the function, which in turn substitutes it (via the homomorphism $[x \mapsto p]$) for the "markers" (if any) found in the pomset of the function's body. The only complication is that the substitution must also be "propagated" into the body's pomset abstraction, which is accomplished through the (recursive) function F. Variations on this function will be the key to capturing other operational semantics, as the next few sections will reveal.

3.5 Pomset Interpretation of Call-by-Need

Call-by-need is an "optimization" of call-by-name, in which every expression is computed at most once. Whereas in the previous two semantics a single pomset was sufficient to capture the desired behavior, this is not so with call-by-need. A simple thought experiment should convince the reader of this: Consider again Figure 2c, in which an expression e has two concurrent uses of the bound variable x, represented by the pomsets $D_1{\cdot}x{\cdot}R_1$ and $D_2{\cdot}x{\cdot}R_2$. Suppose further that the evaluation of the expression to which x is bound is represented by the pomset $D_0{\cdot}p{\cdot}R_0$, and that it has not been previously demanded. Clearly the resultant pomset should have $R_0 \preceq R_1$ and $R_0 \preceq R_2$, simply reflecting the fact that the value of x must be completely computed before it can be used. On the other hand, in call-by-need evaluation D_0 cannot precede *both* D_1 and D_2, nor is it required to *follow* both. In other words, the resulting pomset should either have $D_1 \preceq D_0$ or $D_2 \preceq D_0$, but not both. This fact simply represents the non-deterministic "race" for the evaluation of the expression bound to x, and there is no way in which to express the result with a single pomset. Rather, a set of pomsets, or *process*, is needed, representing a choice between (possibly many) evaluation orders.

In anticipation of our need to capture this non-deterministic behavior, we define a form of restricted concatenation tailored specially for our use. Informally, $p_1 \odot_x p_2$ returns a *process* representing the call-by-need evaluation of p_2 in which x is bound to p_1. Each pomset in the result has the property that p_1 is substituted for *only one* occurrence of x in p_2, yet p_1 is required to precede *every* event R_x in p_2 that represents the return of x's value. Figure 3 shows an example of such an operation. Note in this example that ${}^2x$ never initiates the evaluation of p, since it always occurs *after* ${}^1x$.

We can define $\odot_x$ formally as follows:

$$\begin{aligned}p_1 \odot_x p_2 = \ & \text{let } \langle V, \Sigma, \preceq, \mu\rangle = p_1 \cdot_x p_2 \\ & \text{in } \{\langle V, \Sigma, \preceq \cup \{\langle v, w\rangle \mid w \in p_1\}, \mu\rangle[x{\mapsto}\epsilon] \\ & \quad \mid v \in p_2, \mu(v) = x, \neg(\exists(u \in p_2){:}\ \mu(u) = x, u \prec v)\}\end{aligned}$$

As with the other binary pomset operations, we extend $\odot_x$ to processes in the obvious way. We also define an auxiliary function *eval?* by:

$$eval?(x, ps) = \exists(p \in ps, v \in p){:}\ \mu(v) = x$$

$$p \odot_x \left(\begin{array}{c} D_1 \to {}^1x \to R_1 \cdots D_2 \to {}^2x \to R_2 \\ \\ D_3 \to {}^3x \to R_3 \end{array} \right) \equiv$$

$$\left\{ \left(\begin{array}{ccc} D_1 \to p \to R_1 & \cdots & D_2 \to R_2 \\ & \searrow & \\ D_3 & \longrightarrow & R_3 \end{array} \right), \left(\begin{array}{ccc} D_1 & \longrightarrow & R_1 \cdots D_2 \to R_2 \\ & \nearrow & \\ D_3 \to p & \to & R_3 \end{array} \right) \right\}$$

Figure 3: Example of Call-by-Need Restricted Concatentation

Note that since we are defining an exact semantics, x is either evaluated in *every* pomset in ps, or *none* of them.

We are now ready to define call-by-need pomset semantics, which we do by extending call-by-name semantics by first lifting the underlying domain structure to operate on processes rather than pomsets:

$$b \in Beh = Proc \times (D \to Beh \to Beh)$$

This change induces the following changes on $\mathcal{B}$ and $\mathcal{B}'$. First, a simple change to $\mathcal{B}'$ to return a process instead of a pomset:

$$\mathcal{B}'[\![{}'e]\!]\ env\ benv = \begin{array}{l} \text{let } \langle ps, f\rangle = \mathcal{B}[\![e]\!]\ env\ benv \\ \text{in } \langle \{D_l\}{\cdot}ps{\cdot}\{R_l\}, f\rangle \end{array}$$

There are several similar simple changes to $\mathcal{B}$:

$$\mathcal{B}[\![c]\!]\ env\ benv = \langle \{\epsilon\}, err\rangle$$

$$\mathcal{B}[\![x]\!]\ env\ benv = \langle \{[\![x]\!]\}, benv[\![x]\!]\rangle$$

$$\mathcal{B}[\![e_1 \to e_2, e_3]\!]\ env\ benv = \begin{array}{l} \text{let } \langle ps_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1, ..., 3 \\ \text{in if } \mathcal{E}_n[\![e_1]\!]env \text{ then } \langle ps_1{\cdot}ps_2, f_2\rangle \\ \quad \text{else } \langle ps_1{\cdot}ps_3, f_3\rangle \end{array}$$

$$\mathcal{B}[\![e_1\ op\ e_2]\!]\ env\ benv = \begin{array}{l} \text{let } \langle ps_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1, 2 \\ \text{in } \langle ps_1 \| ps_2, err\rangle \end{array}$$

$$\mathcal{B}[\![e_1\ e_2]\!]\ env\ benv = \begin{array}{l} \text{let } \langle ps_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1, 2 \\ \quad \langle ps, f\rangle = f_1\ (\mathcal{E}_n[\![e_2]\!]env)\ \langle ps_2, f_2\rangle \\ \text{in } \langle ps_1{\cdot}ps, f\rangle \end{array}$$

Recall that we extended pomset operations pointwise to processes, and thus except for a change in identifier names, the call-by-name semantics for the conditional, binary operator, and function application are identical to those given earlier. The key change, of course, arises in the semantics of lambda abstraction, where instead of "blindly" substituting the argument pomset into the function body, we do so "selectively" using $\odot_x$:

$$\mathcal{B}[\![\lambda x.e]\!]\ env\ benv = \langle \{\epsilon\}, \lambda d\langle ps, f\rangle.\ \begin{array}{l} \text{let } b = \mathcal{B}'[\![e]\!]\ env[d/x]\ benv[f/x] \\ \quad F\ \langle ps', f'\rangle = \begin{array}{l} \text{if } eval?(x, ps') \text{ then } \langle ps \odot_x ps', f'\rangle \\ \quad \text{else } \langle ps', \lambda d\ b.F(f'\ d\ b)\rangle \end{array} \\ \text{in } F\ b\rangle \end{array}$$

Note in the definition of F that if x appears in the pomset of the body (i.e., it "gets evaluated"), then its substitution is no longer propagated through the pomset abstractions. Although this is consistent with the notion of "evaluate at most once," we will see in the next section that it alone is not sufficient to capture "fully lazy evaluation."

The definition for fix remains the same.

An Example. To contrast this semantics with that of call-by-value, consider the same example given earlier:

$$^0(^1(fix\ f.^4(\lambda x.^5(\lambda y.^6(^7(^8x = 0) \rightarrow ^{10}y,\ ^{11}(^{12}(^{14}f\ ^{15}(^9x - 1))\ ^{13}(^{16}y + 1)))))))\ ^2a)\ ^3b$$

Its pomset, under call-by-need evaluation, is:

$$\begin{array}{c} D_0 \cdot D_1 \cdot D_4 \cdot R_4 \cdot R_1 \cdot D_5 \cdot R_5 \cdot R_0 \cdot D_6 \cdot D_7 \cdot D_8 \cdot D_2 \cdot R_2 \cdot R_8 \cdot R_7 \cdot D_{11} \cdot D_{12} \cdot D_{14} \cdot R_{14} \cdot D_5 \cdot R_5 \cdot R_{12} \cdot \\ [D_6 \cdot D_7 \cdot D_8 \cdot D_{15} \cdot D_9 \cdot R_9 \cdot R_{15} \cdot R_8 \cdot R_7 \cdot D_{11} \cdot D_{12} \cdot D_{14} \cdot R_{14} \cdot D_5 \cdot R_5 \cdot R_{12}]^{a-1} \cdot \\ D_6 \cdot D_7 \cdot D_8 \cdot D_{15} \cdot D_9 \cdot R_9 \cdot R_{15} \cdot R_8 \cdot R_7 \cdot D_{10}[D_{13} \cdot D_{16}]^a \cdot D_3 \cdot R_3 \cdot [R_{16} \cdot R_{13}]^a \cdot R_{10} \cdot R_6 \cdot [R_{11} \cdot R_6]^a \end{array}$$

This result should be compared to that given earlier for call-by-value. There are several things worth noting:

1. There is no parallelism (except the trivial parallelism in the arithmetic expressions). This is due to the fact that in function applications the argument is not evaluated in parallel with the function; rather the argument's evaluation is delayed until it is needed.

2. Each delayed evaluation causes a "non-local transfer of control" to the context in which the argument was bound. For example, the sequence $D_7 \cdot D_8 \cdot D_2 \cdot R_2 \cdot R_8 \cdot R_7$ represents the evaluation of x in the *first* call to f, and $D_7 \cdot D_8 \cdot D_{15} \cdot D_9 \cdot R_9 \cdot R_{15} \cdot R_8 \cdot R_7$ represents the evaluation of x in each of the remaining (recursive) calls to f.

3. Whereas x's evaluation is only delayed one level upon each call to f, y's evaluation is delayed until the recursion reaches its deepest level, at which point it "unwinds" and "reaches back" to every level above it, performing a additions of 1. This process is represented by the pomset $D_{10}[D_{13} \cdot D_{16}]^a \cdot D_3 \cdot R_3 \cdot [R_{16} \cdot R_{13}]^a \cdot R_{10}$.

We feel that these important yet subtle differences between call-by-value and call-by-need evaluation are made acutely apparent through the use of pomset semantics.

Fully Lazy Evaluation. Consider the PFL expression $g(f\ a)$, where f and g are defined by:

$$\begin{array}{l} f = \lambda x.\lambda y.x \\ g = \lambda y.(y\ 1) + (y\ 2) \end{array}$$

Adding labels, we have:

$$\begin{array}{l} f = {}^1(\lambda x.^2(\lambda y.^3 x)) \\ g = {}^4(\lambda y.^5(^6(^8y\ 1) + {}^7(^9y\ 2))) \end{array}$$

where we consider the definitions of f and g to be substituted directly into the result expression $(g\ ^{10}(f\ a))$. The pomset for the result, as given by the call-by-need semantics just defined, is:

$$(D_{10} \cdot D_1 \cdot R_1 \cdot D_2 \cdot R_2 \cdot R_{10})\ \odot_y\ (D_4 \cdot R_4 \cdot D_5 \cdot \left(\begin{array}{c} (D_6 \cdot D_8 \cdot y \cdot R_8 \cdot D_3 \cdot D_a \cdot p_a \cdot R_a \cdot R_3 \cdot R_6) \| \\ (D_7 \cdot D_9 \cdot y \cdot R_9 \cdot D_3 \cdot D_a \cdot p_a \cdot R_a \cdot R_3 \cdot R_7) \end{array} \right) \cdot R_5)$$

where we assume p_a to be the pomset for a.

But there is something slightly wrong with this result: Note that, although the "competition" for the evaluation of y is handled correctly (and thus y is evaluated only once), the evaluation of a is done *twice*, once for each application of y! This curious behavior is what Hughes describes as being *not* fully lazy [10], and it is a property actually exhibited by some existing functional language implementations. The problem stems from our delaying the evaluation of an argument to only one level of function application, rather than to the arbitrary number of levels possibly exhibited by a higher-order function. The substitutions must somehow be delayed until all "sources of contention" can be identified.

Unfortunately, whereas it was easy to implement one level of delay, we cannot delay substitutions arbitrarily without some extra mechanism. That mechanism is what we call a *substitution list*, or just *substitution*. Instead of propagating substitutions directly into pomset abstractions, we delay them by pairing them with the pomset abstraction until later needed. The resulting pair, pomset abstraction plus substitution list, is analogous to the "code plus environment" representation of a conventional closure.

Now for the details. The domain of behaviors is redefined so as to have a substitution list component:

$$b \in Beh = Proc \times Subst \times (D \to Beh \to Beh)$$
$$u \in Subst = (Id \times Proc)^*$$

We denote a particular substitution list as $[x_1 \mapsto ps_1, x_2 \mapsto ps_2, ..., x_n \mapsto ps_n]$ (similar to a homomorphism); the empty substitution list is thus $[]$.

This change induces many small changes in $\mathcal{B}$ and $\mathcal{B}'$:

$$\mathcal{B}'[\![^l e]\!]\ env\ benv = \text{let } \langle ps, u, f\rangle = \mathcal{B}[\![e]\!]\ env\ benv \text{ in } \langle \{D_l\}\cdot p\cdot\{R_l\}, u, f\rangle$$

$$\mathcal{B}[\![c]\!]\ env\ benv = \langle \{\epsilon\}, [], err\rangle$$

$$\mathcal{B}[\![x]\!]\ env\ benv = \langle \{[\![x]\!]\}, [], benv[\![x]\!]\rangle$$

$$\mathcal{B}[\![e_1 \to e_2, e_3]\!]\ env\ benv = \text{let } \langle ps_i, u_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1,2,3 \text{ in if } \mathcal{E}_n[\![e_1]\!]env \text{ then } \langle ps_1\cdot ps_2, u_2, f_2\rangle \text{ else } \langle ps_1\cdot ps_3, u_3, f_3\rangle$$

$$\mathcal{B}[\![e_1\ op\ e_2]\!]\ env\ benv = \text{let } \langle ps_i, u_i, f_i\rangle = \mathcal{B}'[\![e_i]\!]\ env\ benv,\ i = 1,2 \text{ in } \langle ps_1 \| ps_2, [], err\rangle$$

$$\mathcal{B}[\![fix\ x.e]\!]\ env\ benv = \text{let } x' = \mathcal{E}_a[\![fix\ x.e]\!]env,\ \langle p', u', f'\rangle = \mathcal{B}'[\![e]\!]\ env[x'/x]\ benv[f'/x] \text{ in } \langle p', u', f'\rangle$$

The only difference between these equations and the corresponding ones for call-by-need is that they contain substitution lists, which are either "carried along" (such as for the conditional) or set to nil (as for *op*).

To make the reading of lambda abstraction easier, we lift out the definition of the function F that propagates substitutions:

$$F(ps, u, acc) = \text{if } u = [] \text{ then } \langle ps, acc\rangle \text{ else let } [x \mapsto p, rest] = u \text{ in if } eval?(x, ps) \text{ then } F(p \odot_x ps, rest, acc) \text{ else } F(ps, rest, [x \mapsto p, acc])$$

Given the previous discussion, this definition should be fairly clear – if an application of a pomset abstraction results in x being evaluated, then x's substitution is performed, otherwise it is retained in the substitution list. As before, F is used in the semantics of lambda abstraction, given below alongside function application:

$$\mathcal{B}[\![\lambda x.e]\!]\ env\ benv = \langle\{\epsilon\}, [], \lambda d\ \langle ps, u, f\rangle.\ \text{let } \langle ps', u', f'\rangle = \mathcal{B}'[\![e]\!]\ env[d/x]\ benv[f/x] \\ \langle ps'', u''\rangle = F(ps', [x \mapsto ps, u], u') \\ \text{in } \langle ps'', u'', f'\rangle\ \rangle$$

$$\mathcal{B}[\![e_1\ e_2]\!]\ env\ benv = \text{let } \langle ps_i, u_i, f_i\rangle = alpha(\mathcal{B}'[\![e_i]\!]\ env\ benv),\ i = 1,2 \\ \langle ps, u, f\rangle = f_1\ (\mathcal{E}_n[\![e_2]\!]env)\ \langle ps_2, u_1 \cup u_2, f_2\rangle \\ \text{in } \langle ps_1 \cdot ps, u, f\rangle$$

To understand these equations, realize that the sharing that is responsible for the contention for values is manifested solely through bound variables. This means that, if the bound variable is indeed evaluated, then it is safe to do the substitution at the point at which the binding is first made. On the other hand, if it is *not* evaluated, then it must be "carried along" in the substitution component of the pomset abstraction.

There is one subtle aspect of function application: note that u_1 is combined with u_2 when passed to the pomset abstraction f_1. This reflects the fact that a pomset abstraction in function application position *cannot* be further shared, and thus it is safe to perform its substitutions. Unfortunately, this raises one other complication, an unfortunate consequence of our decision to give pomset interpretations over an alphabet of syntactic objects: because expressions may be evaluated in different contexts, there is the possibility of name conflict. Although such name conflicts did not arise until now, it was an inevitable consequence of our need to delay substitutions more and more. We solve this problem simply by renaming (i.e. "alpha-converting") certain of the substitutions, which we do by the "pseudo-function" *alpha*, as shown.

Returning to the problematical example that began this section, we see that under our new semantics the inner concurrence (corresponding to the body of g) becomes:

$$(D_a \cdot a \cdot R_a)\ \odot_x\ (D_5 \cdot \begin{pmatrix} (D_6 \cdot D_8 \cdot y \cdot R_8 \cdot D_3 \cdot x \cdot R_3 \cdot R_6) \| \\ (D_7 \cdot D_9 \cdot y \cdot R_9 \cdot D_3 \cdot x \cdot R_3 \cdot R_7) \end{pmatrix} \cdot R_5)$$

which when combined into the whole result:

$$(D_{10} \cdot D_1 \cdot R_1 \cdot D_2 \cdot R_2 \cdot R_{10})\ \odot_y\ (D_4 \cdot R_4 \cdot \Bigg((D_a \cdot a \cdot R_a)\ \odot_x\ (D_5 \cdot \begin{pmatrix} (D_6 \cdot D_8 \cdot y \cdot R_8 \cdot D_3 \cdot x \cdot R_3 \cdot R_6) \| \\ (D_7 \cdot D_9 \cdot y \cdot R_9 \cdot D_3 \cdot x \cdot R_3 \cdot R_7) \end{pmatrix} \cdot R_5) \Bigg))$$

gives precisely the correct behavior for fully lazy evaluation.

3.6 Pomset Interpretation of Call-by-Speculation

Although Figure 2 given at the start of this section gives the basic intuition behind call-by-speculation, it fails to capture an important subtlety in the behavior. This subtlety is best described by an example: Suppose the call $^l(f\ a)$ is done via call-by-speculation, but a is never needed in the body of f (unlike the case in Figure 2). The resulting pomset will then not have a definitive "exit point." That is, when $\mathcal{B}'$ wraps $D_l \cdot R_l$ around the result, it must *not* require that the speculative evaluation of the argument complete before R_l (if it did, we would end up doing "eager evaluation" as described in Section 3.3). The correct pomset is shown in Figure 4, where p_f, p_a and p are the pomsets for f, a, and the body of f, respectively. If a does not terminate, of course, this results in the possibility of a "runaway process," or "irrelevant task,"

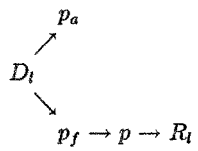


Figure 4: Correct Pomset for Call-by-Speculation

but that is exactly the behavior that we want, and is necessary in order to retain normal-order standard semantics.

It is possible to define a form of call-by-speculation in which the speculative *demand*, at least, for the argument is guaranteed to occur before the function call is initiated. In fact, in such a semantics a single pomset (i.e. singleton process) once again becomes sufficient, since we know immediately what the first demand on each argument is. However, the most general version (i.e., the one having the weakest ordering) does not have this constraint, and the pomset abstractions become the complicating factor again, because a speculatively invoked argument might be demanded later by a subsequently applied higher-order function.

The necessary changes to the formal pomset semantics are straightforward, but because of space limitations we only outline the approach, starting with the pomset semantics of call-by-need:

1. In the semantics for lambda abstraction, instead of returning essentially the pomset of the body, we return a concurrence of it and the pomset $D{\cdot}\hat{x}{\cdot}R$, where $\hat{x}$ represents the speculative demand for the argument.

2. Once an identifier is eventually demanded, a standard $\odot$ operation is performed on the *composite* result – i.e., on the result of the application as well as the context in which it lies. This requires delaying subsitutions even further.

3. Because we have delayed the establishment of the "connections" between pomsets even further, name clashes become more severe and we are forced once again to create unique contexts via alpha-conversion.

4 Using Pomsets to Control Operational Behavior

The general motivation for our work stems not just from the need for a formal mechanism through which to reason about parallel functional program evaluation. If we accept the fact that annotations or other meta-linguistic devices are needed to more precisely control operational behavior, then we also need a formal semantics to reason about such annotations. Furthermore, just as denotational semantics can be used as a tool for language design, pomset semantics can be used as a tool for meta-language design. One of the guiding principles of functional programming language research has been the insistence on formal methods – we are simply extending that principle into an area that we feel has been ignored in the past.

Of course, now that we have completed our tour of "standard" evaluation strategies (in total we actually explored *six* of them!), it should not be surprising that we can adequately describe the operational semantics of various proposed annotations. Rather than do that directly, however, we will describe informally the semantics of two simple annotations that can then be used to

achieve the effect of many of the proposed annotations. Our goal is not to propose a concrete syntax or semantics for such annotations, since it is not yet entirely clear to us exactly what the best combination of expressiveness, flexibility, and safety is. Our goal is only to set up the framework in which such techniques can be used, and to demonstrate that pomset algebra, possibly extended with logic, may be suitable as the meta-language.

We divide this section into parts, one to discuss the control of temporal order, the other the control of spatial order.

4.1 Controlling Evaluation Order

Our approach to controlling the evaluation order in a program will be to augment the pomset associated with an expression with pomset-like annotations. Specifically, an *augment expression* has the form "$exp \oplus pom$," where exp is any PFL expression and pom is a pomset of events over labelled expressions in exp. Rather than propose concrete syntax for pomset expressions, we will use the same mathematical notation that we have used throughout this paper. Similarly, labelled expressions will continue to be denoted by ${}^{l}e$.

If this technique is to work, then we must start with the most general of evaluation strategies (otherwise we are faced with the problem of *removing* constraints from a pomset), which as we will show in the next section is call-by-speculation. However, this is not normally considered to be the "default" evaluation strategy, and thus we take a slightly different approach, summarized in the following rules:

1. Fully lazy call-by-need is the default evaluation strategy (since this is the most common semantics used in practice).

2. If a labelled subexpression, as in "$(\ldots\ {}^{l}e\ldots) \oplus p$," is referenced in p, we consider that subexpression as being evaluated speculatively. In other words, we consider the overall expression to have been transformed into "$((\lambda x.\ \ldots x\ldots)\ {}^{l}e) \oplus p$," with the application being done via call-by-speculation.

3. For a labelled expression ${}^{l}e$, we use l (i.e. the label itself) to refer to the entire pomset associated with e, and D_l and R_l to refer (as always) to the demand and return events, respectively, of e. Finally, we let D and R refer to the demand and return events of the entire expression being augmented.

The operational semantics of an augment expression $e \oplus p$ is that e's pomset is modified to reflect any additional events or constraints in p. The formal definition is essentially identical to concurrence, except that we no longer consider the event sets to be disjoint (otherwise we would have no way of referring to existing events). If an event in p is not contained in e's pomset, then it is added to the resulting pomset as a new event. Some example should help clarify this interpretation.

Consider the application $(f\ a)$. Under a call-by-need interpretation the expression a may or may not be evaluated. We can turn this application into call-by-speculation by writing:

$$(f\ {}^{l}a) \oplus l$$

To achieve eager evaluation we write:

$$(f\ {}^{l}a) \oplus (l \cdot R)$$

which requires that the evaluation of a complete before the entire call completes. Finally, to achieve call-by-value we write:

$$({}^{m}f\ \ {}^{l}a) \oplus (l{\cdot}R_m)$$

which requires that the evaluation of a complete before the function is called.

As another example, suppose in the expression $(f\ x) + (g\ y)$ we wish to inhibit parallelism (perhaps out of concern for limited resources) by requiring that $(f\ x)$ be evaluated completely before $(g\ y)$. This can be done quite simply by:

$$({}^{l}(f\ x) + {}^{m}(f\ y)) \oplus (l{\cdot}m)$$

We are also experimenting with ways to use augment expressions in a recursive setting. For example, it seems desirable to allow them to do such interesting things as ensuring that two functions "recurse in lock-step," as in:

$$\begin{array}{l}(sum\ lst)/(len\ lst) \oplus (l{\cdot}m)^* \\ \quad \text{whererec } sum\ lst \ = \ (null?\ lst) \rightarrow 0,\ (head\ lst) + {}^{l}(sum\ (tail\ lst)) \\ \qquad\qquad len\ lst \ = \ (null?\ lst) \rightarrow 0,\ 1 + {}^{m}(len\ (tail\ lst))\end{array}$$

The intent here is for the pomset expression $(l{\cdot}m)^*$ to only allow zero or more evaluations of the expression labelled l followed by the expression labelled m. Both *sum* and *len* force successive tails of *lst* in synchrony, and thus both implicitly abandon their references to successive elements of *lst* in synchrony, thereby permitting evaluation in constant space. Normally to achieve this kind of synchronization one would have to restructure the program by combining *sum* and *len* into a single function that returned a composite result which was in turn decomposed for use where needed. On the other hand, we point out that this use of augment expressions is experimental, since we have not completely resolved the formal relationship between an arbitrary pomset expression such as $(l{\cdot}m)^*$ with the pomset that it is refining.

4.2 The Orthocurrence of Process and Processor

When writing functional programs for execution on a distributed multiprocessor, there is often a need to express the "process-to-processor mapping" explicitly. Many examples of such applications may be found in [5,8], where *mapped exressions*, having the simple form "*exp* on *pid*," are used to declare that the expression *exp* is to be executed on the processor identified by *pid*, and the variable *self* is used to reference the "currently executing processor," thus allowing relative rather than absolute mappings.

To capture this behavior it is obvious that the standard semantics must first be extended to include the meaning of the (dynamic) variable *self* – this is very straightforward, and in fact is done in [4]. In addition, the pomset semantics must have some notion of processor names, and a mechanism to attach them to the pomsets representing expression evaluation. This is easily done using the pomset operation *orthocurrence* discussed in Section 2.1 to pair events in the temporal ordering with events (i.e. "places") in the spatial ordering. We can then give a rough semantics for mapped expressions as follows:

$$\begin{array}{lll}\mathcal{B}[\![e \text{ on } pid]\!]\ env\ benv = & \text{let } d = \mathcal{E}[\![pid]\!]env & \\ & \quad \langle p, f\rangle = \mathcal{B}'[\![e]\!]\ env[d/self]\ benv & \\ & \text{in } \langle d \otimes p, f\rangle & \end{array}$$

where "d" in "$d \otimes p$" is to be interpreted as a singleton pomset. For a program with no annotations, all expression evaluations map to the same "root" processor.

Once this semantics is established, we can see that limiting the orthocurrence to singleton pomsets is unnecessary. In fact, there are many situations, particularly in scientific computation, where the programmer knows that the demands for some variable binding $x = exp$ will appear in a particular order along a path through the network. Mapped expressions say where an expression is evaluated, but say nothing about the path for data movement of the result. If the demands appear on a sequence of processors in the spatial order $i_0, i_1, i_2, \ldots, i_n$, it is desirable for the demand for x on i_0 to cause evaluation of exp on i_0, and for the demand on each other processor i_j to get the copy of the binding for x from its immediately preceding neighbor i_{j-1}.

Generalizing this idea, paths can in fact be represented unambiguously by any tree-shaped partial order, or ambiguously (allowing the possibilty of alternate data paths) by an unconstrained partial order. In other words, letting Ψ be the domain of processor names, we interpret a pomset over alphabet Ψ as a spatial ordering on the designated processors. By generalizing mapped expressions to take arbitrary pomsets of processor ids rather than singleton pomsets, we arrive at the desired semantics.

Although quite general, this approach may not be the most convenient in practice. Consequently, we have been experimenting with the notion of a *path function*, which specifies a tree-shaped partial order of processors – the root of the tree, which is a fixpoint of the path function, is where expression evaluation actually takes place. A path function annotation allows recursive definition of a data path, which is frequently required in scientific computation. A future paper will outline our experience with this class of annotations.

5 Toward a Standard Operational Semantics

In this section we discuss the viability of using pomsets and processes as a foundation on which to base a formal operational semantics for functional programs and the machines on which they are executed.

5.1 The Existence of a "Standard" Operational Semantics

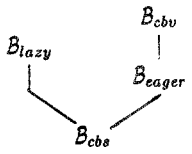
The reader has likely observed that the operational semantics in Section 3 became increasingly more complex as we matured in evaluation strategies, with call-by-need and call-by-speculation semantics being the most complex, and call-by-value being the least. This is an interesting observation, because in standard semantics call-by-need is usually considered the simplest, "most elegant," and easiest to reason about. On the other hand, programs relying on lazy evaluation are often recognized as being difficult to *trace* during debugging, because the delayed evaluation of expressions creates very non-local effects.[3] We believe that the complexity of our operational semantics is simply a reflection of that fact.

The question immediately arises as to whether there is one "standard" operational semantics that is most general – that is, that contains the least number of constraints on evaluation order. For a given standard semantics, we believe there is. Although we do not give direct evidence of that here, we can at least make the following interesting observations about the semantics given in Section 3.

First recall the definition of augmentation, which we now point out is a partial order. We write $p_1 \sqsubseteq p_2$ iff p_2 is an augment of p_1. Extending $\sqsubseteq$ to functions in the standard way, we have that $f \sqsubseteq g$ iff $f(a) \sqsubseteq g(a)$ for all a. Let $\mathcal{B}_{cbv}$, $\mathcal{B}_{lazy}$, $\mathcal{B}_{eager}$, and $\mathcal{B}_{cbs}$ be the pomset

[3]This is not to say that lazy functional programs are difficult to debug, but rather that traditional tracing techniques are perhaps inadequate for such debugging.

semantic functions for call-by-value, lazy evaluation, eager evaluation, and call-by-speculation, respectively. Now note that the following relationships hold: $\mathcal{B}_{cbs} \sqsubseteq \mathcal{B}_{lazy}$, and $\mathcal{B}_{cbs} \sqsubseteq \mathcal{B}_{eager} \sqsubseteq \mathcal{B}_{cbv}$. We can display this graphically as:



where the weakest elements are at the bottom. Thus call-by-speculation is the most general of our operational semantics in that it induces the weakest ordering on expression evaluation. Call-by-need adds constraints that effectively *delay* the *demand* for expression evaluation, and call-by-value and eager evaluation add constraints that effectively *promote* the *return* from expression evaluation.

5.2 Implementation Semantics

Regardless of whether or not we choose one operational semantics as "standard," we can at least use pomset semantics as a tool to describe the semantics of a particular *implementation*. In that regard, we make the following observations.

First, machines are finite. The operational semantics that we have defined, however, is potentially unbounded in its consumption of resources. Thus all that we should require of an actual machine M when it executes a (possibly annotated) functional program P is that it observe the constraints of the program's "standard" pomset; in other words, it is free to *augment* that pomset due to resource limitations.

We can formalize this in the following way: Let p be program P's standard pomset, and p/M be the pomset that represents P's execution on machine M. Then p/M must be in the augment closure of p; i.e. $p/M \in \alpha(p)$. If M is a sequential machine, we can say more: it must be that $p/M \in \lambda(p)$ (recall that $\lambda(p)$ is the set of linear augments of p). Finally, a notion of *local linearization*, $\underline{\lambda}(p)$, was defined by Pratt on pomsets whose events are pairs – it is the set of augments in which the subsets of pairs with the same "location" are totally ordered. Thus if p is a pomset interpretation with mapping semantics, and M is a multiprocessor, then we require that $p/M \in \underline{\lambda}(p)$.

Using this approach, care must be taken in defining the semantics of speculative computation. In particular, do we allow the implementation to "delete irrelevant tasks"? That is, to preempt the execution of speculatively invoked computations whose results are later determined not to be needed? Formally, if we are to give the implementation the liberty to do this, then instead of using the speculative expression e's normal pomset p in the semantics, we should use its prefix closure $\pi(p)$. This allows the implementation to choose an arbitrary prefix of p as its realization of the evaluation of e.

5.3 Designing a Meta-Language

Recall that the standard normal-order semantics of PFL is given by $\mathcal{E}_n$. Suppose now that we extend PFL with a class of annotations $\mathcal{A}$, and that $\mathcal{A}_n$ is the new normal-order semantics. A natural goal in designing the extended language is to prove the following sort of *consistency theorem*:

Theorem: Let p be any annotated PFL program, and p' be the same as p but with all annotations removed. Then $\mathcal{A}_n[\![p]\!] = \mathcal{E}_n[\![p']\!]$.

In reality, some annotations may alter the standard semantics, although usually in minor ways. In all uses of annotations that we have encountered, the change usually involves *termination properties.* More specifically, adding annotations may sometimes cause a terminating program to diverge or "deadlock." In such cases it is desireable to at least state formally the conditions under which such non-termination can be avoided.

6 Future Work

We have concentrated in this paper primarily on the use of pomsets to *define* operational semantics – more work is needed on their use in *controlling* operational behavior. This will require experience with a variety of applications to determine the degree of expressiveness needed in the source language (among the more interesting applications is control of non-determinism). Although the generality of a single mechanism is attractive, there are certainly classes of behaviors that would be better expressed using special syntax designed for them. For example, previously proposed syntaxes for controlling execution order and mapping program to machine can be thought of as "macros" that expand into pomset expressions. A future task for us is to identify those patterns of use that are most common, and design a practical language with syntax that captures those patterns.

Conversely, despite our attempt at generality, there are at least two other operational behaviors that we can think of that we cannot handle in our current framework: the *prioritization* of eagerly computed tasks, and the explicit *preemption*, or suspension, of an expression's evaluation. We feel that these can easily be described using pomsets once suitable "handles" to the appropriate actions are provided. Note, for example, that a priority relation is in essence a partial order.

Another concern for us, of course, is implementation issues. At Yale a virtual parallel graph reducer called *Alfalfa* is currently being implemented on two commercial multiprocessors: an Intel iPSC hypercube and an Encore shared-memory machine [3]. Our plan is for this system to eventually support both implicit (dynamic) and explicit (annotated) task allocation and scheduling.

Finally, our use of pomsets as a vehicle for expressing operational behavior was pragmatically motivated, and there are still some troublesome technical issues that we have not resolved. For example, a non-terminating program generates an infinite pomset – is this mathematically sound? Also, in the semantics of fix we defined a recursive pomset abstraction, yet did not define an ordering on the domain of pomsets – does a unique fixpoint exist? Indeed, the need for a domain of pomset abstractions is troublesome in itself, suggesting the possibility of "higher-order pomsets." We have made some progress in this area, but more work remains.

7 Acknowledgements

We wish to thank Los Alamos National Laboratory and MCC for their generous support of the Santa Fe Graph Reduction Workshop held in October, 1986; a special thanks is extended to Joe Fasel and Bob Keller for their organizational efforts. At the workshop it was pointed out that no completely general method had been devised for controlling the evaluation order of a functional program – subsequent "back-of-the-matchbook" discussions with John Hughes sowed the seeds of this research. Also thanks to Jonathan Young for commenting on an earlier draft of

this manuscript, and to the rest of the Wrestling Team at Yale for helping debug many of our ideas.

References

[1] F.W. Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Trans. on Prog. Lang. and Sys.*, 6(2), April 1984.

[2] B. Goldberg. Detecting sharing of partial applications in functional programs. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*, page to appear, Springer Verlag LNCS ..., September 1987.

[3] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, Department of Computer Science, expected Spring 1987.

[4] P. Hudak. Denotational semantics of a para-functional programming language. *Int'l Journal of Parallel Programming*, 15(2):103–125, 1986.

[5] P. Hudak. Para-functional programming. *Computer*, 19(8):60–71, August 1986.

[6] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. In *Proceedings of 1985 Int'l Conf. on Parallel Proc.*, pages 831–839, August 1985. Also appeared in IEEE Trans. on Computers, Vol C-34, No. 10, October 1985, pages 881-891.

[7] P. Hudak and B. Goldberg. Serial combinators: "optimal" grains of parallelism. In *Functional Programming Languages and Computer Architecture*, pages 382–388, Springer-Verlag LNCS 201, September 1985.

[8] P. Hudak and L. Smith. Para-functional programming: a paradigm for programming multiprocessor systems. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 243–254, January 1986.

[9] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 97–109, January 1986.

[10] R.J.M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proc. 1982 ACM Conf. on LISP and Functional Prog.*, pages 1–10, ACM, August 1982.

[11] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. *IEEE Computer*, 17(7):70–82, July 1984.

[12] R.M. Keller and G. Lindstrom. Approaching distributed database implementations through functional programming concepts. In *Int'l Conf. on Distributed Systems*, May 1985.

[13] V. Pratt. Modeling concurrency with partial orders. *Int'l Journal of Parallel Programming*, 15(1):33–72, February 1986.

[14] N.S. Sridharan. *Semi-applicative programming: an example.* Technical Report, BBN Laboratories, November 1985.

[15] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architectures. *Computing Surveys*, 14(1):93–143, March 1982.