XML Stream Processing Using a Lazy Concurrent Language

Shin-Cheng Mu and Ta-Chung Tsai
Institute of Information Science,
Academia Sinica, Taiwan
{scm,tachung}@iis.sinica.edu.tw

Keisuke Nakano
Department of Mathematical Informatics,
University of Tokyo, Japan
ksk@mist.i.u-tokyo.ac.jp

Abstract

Motivated by previous work on XML stream processing, we noticed that programmers need concurrency to save space, especially in a lazy language. User-controllable concurrency provides the possibility of reducing space usage in many programs. With lower garbage-collection overhead resulting from concurrent execution, the elapsed time of programs, stream processing ones in particular, is tremendously decreased.

The challenge is how to encapsulate concurrency without compromising expressiveness and flexibility of languages. We propose the idea of *pushing datatypes* — when a pushing closure is demanded, all expressions referring to it are evaluated concurrently to weak head normal forms. The closure is no more alive and may thus be garbage collected. Semantically, it is a non-intruding extension because it does not change the denotational semantics of an expression. It is also easy to be implemented on top of a language providing concurrent threads and inter-thread synchronisation. We have developed a prototype using Haskell and showed *pushing datatypes* can be used to effectively reduce space usage and thus result in shorter elapsed time in many programs.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

Keywords XML, Space Leak, Lazy Evaluation, Concurrency

1. Introduction

Functional programming languages that are suitable for XML processing, for example, XDuce [8] and CDuce [2], have successfully developed a growing user base. In the authors' opinion, the languages owe their success not only to their type systems designed for XML, but also to their expressiveness for general purpose computation. They allow programmers to implement an entire application for whatever purpose that happens to make use of XML data.

XML processing languages with tree-based APIs usually build the entire XML parse tree in memory before processing. On the other hand, stream-based APIs (e.g. SAX [3]), while enabling finer control of space usage, are more cumbersome to use. To achieve efficient heap usage in a tree-based language, Nakano and Mu [17] designed a stream-based abstract machine and showed how to compile a tree-based program to the model. On the other hand, Frisch and Nakano [5] described, using term rewriting, semantics for tree-based programs that process XML input in a stream-based manner. The downside of both approaches, however, is the loss of generality. Users may not like to switch completely to a specialised abstract machine or term-rewriting semantics merely because the input of a program happens to be XML.

Some seasoned functional programmers and referees raised their doubts on the authors' previous work [17]: rather than introducing a new computational model or new semantics, can we not let lazy evaluation do the job? Given a function $parse$ constructing an XML tree from the input stream, and a function $print$ flattening the tree back to a stream of tags, we simply evaluate $print \cdot f \cdot parse$, for a user-defined f, in a lazy manner. Is this not good enough?

In response to the question above, the authors' answer is: "No, unless we assume the presence of concurrency." As will be shown later, sequential execution of the above expression may introduce unexpected space leak, which can be plugged by concurrency. In fact, the authors would like to point out that the solutions of Nakano-Mu and Frisch-Nakano, though looking quite different from the outset, can be seen as different ways to introduce concurrency in their strict languages.

Insipired by the two previous work [5, 17], the authors suggest to extend user-controllable concurrency in non-strict functional languages. Programmers may apply concurrency to plug certain space leak for their specific programs. The question is how to encapsulate concurrency so that expressiveness and flexibility are not subject to certain abstract machine or semantics. We would like the extension to be a non-intruding one, that is, it does not impair referential transparency, and does not alter the denotational semantics. We do not need inter-process communication and non-determinism, but we do need some form of inter-thread synchronisation.

Our proposal is to associate concurrency with datatypes. We propose that the programmer is allowed to declare certain user-defined types as what we nickname *pushing datatypes*. The name is derived from the push parsing model of XML. When a closure of such a type is forced, all the bindings in the heap that refer to the closure are evaluated concurrently. The intention is that, after evaluation, the pushing closure may no longer be alive and be garbage-collected.

We have a prototype implementation, in a small language we nicknamed PUSH, to verify our claim about its memory usage.[1] The first prototype is implemented using Haskell as a convenient language for embedding. However, PUSH does *not* intend to propose new language features for Haskell. It is currently a small core language of its own, designed for XML stream processing while maintaining the ability for general-purpose programming, upon which we may gradually develop its own type system and optimisation strategies. Besides, the core implementation of the se-

Plan-X 9 January 2008, San Francisco, California.

[1] A prototype is available at: `http://www.iis.sinica.edu.tw/~scm`.

mantic of pushing datatypes contains only a few primitive functions. They can be easily implemented as a library in any non-strict language providing concurrent threads and inter-thread synchronisation. Without PUSH, programmers can still benefit from the idea of pushing datatypes.

In Section 2 we look at some examples motivating pushing datatypes, before we formally give them an operational semantics and a denotational semantics in Section 3. The implementation and the experiments are shown in Section 4. In Section 5, we discuss related work and draw conclusions.

2. Pushing Datatypes

For presentation, we consider a very simplified version of XML. A stream of XML tags (events) is represented by the following datatype:

$$xs \in XStream \quad ::= \quad [\,] \mid \langle \texttt{a} \rangle xs \mid \langle / \rangle xs.$$

The symbol $[\,]$ denotes end of stream, $\langle \texttt{a} \rangle xs$ a stream beginning with a start-tag with label `a`, and $\langle / \rangle xs$ a stream beginning with an end-tag, where we assume that the input is well-formed and omit the label. PCDATA is also omitted and represented by $\langle x \rangle \langle / \rangle$ for some string x.

2.1 The "Space Leak"

Assume that the input is an XML stream representing an article consisting of a sequence of paragraphs, marked up by tag `p` (we use typewriter font to denote string constants). Within the paragraphs, some segments may be marked up as keywords by the tag `key`: $\langle \texttt{p} \rangle \ldots \langle \texttt{key} \rangle \langle \texttt{1} \rangle \langle / \rangle \langle / \rangle \ldots \langle / \rangle \langle \texttt{p} \rangle \ldots \langle \texttt{key} \rangle \langle \texttt{2} \rangle \langle / \rangle \langle / \rangle \ldots \langle / \rangle$. The task is to copy the input to the output, but converting all the `key` tags to `em`. Furthermore, append in the end of the output an unordered list (`ul`) of keywords that appeared in the article, each marked up by `li`: $\langle \texttt{p} \rangle \ldots \langle \texttt{em} \rangle \langle \texttt{1} \rangle \langle / \rangle \langle / \rangle \ldots \langle / \rangle \langle \texttt{p} \rangle \ldots \langle \texttt{em} \rangle \langle \texttt{2} \rangle \langle / \rangle \langle / \rangle \ldots \langle / \rangle \langle \texttt{ul} \rangle \langle \texttt{li} \rangle \langle \texttt{1} \rangle \langle / \rangle \langle / \rangle \langle \texttt{li} \rangle \langle \texttt{2} \rangle \langle / \rangle \langle / \rangle \langle / \rangle$.

Figure 1 shows one of the possible ways the program may be written. Forests of rose trees are represented by *Forest* and *PForest*. They are isomorphic and we will see why we need two datatypes for forests later. The symbol $[\,]$ is overloaded to represent empty forests. The forest $\lceil a \rceil ts : us$ is built with a ternary constructor where a is the root label of the first tree, ts its children, and us the rest of the trees. Some definitions are omitted. The function *copy* injects a *PForest* into a *Forest*, and $(+\!\!+)$, denoting concatenation, is overloaded for forests and streams. The function *fst* extracts the first component of a pair.

Now assume that the output of *main* is consumed one by one, and the program is evaluated by a lazy evaluator. The function *print* demands output from *article* $[\,]$ *ws* where *ws* is a closure. The expression expands to: $\lceil \texttt{p} \rceil key2Em\ ts : \ article\ (allKeys\ ts)\ us$, for some ts and us. The next sub-tree demanded is $key2Em\ ts$, resulting in $\lceil \texttt{p} \rceil (\lceil a \rceil key2Em\ ts_1 : key2Em\ us_1) : article(allKeys\ ts)\ us$. The next part demanded is $key2Em\ ts_1$. The problem is, due to the order of demands from *print*, only the left-most subtree gets expanded. Meanwhile we still have a pointer to ts in $allKeys\ ts$. The entire parse tree resides in memory until successive calls to $key2Em$ have finished traversing the entire article and the subtrees under `ul` are finally demanded.

It does not help to tuple $key2Em$ and $allKeys$ into one function $trav\ ts = (key2Em\ ts, allKeys\ ts)$ that produces the result in a single pass. Consider a long paragraph with one keyword at the very end. Were the second component (for $allKeys$) demanded first, it has to force the evaluation of ts to the end of the paragraph to produce the first result. For the same reason it does not help to use `seq` to force a closure – the evaluation of closures should happen concurrently.

$$
\begin{array}{l}
vs, ws \in Forest \ ::= \ [\,] \mid \lceil a \rceil vs : ws \\
ts, us \in PForest \ ::= \ [\,] \mid \lceil a \rceil ts :: us \\
\\
parse :: XStream \to (PForest, XStream) \\
parse\ [\,] \ = \ ([\,], [\,]) \\
parse\ (\langle a \rangle xs) \ = \ \textbf{let}\ (vs, ys) = parse\ xs \\
\qquad\qquad\qquad\qquad\quad (ws, zs) = parse\ ys \\
\qquad\qquad\quad \textbf{in}\ (\lceil a \rceil vs :: ws, zs) \\
parse\ (\langle / \rangle xs) \ = \ ([\,], xs) \\
\\
key2Em :: PForest \to Forest \\
key2Em\ [\,] \ = \ [\,] \\
key2Em\ (\lceil \texttt{key} \rceil ts :: us) = \lceil \texttt{em} \rceil copy\ ts : key2Em\ us \\
key2Em\ (\lceil a \rceil ts :: us) = \lceil a \rceil key2Em\ ts : key2Em\ us \\
\\
allKeys :: PForest \to Forest \\
allKeys\ [\,] \ = \ [\,] \\
allKeys\ (\lceil \texttt{key} \rceil ts :: us) \ = \ \lceil \texttt{li} \rceil copy\ ts : allKeys\ us \\
allKeys\ (\lceil a \rceil ts :: us) \ = allKeys\ ts +\!\!+ allKeys\ us \\
\\
article :: Forest \to PForest \to Forest \\
article\ ks\ [\,] \ = \ \lceil \texttt{ul} \rceil ks : [\,] \\
article\ ks\ (\lceil \texttt{p} \rceil ts :: us) = \\
\quad \lceil \texttt{p} \rceil key2Em\ ts : \ article\ (ks +\!\!+ allKeys\ ts)\ us \\
\\
print :: Forest \to XStream \\
print\ [\,] \ = \ [\,] \\
print\ (\lceil a \rceil vs : ws) \ = \langle a \rangle print\ vs +\!\!+ \langle / \rangle print\ ws \\
\\
main :: XStream \to XStream \\
main = print \cdot article\ [\,] \cdot fst \cdot parse
\end{array}
$$

Figure 1. Example: article processing.

The situation echoes the observation of Hughes [9] that lazy sequential evaluators are bound to have a certain class of space leaks. To plug the space leak, Wadler [27] and Sparud [25] proposed introducing concurrency in the low-level implementation. They dealt with a "system level" problem that functions returning multiple results introduce space leak. Our problem, which is different from theirs, occurs at the "user level." One may argue that the behaviour *is* what is expected of lazy evaluation and should not be called a space leak — we should not evaluate $allKeys$, because we do not yet know whether it is eventually needed. However, when the price to pay for saving the computation is too huge, the programmer is willing to perform some computation that may be wasted in return for a smaller heap residency and, due to less garbage collection, shorter elapsed time. The situation happens particularly often in stream-based XML processing that usually makes simple computations but processes large input data. We propose an intiutive mechanism, pushing datatypes, for programmers to adjust the trade-off.

2.2 Introducing Pushing Datatypes

The abstraction we propose in PUSH is to allow programmers to declare a datatype T as a *pushing* datatype by a single declaration: $pushing\ T$. Closures of applications $f\ p$, $g\ p$, where p has a pushing type, are initially inactive. If one of them triggers the evaluation of p, the latter is pushed to all the closures, which start to evaluate concurrently until they reach weak head normal forms (*whnf*). By declaring *PForest* as a pushing datatype, the previous example works as we have wished: when $key2Em\ ts$ demands ts, $allKeys\ ts$ starts evaluation as well, thus the first constructor of ts can immediately be freed.

Currently, we assume that all pushing datatypes are first-order and acyclic. Apart from *pushing* declarations, programming in PUSH is no different from programming in an ordinary functional language. There is no need, for example, to program in IO monad. The program in Figure 2 reads an input tree, and replaces every

```
doc [] = []
doc (⌈japanese⌉ts: us) = case (katana ts) of
    T → ⌈japanese⌉(doc ts): (doc us)
    F → ⌈samurai⌉(doc ts): (doc us)
doc (⌈a⌉ts: us) = ⌈a⌉doc ts: doc us

katana [] = F
katana (⌈katana⌉ts: us) = T
katana (⌈a⌉ts: us) = case (katana ts) of
    T → T
    F → katana us
```

Figure 2. Example: *samurai*.

```
xs ∈ PList ::= [] | Int:: xs
pushing PList

copy [] = copy ys
copy (a:: xs) = a: copy xs

[] ++ ys = copy ys
(a:: xs) ++ ys = a: (xs ++ ys)

len n [] = n
len n (a:: xs) = len (n + 1) xs

join xs ys = (xs ++ ys, len 0 xs, len 0 ys)
```

Figure 3. Example: list concatenation.

node `japanese` by `samurai` if it has a child labelled `katana`.[2] In this example we take full advantage of lazy evaluation. The tree is traversed as if it is fully loaded into memory. When being executed, tags in the input stream are read, parsed, and printed immediately, until we see a start-tag `japanese`. The data after the tag is cached until we either see a `katana` tag, or reach the end-tag corresponding to `japanese`.

For another example making use of pushing datatypes, see Figure 3. The function *join* concatenates two lists as well as counting the numbers of items processed in each list. Since *PList* is declared to be a pushing datatype, the calls to *len* is activated as soon as the lists are demanded in turns by $(+\!\!+)$ and may therefore be freed early.

3. Semantics

The formulation of our semantics is strongly influenced by Launchbury [13] and Baker-Finch et al [1].

For simplicity, we assume in this paper that the source language is type checked by a monomorphic type system. This is not a restriction, however, since we can always specialise polymorphic functions according to their usage.

Before the types are thrown away, we remember which variables have pushing types and distinguish them by a predicate π. The source language is then transformed to a core language:

$$\begin{array}{lll} x, y \in \mathit{Var}, & c \in \mathit{Constructor}, & \\ e \in \mathit{Exp} & ::= & c\, x_1 \ldots x_n \mid x \mid \lambda x.e \mid e\, x \\ & \mid & \mathbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathbf{in}\ e \\ & \mid & \mathbf{case}\ e\ \mathbf{of}\ \{c_i\, y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n}. \end{array}$$

The syntax shows that functions can only be applied to variables. Launchbury also required that all variables be renamed to distinct names. These restrictions significantly simplify the semantics.

We impose some more restrictions. Applications to pushing variables may only happen in **let**-bindings. In every sequence of applications there can be at most one pushing variable, which must be the argument of the outer-most application. For example, the expression **case** $f\ p$ **of** ..., where $\pi(p) = \mathit{true}$, has to be lifted to **let** $g = f\ p$ **in case** g **of** .... The expression $(f\ p)\ x$, a valid expression in Launchbury's core language, has to be split into **let** $g = f\ p$ **in** $g\ x$. Figure 4 shows how *article* and *key2Em* look like in the core language, with pushing variables underlined.

[2] Under the Japanese feudal system, only a *samurai* was supposed to carry a *katana* (a Japanese sword).

```
article = λks.λp.case p of
  [] → ⌈ul⌉ks: []
  ⌈p⌉ts:: us → let ems = key2Em ts
                   keys = allKeys ts
                   ks' = ks ++ keys
                   axs = article ks' us
               in ⌈p⌉ems: axs
key2Em = λq.case q of
  [] → []
  ⌈key⌉vs:: ws → let vs' = copy vs
                     ws' = key2Em ws
                 in ⌈em⌉vs': ws'
  ⌈a⌉vs:: ws → let vs' = key2Em vs
                   ws' = key2Em ws
               in ⌈a⌉vs': ws'
```

Figure 4. Some example functions translated to the core language.

A *whnf* is either a λ abstraction or a constructor application $c\ e_1\ \ldots\ e_n$. In the next section, when using the identifier v we imply that it is in *whnf*.

3.1 Modelling Lazy Evaluation

In this section we define a relation $(\stackrel{1}{\Rightarrow})$ modelling a step of reduction in the operational semantics.

We model a *heap* as a mapping from a variable to an expression and a *status*. A status is one of A (active; currently running), B (blocked; waiting for some resources to be ready), and I (inactive; either not activated yet, or finished evaluation and in *whnf*). We write $\{H, x \stackrel{\alpha}{\mapsto} e\}$ to denote a heap H extended with a mapping from x to expression e with status α. The initial heap, denoted by *Init*, consists of a collection of bindings, all inactive, and a special binding for *main*, the only binding that is active. Evaluation stops when *main* is in *whnf*.

The relation $(\stackrel{a}{\longrightarrow})$ non-deterministically chooses an active binding from a heap:

$$\{H, z \stackrel{\mathsf{A}}{\mapsto} e\} \quad \stackrel{a}{\longrightarrow} \quad H : z \stackrel{\mathsf{A}}{\mapsto} e.$$

The notation $H : z \stackrel{\mathsf{A}}{\mapsto} e$ denotes that the binding $z \stackrel{\mathsf{A}}{\mapsto} e$ is the chosen one, while H is the rest of the heap apart from z.

The relation $(\stackrel{1}{\longrightarrow})$ is inductively defined by the rules in Figure 5. The left-hand side denotes the heap before a small step, while the right-hand side and the symbol K denote *modifications* to the heap. A **let** expression allocates more bindings into the heap. The newly allocated bindings are initially inactive. The notation $\overline{x} = \overline{e}$ denotes vectors of bindings, introduced for brevity. Assume z points to another variable x. If the value of x is in *whnf*, we just copy the value after renaming — $\hat{v}$ denotes α-conversion of **let** and **case** bound variables in v [13]. If x is not in *whnf* yet, we block z and activate the evaluation of x instead. In an application, if the functional part is already a λ abstraction, we simply perform β reduction. Otherwise we attempt to reduce the function. The rules for **case** are similar to those for application.

Finally, if an active variable z evaluates to *whnf*, we can reactivate the threads blocked on z before making z inactive. The notation $\overline{x} \stackrel{\mathsf{B}}{\mapsto} \overline{e^z}$ denotes *all* the bindings blocked on z, and e^z denotes an expression blocked on z. A blocked binding could only be the result of rules VAR2, APP2, and CASE2. In other words it

$$H : z \overset{\mathsf{A}}{\mapsto} \mathbf{let}\ \overline{x} = \overline{e}\ \mathbf{in}\ f \quad \overset{1}{\longrightarrow} \quad \overline{x} \overset{\mathsf{I}}{\mapsto} \overline{e}, z \overset{\mathsf{A}}{\mapsto} f \qquad \textsc{Let}$$

$$\{H, x \overset{\mathsf{I}}{\mapsto} v\} : z \overset{\mathsf{A}}{\mapsto} x \quad \overset{1}{\longrightarrow} \quad z \overset{\mathsf{A}}{\mapsto} \hat{v} \qquad \textsc{Var1}$$

$$\{H, x \overset{\mathsf{I}}{\mapsto} e\} : z \overset{\mathsf{A}}{\mapsto} x \quad \overset{1}{\longrightarrow} \quad x \overset{\mathsf{A}}{\mapsto} e, z \overset{\mathsf{B}}{\mapsto} x, \quad \text{if } e \text{ not } \textit{whnf} \qquad \textsc{Var2}$$

$$H : z \overset{\mathsf{A}}{\mapsto} (\lambda y.e)\ x \quad \overset{1}{\longrightarrow} \quad z \overset{\mathsf{A}}{\mapsto} e[x/y] \qquad \textsc{App1}$$

$$\frac{H : z \overset{\mathsf{A}}{\mapsto} e \quad \overset{1}{\longrightarrow} \quad z \overset{\alpha}{\mapsto} e', K}{H : z \overset{\mathsf{A}}{\mapsto} e\ x \quad \overset{1}{\longrightarrow} \quad z \overset{\alpha}{\mapsto} e'\ x, K} \qquad \textsc{App2}$$

$$H : z \overset{\mathsf{A}}{\mapsto} \mathbf{case}\ c_k\ \overline{x_k}\ \mathbf{of}\{c_i\ \overline{y_i} \to e_i\} \quad \overset{1}{\longrightarrow} \quad z \overset{\mathsf{A}}{\mapsto} e_k[\overline{x_k}/\overline{y_k}] \qquad \textsc{Case1}$$

$$\frac{H : z \overset{\mathsf{A}}{\mapsto} e \quad \overset{1}{\longrightarrow} \quad z \overset{\alpha}{\mapsto} e', K}{H : z \overset{\mathsf{A}}{\mapsto} \mathbf{case}\ e\ \mathbf{of}\{c_i\ \overline{y_i} \to e_i\} \overset{1}{\longrightarrow} z \overset{\alpha}{\mapsto} \mathbf{case}\ e'\ \mathbf{of}\{c_i\ \overline{y_i} \to e_i\}, K} \qquad \textsc{Case2}$$

$$\{H, \overline{x} \overset{\mathsf{B}}{\mapsto} \overline{e^z}\} : z \overset{\mathsf{A}}{\mapsto} v \quad \overset{1}{\longrightarrow} \quad \overline{x} \overset{\mathsf{A}}{\mapsto} \overline{e^z}, z \overset{\mathsf{I}}{\mapsto} v \qquad \textsc{UnBlock1}$$

Figure 5. Small step transitions for a single thread. Note that v is in *whnf*.

$$\begin{array}{lcl} \mathit{Nat} & ::= & \mathit{Z} \mid \mathit{S}\ \mathit{Nat} \\ \mathit{Bool} & ::= & \mathit{True} \mid \mathit{False} \\ \mathit{isZero} & = & \lambda n.\ \mathbf{case}\ n\ \mathbf{of}\ \mathit{Z} \to \mathit{True} \\ & & \qquad\qquad \mathit{S}\ k \to \mathit{False} \\ \mathit{plus2} & = & \lambda m.\mathbf{let}\ k = \mathit{S}\ m\ \mathbf{in}\ \mathit{S}\ k \\ \mathit{bot} & = & \mathit{bot} \\ \mathit{main} & = & \mathbf{let}\ x = \mathit{plus2}\ \mathit{bot} \\ & & \mathbf{in}\ \mathit{isZero}\ x \end{array}$$

Figure 6. A small sample program.

must be of the following form:

$$e^z \ ::= \ z \mid e^z\ x \mid \mathbf{case}\ e^z\ \mathbf{of}\ \{c_i\ \overline{y_i} \to e_i\}.$$

Note that a binding can only be blocked on one variable at a time.

One step of reduction consists of choosing an active binding, performing a single transition accordingly, and merging the modified bindings back:

$$H \overset{1}{\Rightarrow} H'' \ \equiv \ (H \overset{a}{\longrightarrow} H' : b) \ \wedge \ (H' : b \overset{1}{\longrightarrow} K) \ \wedge \ (H'' = H[K]).$$

The reflexive, transitive closure of $(\overset{1}{\Rightarrow})$ is denoted by $(\overset{1}{\Rightarrow}^*)$. The notation $H[K]$ denotes heap H modified by K, defined by:

$$H[K] \quad = \quad \{x \overset{\alpha}{\mapsto} e \in H \mid x \notin \mathit{dom}(K)\} \cup K.$$

Although we have been using a concurrent notation so far, we have merely modelled sequential lazy evaluation. Indeed, $(\overset{1}{\longrightarrow})$ is deterministic. The following property can be easily proved by induction.

LEMMA 3.1. Assume $\mathit{Init} \overset{1}{\Rightarrow}^* H$, then (1) there is exactly one active binding in H, and (2) the blocked bindings in H form a chain from main to the only active binding: $\mathit{main} \overset{\mathsf{B}}{\mapsto} e^{z_1}, z_1 \overset{\mathsf{B}}{\mapsto} e^{z_2} \ldots z_n \overset{\mathsf{A}}{\mapsto} e$.

Therefore, $(\overset{a}{\longrightarrow})$ has only one active binding to choose. With the introduction of pushing datatypes in the next section, we may have more than one active bindings in the heap. For a variable z in the heap, if $z \overset{\mathsf{B}}{\mapsto} e^{z_1}, z_1 \overset{\mathsf{B}}{\mapsto} e^{z_2} \ldots z_n \overset{\mathsf{A}}{\mapsto} e$, then z_n is called the *primary binding* of z.

$$\begin{array}{ll} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} \mathbf{let} \ldots \mathbf{in}\ \mathit{isZero}\ x,\ \ldots\} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} \mathit{isZero}\ x,\ x \overset{\mathsf{I}}{\mapsto} \mathit{plus2}\ \mathit{bot},\ \ldots\} \qquad \textsc{Let} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} (\lambda n.\mathbf{case}\ n\ \mathbf{of} \ldots)\ x,\ x \overset{\mathsf{I}}{\mapsto} \mathit{plus2}\ \mathit{bot},\ \ldots\} \qquad \textsc{App2} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} \mathbf{case}\ x\ \mathbf{of}\ \ldots,\ x \overset{\mathsf{I}}{\mapsto} \mathit{plus2}\ \mathit{bot},\ \ldots\} \qquad \textsc{App1} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{B}}{\mapsto} \mathbf{case}\ x\ \mathbf{of}\ \ldots,\ x \overset{\mathsf{A}}{\mapsto} \mathit{plus2}\ \mathit{bot},\ \ldots\} \qquad \textsc{Case2, Var2} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{B}}{\mapsto} \ldots,\ x \overset{\mathsf{A}}{\mapsto} (\lambda m.\mathbf{let}\ k_1 = S\ m\ \mathbf{in}\ S\ k_1)\ \mathit{bot},\ \ldots\} \qquad \textsc{App2} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{B}}{\mapsto} \mathbf{case}\ x\ \mathbf{of}\ \ldots,\ x \overset{\mathsf{A}}{\mapsto} \mathbf{let}\ k_1 = S\ \mathit{bot}\ \mathbf{in}\ S\ k_1,\ \ldots\} \qquad \textsc{App1} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{B}}{\mapsto} \mathbf{case}\ x\ \mathbf{of}\ \ldots,\ x \overset{\mathsf{A}}{\mapsto} S\ k_1,\ k_1 \overset{\mathsf{I}}{\mapsto} S\ \mathit{bot},\ \ldots\} \qquad \textsc{Let} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} \mathbf{case}\ x\ \mathbf{of}\ \ldots,\ x \overset{\mathsf{I}}{\mapsto} S\ k_1,\ k_1 \overset{\mathsf{I}}{\mapsto} S\ \mathit{bot},\ \ldots\} \qquad \textsc{UnBlock1} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} \mathbf{case}\ (S\ k_1)\ \mathbf{of}\ \ldots,\ x \overset{\mathsf{I}}{\mapsto} S\ k_1,\ k_1 \overset{\mathsf{I}}{\mapsto} S\ \mathit{bot},\ \ldots\} \qquad \textsc{Case2, Var1} \\ \overset{1}{\Rightarrow} & \{\mathit{main} \overset{\mathsf{A}}{\mapsto} \mathit{False},\ x \overset{\mathsf{I}}{\mapsto} S\ k_1,\ k_1 \overset{\mathsf{I}}{\mapsto} S\ \mathit{bot},\ \ldots\} \qquad \textsc{Case1} \end{array}$$

Figure 7. Tracing the program in Figure 6.

$$\{H, x \overset{\mathsf{AB}}{\mapsto} e\} : z \overset{\mathsf{A}}{\mapsto} x \quad \longrightarrow \quad z \overset{\mathsf{B}}{\mapsto} x \qquad \textsc{Var3}$$

$$\{H, \overline{x} \overset{\mathsf{B}}{\mapsto} \overline{e^z}, \overline{y} \overset{\mathsf{I}}{\mapsto} \overline{f\ z}\} : z \overset{\mathsf{A}}{\mapsto} v \quad \longrightarrow \quad \overline{x} \overset{\mathsf{A}}{\mapsto} \overline{e^z}, \overline{y} \overset{\mathsf{A}}{\mapsto} \overline{f\ z}, z \overset{\mathsf{I}}{\mapsto} v, \quad \text{if } \pi(z) \qquad \textsc{UnBlock2}$$

Figure 8. Extra rules for pushing datatypes. The notation AB means "either A or B".

Figure 6 shows a small sample program. Its trace, assuming an initial heap with bindings for main, isZero, $\mathit{plus2}$, and bot, is given in Figure 7.

3.2 Adding Pushing Datatypes

It takes only a slight modification to add pushing datatypes to the semantics. The relation $(\longrightarrow)$ is defined inductively by

- all the rules in Figure 5, with an extra side condition $\mathit{not}(\pi(z))$ for UNBLOCK1, and
- the rules in Figure 8.

The rule VAR3 is added to cover the cases missed by VAR1 and VAR2. In UNBLOCK2, z is the pushing closure that is forced. Besides activating all threads blocked for z, it also activates all *inactive* threads that refer to z. Once all the z are pattern matched, the cell occupied by z is no longer alive and can be garbage-collected. The reduction relation with pushing datatypes is then defined by:

$$H \Rightarrow H'' \ \equiv \ (H \overset{a}{\longrightarrow} H' : b) \ \wedge \ (H' : b \longrightarrow K) \ \wedge \ (H'' = H[K]).$$

3.3 Remarks on Using Pushing Datatypes

It is important to remember that forcing a pushing closure merely triggers evaluation of all *allocated* closures referring to it. It does not guarantee that the pushing closure can be garbage-collected

immediately. Consider the code fragment:

$$\begin{array}{l}\textbf{let } h' = h\ \underline{p}\\ \textbf{in case } h' \textbf{ of } c_1 \to \textbf{let } x_1 = f_1\ \underline{p} \textbf{ in } e_1\\ \qquad\qquad c_2 \to \textbf{let } x_2 = f_2\ \underline{p} \textbf{ in } e_2,\end{array}$$

where p is pushing and $h\ p$ demands p. The closures x_1 and x_2 are not allocated yet, and there is nothing to activate. To garbage-collect p earlier, the programmer has to lift the definitions as in:

$$\begin{array}{l}\textbf{let } h' = h\ \underline{p};\ x_1 = f_1\ \underline{p};\ x_2 = f_2\ \underline{p}\\ \textbf{in case } h' \textbf{ of } \{c_1 \to e_1; c_2 \to e_2\}.\end{array}$$

But that means both x_1 and x_2 will be evaluated when $\underline{p}$ is demanded, even though we will need only one of them. The situation is similar to prefetching instruction and data in a pipeline machine when the branching is uncertain. The authors have encountered a number of situations where it is desirable to perform some more computation in return for a smaller heap residency. Due to less garbage collection, the elapsed time may turn out to be shorter.

The same situation occurs in the following code fragment:

$$\begin{array}{l}\textbf{let } x = f\ \underline{p};\\ \quad z = \lambda y.\textbf{let } r = g\ \underline{p} \textbf{ in } \ldots r \ldots\\ \textbf{in } e.\end{array}$$

The closure r cannot be evaluated when x is demanded because it is not created yet. To compute closure r earlier, it has to be lifted out of z. To do such **let**-lifting, however, the programmer may have to perform some abstraction to the body of g if it depends on y. This is in contrast to the approach of Frisch and Nakano [5], where all occurrences of the specific input variable is globally recorded, and once part of the input is available, partial evaluation is carried out inside-out as much as possible. The overhead of their approach is that they have to maintain back-links on the syntax tree.

Assume that we swap the two parameters of $article$:

$$article \quad = \quad \lambda\underline{p}.\lambda ks.\textbf{case } \underline{p} \textbf{ of} \ldots.$$

The concurrent execution triggered by pushing datatypes immediately returns a *whnf*. To allow more eager computation, the programmer has to perform a **case**-lifting:

$$\begin{array}{rcl}article & = & \lambda\underline{p}.\textbf{case } \underline{p} \textbf{ of}\\ & & \qquad [\,] \to \lambda ks.\ldots\\ & & \qquad \lceil \mathtt{p} \rceil \underline{ts} :: \underline{us} \to \lambda ks.\ldots.\end{array}$$

Comparing to previous approaches to streaming, our current position is to choose a simpler evaluation strategy, the pushing datatypes, that is easy to implement, and leave rooms for optimisation by program transformation. Compilers for functional languages routinely have to perform **let** and **case**-lifting for optimisations targeting at various purposes. We will also have to developing optimisation strategies for PUSH in the presence of pushing datatypes, which will be part of our future work.

The programmer is advised to be cautious of the situation when a pushing data is enclosed in another data structure. Consider the following code fragment:

$$\begin{array}{l}\textbf{let } q = C\ \underline{p};\ x = f_1\ \underline{p};\ y = f_2\ q\\ \textbf{in } e,\end{array}$$

where C is a non-pushing data constructor containing a pushing closure p, while q is a non-pushing closure. Even though $\underline{p}$ is demanded by $f\ \underline{p}$, the closure $C\ \underline{p}$ is it self a *whnf* retaining a pointer to $\underline{p}$. This also forbids $\underline{p}$ from being collected.

3.4 Denotational Semantics

We would like to be sure that $(\Rightarrow)$ is a "safe" extension of the language, in the sense that as long as $(\overset{1}{\Rightarrow})$ terminates properly, there is a way for $(\Rightarrow)$ to terminate, and their results map to the same denotation. Luckily, very similar work has been done by Baker-Finch et al. [1] and can be adopted to our setting.

$$\begin{array}{rcl}\rho \in Env = Var \to Val & & \\ [\![\_]\!] & \in & Exp \to Env \to Val\\ [\![x]\!]_\rho & = & \rho(x)\\ [\![c\ \overline{x}]\!]_\rho & = & c\ [\![\overline{x}]\!]_\rho\\ [\![\lambda x.e]\!]_\rho & = & lift\ (\lambda\epsilon.[\![e]\!]_{\rho[x\mapsto\epsilon]})\\ [\![e\ x]\!]_\rho & = & drop([\![e]\!]_\rho)([\![x]\!]_\rho)\\ [\![let\ \overline{x} = \overline{e}\ in\ f]\!]_\rho & = & [\![f]\!]_{\{\!\{\overline{x}\mapsto\overline{e}\}\!\}_\rho}\\ [\![\textbf{case } e \textbf{ of } \{c_i\ \overline{x_i} \to e_i\}_{i=1}^n]\!]_\rho = & & \\ \quad [\![e_k]\!]_{\rho[\overline{x_k}\mapsto\overline{y_k}]}, \quad \text{if } [\![e]\!]_\rho = c_k\ \overline{y_k} & & \end{array}$$

$$\begin{array}{l}\{\!\{\_\}\!\} :: Heap \to Env \to Env\\ \{\!\{\overline{x} \mapsto \overline{e}\}\!\}_\rho = \mu\rho'.\rho[\overline{x} \mapsto [\![\overline{e}]\!]_{\rho'}]\end{array}$$

Figure 9. Denotational semantics. The notation $\rho[x \mapsto \epsilon]$ represents ρ modified by the binding $x \mapsto \epsilon$, while $\mu\rho.e$ denotes least fixed-point.

The denotational semantics of our language is given in Figure 9, where $[\![\_]\!]$ and $\{\!\{\_\}\!\}$ give the semantics of an expression and a heap with respect to an environment respectively. We assume that Val is a domain containing denotations for constructors and a lifted version of its own function space. The lifting operator and its inverse are denoted by *lift* and *drop*. The readers are referred to Launchbury [13] for details. The ordering on environments $(\leq)$ is defined by:

$$\rho \leq \rho' \quad \equiv \quad \forall x.(\rho(x) \neq \bot \Rightarrow \rho(x) = \rho'(x)).$$

The initial environment ρ_0 is defined by $\rho_0(x) = \bot$ for all x.

THEOREM 3.2. If $H \Rightarrow H'$, then $\{\!\{H\}\!\}_\rho \leq \{\!\{H'\}\!\}_\rho$ for all ρ. The same result holds for $(\overset{1}{\Rightarrow})$.

Proof. Induction on the size of H and on the structure of expressions.

The following theorem says that if a program terminates both with and without pushing datatypes, the reduction chains produce the same value with respect to the denotational semantics.

THEOREM 3.3. If $Init \overset{1}{\Rightarrow}^* \{H, main \overset{\mathsf{A}}{\mapsto} v,\}$ and $Init \Rightarrow^* \{H', main \overset{\mathsf{A}}{\mapsto} v',\}$, then $[\![v]\!]_{\{\!\{H\}\!\}_{\rho_0}} = [\![v']\!]_{\{\!\{H'\}\!\}_{\rho_0}}$.

Proof. By Theorem 3.2 we have $\{\!\{Init\}\!\}_{\rho_0} \leq \{\!\{H, main \overset{\mathsf{A}}{\mapsto} v\}\!\}_{\rho_0}$. In particular, $[\![v]\!]_{\{\!\{H\}\!\}_{\rho_0}} = [\![e]\!]_{\{\!\{Init\}\!\}_{\rho_0}}$. Similarly $[\![v']\!]_{\{\!\{H'\}\!\}_{\rho_0}} = [\![e]\!]_{\{\!\{Init\}\!\}_{\rho_0}}$.

The following theorems are adopted from Baker-Finch [1].

THEOREM 3.4. If $\{H, z \overset{\mathsf{A}}{\mapsto} e\} \Rightarrow^* \{H', z \overset{\mathsf{A}}{\mapsto} v\}$ then $[\![e]\!]_{\{\!\{H\}\!\}_\rho} \neq \bot$. The same result holds for $(\overset{1}{\Rightarrow})$.

THEOREM 3.5. If $[\![e]\!]_{\{\!\{H\}\!\}_\rho} \neq \bot$, there exists H', z, v such that $\{H, z \overset{\mathsf{A}}{\mapsto} e\} \Rightarrow^* \{H', z \overset{\mathsf{A}}{\mapsto} v\}$ in a finite number of steps.

Does a program that terminates under lazy evaluation also terminates if some types are declared as pushing? Certainly, if the former terminates, we can always construct a terminating reduction sequence of the latter by a copycat strategy, although this is not a very interesting sequence. For a more flexible strategy, the following theorem, provable by an inductive proof, allows the scheduler to reduce the active binding on which $main$ is blocked whenever it wants.

THEOREM 3.6. Suppose $\{H, z \overset{A}{\mapsto} e\} \Rightarrow^* \{H', z \overset{A}{\mapsto} v\}$ in n steps by always reducing the primary binding for z. Also suppose that $\{H, z \overset{A}{\mapsto} e\} = \{G, x \overset{A}{\mapsto} e'\}$, where $z \neq x$, and $G : x \overset{A}{\mapsto} e' \longrightarrow K$. Then $G[K] \Rightarrow^* \{G', z \overset{A}{\mapsto} v'\}$ in no more than n steps for some G' and v' by always reducing the primary binding of z.

Therefore, it is always safe for the scheduler to digress, execute some other threads every once in a while, and come back to the primary binding of *main* later. The computation terminates as long as the scheduler executes the primary binding of *main* frequently enough.

3.5 Exceptions

We have not modelled exceptions in our semantics. Firstly, we have to have a denotational semantics having exceptions. A well-known challenge of modelling exceptions in a non-strict language is that the exception an expression raises may depend on its context. One may resort to the "imprecise exception" approach by Peyton Jones et al. [23], in which an expression does not raise one exception, but a set of possible exceptions.

Secondly, to preserve the set of possible exceptions, we need a slightly sophisticated operational semantics, in which only the exception raised by the primary binding is actually thrown. Other exceptions are suspended until demanded. We can then prove that the exceptions a program may raise under $(\overset{!}{\Rightarrow})$ equals that under $(\Rightarrow)$. The details, however, is out of the scope of this paper.

4. Implementation

Currently, our prototype implementation reads a PUSH program, desugar it to the core language, and compiles it to a Concurrent Haskell [22] module. The reason for choosing Haskell, apart from that it is the language the authors are the most familiar with, is that it is high-level enough to demonstrate how PUSH can be implemented in a language providing threads and basic inter-thread communication. An additional benefit is that the compiled Haskell module can be called from a Haskell program that performs other computation. However, we do not intend to propose new language features to Haskell, nor tie PUSH with Haskell.

The core Haskell embedding, which completely implements the semantic of pushing datatypes, only amounts to around 60 lines of code. The small size of core embedding means our method can be easily implemented as a library on top of any non-strict language providing threads and basic inter-thread communication. No modification of compilers is required. Programmers can still benefit from the idea of pushing datatypes by using their preferred languages without adopting PUSH.

4.1 The Concurrent Haskell Embedding

Parts of the embedding are shown in Figure 10. A closure resulting in a value of type a is represented by a pair `(t,r)` of type `Var a`, where `t` is an *activation token*, while `r` stores the computed result. An expression yielding type `a` is simply embedded as `Expr a = IO a`. The function `decl`, in the do-notation `x <- decl e`, declares an initially inactive *let*-binding $x \overset{!}{\mapsto} e$. Initially, both `t` and `r` are empty, with a pending thread blocked on `t` at the call `readMV t`, a variation of the standard `readMVar` which avoids blocking. If a token is put into `t`, it performs the computation given by `e`, and stores the result in `r`. To activate a binding, we put a token into `t` and read the result from `r`, as is done in `var`.

In this prototype implementation, each binding is implemented as a thread. The number of threads could be large, but for the kind of programs we are tackling, the overhead of thread management is still insignificant comparing to the time spent on garbage collection. In the long run, we may apply some of the techniques proposed by Traub [26] to identify and compile sequential segments of programs.

```
type Var a = (MVar (), MVar a)
type Expr a = IO a

var :: Var a -> Expr a
var (t,r) = do _ <- tryPutMVar t ()
               readMV r

decl :: Expr a -> IO (Var a)
decl e = do t <- newEmptyMVar
            r <- newEmptyMVar
            _ <- forkIO (do _ <- readMV t
                            v <- e
                            putMVar r v)
            return (t,r)

dapp :: (Var a -> Expr b, Var a) -> IO (Var b)
dapp (f,p) = do t <- newEmptyMVar
                r <- newEmptyMVar
                i <- forkIO (do _ <- readMV (snd p)
                                _ <- tryPutMVar t ()
                                return ())
                _ <- forkIO (do _ <- readMV t
                                killThread i
                                v <- f p
                                putMVar r v)
                return (t, r)

readMV a =  block (do {
   x <- takeMVar a;
   b <- catch (unblock (return x));
         (\e -> do { tryPutMVar a x; throw e });
   tryPutMvar a b;
   return b })
```

Figure 10. A Haskell embedding.

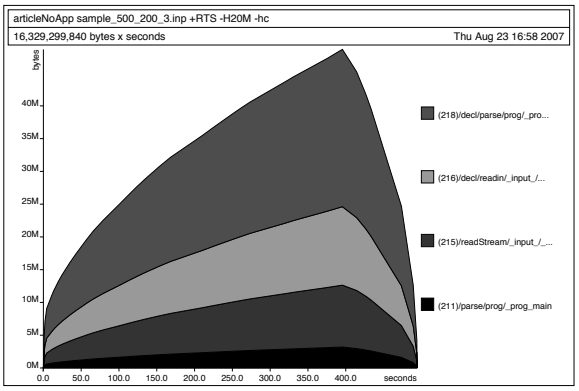
Datatypes in PUSH are embedded as Haskell datatypes, with pointers to sub-structures represented as `Var`s. For example, *Forest* is embedded as:

```
data Forest = Empty
       | Node (Var Label) (Var Forest) (Var Forest)
```

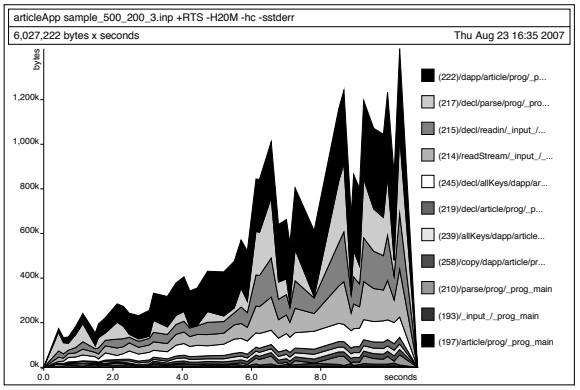
Pushing datatypes are represented in the same way. However, an inactive binding $x \overset{!}{\mapsto} f\ p$ where p is pushing is embedded as `x <- dapp (f,p)`. In the definition of `dapp`, we spark off two threads. The first one is blocked on the result field of `p`, while the second one, like in `decl`, blocks on `t`, waiting for `t` to be filled before computing `f p`. If the closure x is activated normally, the first thread will be killed by the second before the computation of `f p` to free up the space it occupies. If `p` is computed by some other thread before the evaluation of x, however, the first thread triggers the evaluation of x by putting a token into `t`.

PUSH programs can be embedded in Haskell. For example, the function *article* is embedded as below:

```
article :: Var Forest -> Var PForest -> Expr Forest
article ks p =
 do v <- var p
    case v of
     PEmpty -> do empty <- decl (return Empty)
                  ul <- decl (return "ul")
                  return (Node ul ks empty)
     PNode a ts us -> do ems <- dapp (key2Em, ts)
                         keys <- dapp (allKeys, ts)
                         ks' <- decl (catF ks keys)
                         axs <- dapp (article ks', us)
                         return (Node a ems axs)
```



(a) Pushing datatypes off. Y-axis scale: 50 MB.



(b) Pushing datatypes on. Y-axis scale: 1400 KB.

Figure 11. Heap Profiles: 500 paragraphs, 200 words, 3 keywords.

The reader may compare the embedding with its definition in the core language in Figure 4. Now that we know that pushing datatypes does not alter the denotational semantics, we can safely use `unsafePerformIO` to invoke the embedding in a pure context. The following function `listify`, for example, converts an embedded `List` to a Haskell list:

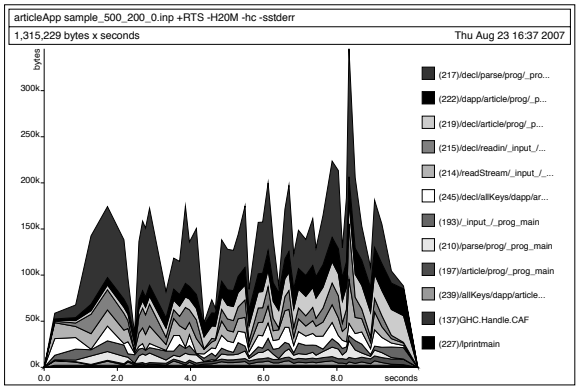
```
listify x = case unsafePerformIO (var x) of
  LNil -> []
  LCons a y -> (unsafePerformIO (var a)) : (listify y)
```

4.2 Benchmarking Results

For benchmarking, we try the `article` example on a sample input file consisting of 500 paragraphs. Each paragraph consists of 200 elements. Only the last 3 elements are keywords, all of them placed at the end of the paragraphs.

We prepared two versions of the embedding, one with pushing datatype turned off, the other with it enabled. We call the former program `LE` and the latter `PU`. The programs are compiled by GHC 6.6 with option `-O`, and run on a computer with a 2.16 GHz Intel Core 2 Duo processor. Figure 11(a) shows the heap profile of `LE`. The size of heap space occupied grows as the input is read. Most of the space is taken by closures generated by `parse`, indicating that the parse tree is not freed at all.

With pushing datatypes turned on, `PU` gives us the profile shown in Figure 11(b). The heap residency is much smaller, with a peak of about 1.4 MB, as opposed to more than 45 MB occupied by `LE`. The closures generated by `parse` no longer occupies the most space. The total space we need still grows linearly with time, because the list of keywords grows as the input is read. For comparison, Figure 12 shows the heap profile when `PU` is run on an input with



Y-axis scale: 350 KB.

Figure 12. Pushing datatypes on, 500 paragraphs, 200 words, 0 keywords.

no keywords at all. The heap profiles for this input appears more random. The top band in the graph varies in each run, but is usually one of `parse`, `dapp`, or `decl`.

If we exclude garbage collection time and consider mutation time only, `PU` is a bit slower than `LE`. For the input with 500 paragraphs, 200 words each, and 3 keywords, `PU` needs about 10.8 seconds of mutation time, while `LE` takes around 9.87 seconds. However, with an initial heap of 20 MB, `PU` spends 3.42 seconds on garbage collection, while `LE` needs 504.82 seconds, ending up spending 97.6 percent of the elapse time on garbage collecting. If we set the initial heap to 200 MB, on the other hand, `PU` spends 2.91 seconds on garbage collection, and `LE` spends an acceptable 8.71 seconds. At this heap size, `PU` is still quicker if we measure the elapsed time. Such results may vary a lot with different input sizes, heap sizes, and various parameters of the garbage collector.

5. Conclusion, Future Work, and Related Work

Can we process XML streams with tree-based programs in a space-efficient manner? While studying this problem, the authors noticed that we need concurrency to achieve reasonable space efficiency, which reminded us of the use of concurrency to plug space leaks in lazy languages [27, 25]. It is a common phenomenon: programmers need concurrency to save space, especially in a lazy language.

While the initial result looks encouraging, for more complex data flow we may need a synchronising mechanism to control the speed the threads consume the data. It remains to see how this can be done while retaining the simplicity of our language.

Many researchers talk about "XML streaming" while referring to slightly different concepts. Our definition of "streaming" means to read a unit of the input, use it, and dispose of it as quickly as possible, regardless of the minimal total space needed or the number of traversals. A number of efforts were devoted to performing XML transformations specified as attribute grammars [24, 20] or folds [10] on streamed XML input. On the other hand, Kodama et al. [12] used a variation of linear type to ensure that a program traverses the input tree left-to-right once only. Plenty of efforts focused on streaming XML query languages. Ludäscher et al. [16] worked on automatic derivation of XML stream processors from a subset of XQuery, while Green et al. [6] and Gupta [7] explored similar idea in XPath. FluXQuery [11] is an XQuery engine that automatically rewrites XQueries to FluX queries, an internal event-based language, while optimizing buffer usage. The BEA/XQRL processor [4] is an XQuery engine with pipelined XML processing, with an optimizer that transforms queries to equivalent but more efficient expressions.

A number of approaches have been proposed to incorporate concurrency into functional languages. To cope with nondeterminacy, however, either referential transparency is lost, or concurrency is encapsulated in a separate layer or a monad. For achieving our goal, this is going further than necessary. The pushing datatype is a lightweight extension. Syntactically, it is a minimal addition. Semantically, having pushing datatypes does not change the denotational semantics of a function. Operationally, the programmer easily grasps an intuition what a pushing datatype does — all allocated applications to a pushing variable are triggered at the same time and evaluate concurrently to *whnf*. We have shown that it is easy to implement pushing datatypes on top of a language providing threads and inter-thread synchronisation.

Our semantics owes much to the natural semantics of lazy evaluation by Launchbury [13], and the small-step operational semantics for parallel lazy evaluation by Baker-Finch et al [1]. On the other hand, Traub [26] has proposed that functional languages be modelled and implemented as concurrent threads. He modelled the semantics in a general way, with lazy evaluation and *lenient evaluation* (all redexes are evaluated in parallel) as special cases.

The language pH [19] is a eagerly-evaluated variation of Haskell evolving from Id [18]. It has a layered semantics which consists of a functional core, a I-structure layer, and a M-structure layer. Programs in the latter two layers are not referentially transparent. The type system determines the layer of a program.

GPH [14] is an extension to Haskell with two primitives, `par` and `seq`, for parallel and sequential composition respectively. Its design goal is to achieve maximum speed on multiprocessors, while ours is to minimise space usage on a single processor. The *evaluate-and-die* strategy [21], however, is closely related to the semantics of Baker-Finch et al. Eden [15], also an extension of Haskell with parallelism, allows programs to execute in a distributed environment. Process creation in Eden is explicit, while communication is implicit. The computation within a process is lazy, while communication is eager. Denotationally, Eden preserves the semantics of its Haskell subset.

References

[1] C. Baker-Finch, D. J. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 162–173, 2000.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2003.

[3] D. Brownell. *SAX2*. O'Reilly, January 2002.

[4] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *The 29th International Conference on Very Large Data Bases*, pages 997–1008, 2003.

[5] A. Frisch and K. Nakano. Streaming XML transformation using term rewriting. In *Programming Language Technologies for XML (PLAN-X 2007)*, pages 2–13, January 2007.

[6] T. J. Green, A. K. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. In *ACM Transactions on Database Systems*, volume 29(4), pages 752–788, 2004.

[7] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2003.

[8] H. Hosoya and B. C. Pierce. XDuce: a typed XML processing language (preliminary report). In G. Vossen and D. Suciu, editors, *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, number 1997 in Lecture Notes in Computer Science, pages 226–244. Springer-Verlag, May 2000.

[9] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. D.Phil thesis, Oxford University, July 1983.

[10] O. Kiselyov. A better XML parser through functional programming. In *4th International Symposium on Practical Aspects of Declarative Languages*, LNCS 2257, pages 209–224, 2002.

[11] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *The 30th International Conference on Very Large Data Bases*, pages 228–239, 2004.

[12] K. Kodama, K. Suenaga, N. Kobayashi, and W.-N. Chin. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In W.-N. Chin, editor, *The 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*, LNCS 3302, pages 41–56, 2004.

[13] J. Launchbury. A natural semantics for lazy evaluation. In *The 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, 1993.

[14] H.-W. Loidl and P. W. Trinder. *A gentle introduction to GPH*. Heriot-Watt University, July 2001.

[15] R. Loogen, Y. Ortega-Mallén, and R. Peña Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[16] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *The 28th International Conference on Very Large Data Bases*, August 2002.

[17] K. Nakano and S.-C. Mu. A pushdown machine for recursive XML processing. In N. Kobayashi, editor, *The 4th Asian Symposium on Programming Languages and Systems (APLAS 2006)*, LNCS 4279, pages 340–356. Springer-Verlag, 2006.

[18] R. S. Nikhil. An overview of the parallel language Id. Technical report, Digital Equipment Corp., Cambridge Research Laboratory, September 1993.

[19] R. S. Nikhil, Arvind, J. Hicks, S. Aditya, J.-W. Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Massachusetts Institute of Technology, 1995.

[20] S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54(2-3):257–290, 2005.

[21] S. Peyton Jones, C. D. Clack, and J. Salkild. High-performance parallel graph reduction. In *Parallel Architectures and Languages Europe (PARLE'89)*, number 365 in Lecture Notes in Computer Science, pages 193–206. Springer-Verlag, 1989.

[22] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *The 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, January 1996.

[23] S. Peyton Jones, A. Reid, C. A. R. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'99)*, Atlanta, 1999.

[24] S. Scherzinger and A. Kemper. Syntax-directed transformations of XML streams. In *Workshop on Programming Language Technologies for XML*, pages 75–86, 2005.

[25] J. Sparud. Fixing some space leaks without a garbage collector. In R. J. M. Hughes, editor, *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 117–122, 9-11 June 1993.

[26] K. R. Traub. Compilation as partitioning: a new approach to compiling non-strict functional languages. In *The 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, 1989.

[27] P. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–608, 1987.