Compiling Strictness into Streams

by

Cordelia V. Hall and David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 209

Compiling Strictness into Streams

by

Cordelia V. Hall & David S. Wise

December, 1986

This material is based on work supported in part by National Science Foundation grant number DCR 84-05241.

Compiling Strictness into Streams

Cordelia V. Hall*
David S. Wise*

Indiana University
101 Lindley Hall
Bloomington, Indiana 47401-4101
U.S.A.

CR Categories and Subject Descriptors:
D.1.1 [Applicative (Functional) Programming]
D.3.4 [Processors] Compilers
F.3.2 [Semantics of Programming Languages] Operational semantics
General Term: Compilation
Additional Key Words and Phrases: strictness analysis; lazy, applicative, functional languages; abstract interpretation; lists; buffering.

Abstract

Lazy programs delay the evaluation of many expressions that will be required by the computation and can thus be evaluated early. Strictness analysis finds some of these expressions, but it has been limited to finite lists or flat domains so far. We discuss a lattice of *strictness patterns* that allows us to analyse first-order lazy programs using streams as well as lists. We then limit the set of patterns so that a compiler using them can be shown to terminate. The behavior and influence of the output device that drives the computation is shown to also drive the analysis, and is discussed in detail.

Introduction

Lazy (normal order) languages [5, 8, 20] defer evaluation of expressions until they are required. As a result, they admit a more powerful semantics allowing infinite list structures, like streams [15], and the construction of programs as mathematical expressions which have the same meaning regardless of evaluation order.

* Research reported herein was supported partially by NSF grant DCR 84-05241

Unfortunately, there is a significant operational expense — both space and time — associated with each delayed evaluation. Local bindings must be retained until every lazy reference to them has been resolved, requiring space to hold many environments. Extra context switching becomes a major expense over time. Also, every use of a potentially postponed evaluation requires a test to determine whether the evaluation has been performed [2]. In contrast, conventional languages use an applicative order (call-by-value) protocol which avoids these costs. Thus, most users have come to expect performance better than faithfully lazy implementations can provide. A compiler that detects unnecessary laziness would produce code that avoids much of this overhead, injecting call-by-value behavior into expressions and regaining the efficiency of conventional implementations wherever possible.

The earliest work on lazy evaluation recognizes that some functions are strict [21, 5]. We are particularly interested in constructing a compiler algorithm that detects the strictness of lists and streams, because lists are a vitally useful data structure, more tractable than unrestricted higher order functions, and flagged syntactically by the use of **cons** [5]. Our source language is a first-order subset of Daisy [14], an applicative language using a lazy **cons**, and our target is Daisy sugared with strictness annotations on any field within a list structure.

Our current solution to extracting strictness from structures has been developed from experience in designing circuits and interactive programming environments in a lazy language. Streams form the basic tool of this work. Whereas imperative programs frequently diverge after partial output produced by output statements, full strictness can prevent a functional program from producing any stream output at all. In such cases, the user may lose important information that would have been printed before an infinite loop occurred, as well as stream output in general. It is essential that we retain stream behavior in programs improved with strictness analysis and, as far as we know, no other work on strictness analysis has addressed the issue of output streams.

The next section is an overview of our solution. We then present an abstract compiler that is a simple version of the compiler we have implemented in the lazy language Daisy [14]. Safety and termination proofs follow, and we conclude with a discussion of other recent work on this problem, and current and future work to be done on our approach.

1. Overview of the method

One of the most challenging aspects of finding strictness in lazy lists is that expressions producing lists may appear in a variety of contexts. For example, the expression `(cons 1 [])` may be strict in its first argument, second argument, neither, or both depending upon whether the entire **program** being evaluated is `(head (cons 1 []))`, `(tail (cons 1 []))`, `(nil? (cons 1 []))`, or just `(cons 1 [])`. These different contexts are represented in our domain of *strictness patterns*, whose elements are list-like objects that specify whether call-by-need or call-by-name is to be applied to variously indexed fields within a structure. Strictness patterns have two roles. When the strictness pattern is determined from the enclosing evaluation context (such as the exhaustive evaluation of a program's result), it is an *inherited* pattern. Strictness patterns can also be created during the analysis of a function application. When `a` is bound using a λ-form, `a` may occur several times in the form's body and inherit several patterns, possibly pieces of the pattern inherited by the application. Some assemblage of these patterns forms a *synthesized* pattern, which becomes the pattern inherited by the argument. For example, if the entire program to be evaluated is `((lambda (a) (head (cons a [])))(cons 1 b))`, then the pattern synthesized by compilation of the lambda expression permits `(cons 1 b)` to be strict only in its first argument.

Applications of a function `f` become especially interesting because several calls to `f` may inherit several distinct strictness patterns; thus, the definition of `f` inherits each of them. We associate each of these distinct patterns with `f`'s definition, creating several new *versions* of `f`, each associated with one of the inherited patterns. A call to `f` is replaced by a call to the appropriate *version* of `f`.

A less powerful alternative to this approach is to treat `f`'s definition as inheriting some lower (i.e. less strict) bound of all those patterns inherited in the applications. Other proposals for strictness analysis follow this strategy. Because some strictness patterns represent only coarse approximations of uncomputable patterns, however, that lower bound is likely to be trivial, calling for lazy evaluation. In addition, patterns may complement each other, so that one may be strict where the other is not, causing the lower bound to be a poor approximation. Thus, less strictness would be introduced using this alternative. We accept growth in the number of versions, constrained by a compiler tuning parameter, a finite *resource*. Once the versions are identified, they can be merged when appropriate.

We have built a practical strictness compiler that always terminates. In the next section,

we introduce a complete lattice of strictness patterns that contains both finitely representable and infinite limit points. The set of useful strictness patterns is reduced to those patterns that can be represented by a finite graph, with or without cycles (cf. rationals, as opposed to irrational numbers). In addition, we give the compiler a resource bound on the number of steps it may take in refining a strictness pattern towards an optimal solution for strictifying a recursive, first-order function. This approach provides a tradeoff under which we accept possibly inferior approximations to computable limit points, in exchange for guaranteed termination of refinement toward non-finitely representable limit points.

2. Lattice of Strictness Patterns

All functions in our source and target language are regarded as taking one argument that may be destructured as necessary. For this reason, we will expand the definition of a strict function to include trees and specify an *index* for each part of the argument structure in which the function is strict. Consider the conventional labeling of the argument list as a binary tree with root labeled '1', right children labeled with '1' and left children labeled with '0'. The index of each element in this tree is the number represented by the concatenation of bit labels along the path from root to its location. The following expression displays the indexing of a tree:

$$1\langle\ 2\langle 4..., 5...\rangle,\ 3\langle 6..., 7...\rangle\rangle\rangle.$$

A list marked with \$ at any indexed sublist is to be evaluated by a suitably modified lazy interpreter using call-by-value for those fields. Strictness at a given index does not necessarily imply that a function is strict at any other index of its argument. For example, the evaluation of

```
$<$<b, c>, d>
```

is lazy in the the values of `b,c,d` but strict in the external structure of the pair `<b c>` and in the outermost pair (cf. Landin's stream construct [15]).

A pattern π is interpreted as follows;

- $\pi = \perp$ indicates no strictness, at all.
- $\pi = \$\overline{\pi}$ indicates strictness at Position 1 (tree's root), and any additional strictness implied by $\overline{\pi}$; *the absence of a leading dollar sign indicates no strictness at Position 1.*
- For $i > 1$, whether $\$\langle\pi_1, \pi_2\rangle$ or $\langle\pi_1, \pi_2\rangle$ is strict in Position i is determined recursively.

Let $\lfloor lg\ i \rfloor = k+1$; i can be expressed either as $i = 2n + j$ or as $i = 3n + j$ for integers $0 \leq j < 2^k = n$. It is strict in Position i if and only if π_1 in the former case, respectively π_2 in the latter case, is strict in Position $n + j$.

Let $\$\mathbf{P} = \{\$\pi | \pi \in \mathbf{P}\}$, let $+$ connote coalesced sum, and let us require all lifting to be explicit [19]. Then the domain $\mathbf{P}$ may be defined by the reflexive equation,

$$\pi : \mathbf{P} = \$\mathbf{P} + (\mathbf{P} \times \mathbf{P})_\perp$$

subject to the homomorphic collapse required by the following two rules:

RULE 1. $\pi \sqsubseteq \$\pi$.

Strictness strengthens a pattern.

RULE 2. $\$\$\pi \sqsubseteq \$\pi$.

PROPOSITION. $\$\pi = \$\$\pi$.

Strictness is idempotent.

LEMMA: fix $\lambda\pi.\$\langle\pi\ .\ \pi\rangle = \top_{\mathbf{P}}$.

PROOF: $\mathbf{P}$ is a c.p.o. from the construction. Let $\overline{\pi} = \text{fix}\ \lambda\pi.\$\langle\pi\ .\ \pi\rangle$. It suffices to show that for any isolated π, $\pi \sqsubseteq \overline{\pi}$. Then if each π_i is isolated, $\pi_i \sqsubseteq \pi_{i+1}$, we have $\bigsqcup_i\ \pi_i \sqsubseteq \overline{\pi}$, and thus , $\forall\ \pi\ (\pi \sqsubseteq \overline{\pi})$ and so $\overline{\pi} = \top_{\mathbf{P}}$.

Now if π is isolated, then π may be written in a finite expansion of dollar signs, angle brackets and $\perp_{\mathbf{P}}$. By adding \$ to every possible position in that expression, we get a π' where $\pi \sqsubseteq \pi'$ by Rule 1. Since $\perp \sqsubseteq \overline{\pi}$, by substituting $\overline{\pi}$ for every $\perp_{\mathbf{P}}$ in the finite expansion of π', we obtain π'', where $\pi \sqsubseteq \pi' \sqsubseteq \pi'' = \overline{\pi}$. ■

LEMMA: Any two elements of π have a least upper bound.

The proof is derived from the fact that two isolated π's have a least upper bound, again inferred from their necessarily finite expressions.

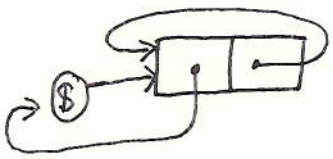
THEOREM 1. $\mathbf{P}$ is a complete lattice.

3. The compiler

An important element of $\mathbf{P}$ is the *printer* pattern

$$\pi_0 = \text{fix}\ \lambda\pi.\langle\$\pi\ .\ \pi\rangle \neq \top_{\mathbf{P}}.$$

which can be abstractly represented as the finite cyclic graph:



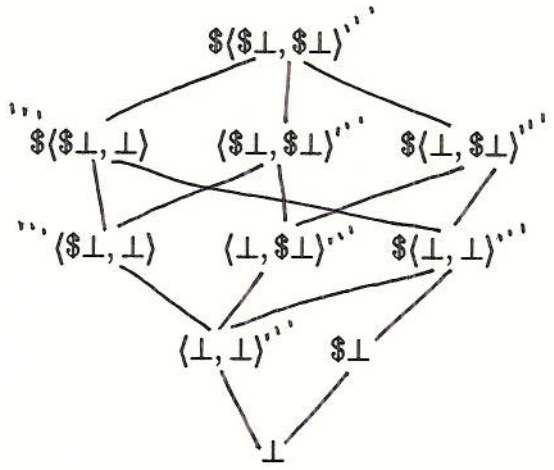


Figure 1. Partial lattice **P** of strictness patterns

Important in the operational semantics is that the meet, join, and equality of two such patterns, represented as finite cyclic graphs, can be finitely computed (derived similarly to taking the intersection of regular expressions.) In the equations that follow, we deal only with finitely representable elements from **P**, which form an (incomplete) sublattice.

The compilation of the program inherits the *printer* pattern, π_0, which is strict only in the heads of all streams and sub-streams. This strictness pattern implies a leftmost-outermost evaluation order. In this way, any stream produced by the compiled code will retain its lazy behavior.

The compilation of function versions has already been described. Identifiers bound in a *fix* expression will be treated similarly, as they may also represent recursively defined streams. However, identifiers bound in a *lambda* expression will not be converted into versions. Instead, they will form part of the mechanism for synthesizing patterns by passing on the *join* of all their inherited patterns. The *join* of synthesized patterns might produce a pattern higher in the sublattice than π_0 (see Rule 13 below), so we shall instead take the *meet* of any least upper bound with the printer pattern itself. This *meet* guarantees that π_0 is an upper bound for all patterns either inherited or synthesized during compilation.

The compiler builds up strictness information about identifiers and functions in a compile-time environment. It receives an integer resource that bounds the number of different versions

that can be created for any given function or identifier. Function invocations that can be recognized as references to extant versions consume no additional resources.

While our source language delays only the evaluation of **cons** arguments, the abstract compiler we present marks any expression that it can infer to be strict.

3.1 Restricted Daisy syntactic categories

id $\in$ *IDE* ; (identifiers)

e $\in$ *EXP* ; (expressions)

const $\in$ CONST. (constants)

3.2 Restricted Daisy Syntax

e ::= const | ⟨⟩ | arith-prim:⟨e . e⟩ | head:e | tail:e |
⟨e0 . e1⟩ | if:⟨e0 e1 e2⟩ | fix:[id e] |
(fix:[id0 λid1.e]):e | id:e | id

In the rest of the paper, \ stands for λ.

3.3 Restricted Daisy Value Domains and Semantic Functions

A; (atoms)

$S = A + (S \times S) + (S \longrightarrow S)$. (structures)

Rightfully, there should be a page of semantic equations here. This restricted Daisy is not a terribly startling lazy language, however, and a semantics is readily available elsewhere [11]. The only details in the syntax that may be distracting are that colon is an infix *apply* operator occuring between *function* and *argument*; that angle brackets construct lists, like Lisp's `list`, and arguments are very often list-shaped; and that square brackets isolate identifiers after `fix`, which would be easier read with an (understood) outermost λ and infix period wrapping the bracketed structure.

There is an irony in expressing strictness (to be introduced with dollar sign $) denotationally. Johnson [11] didn't need continuations to express laziness because it was implicit in his normal-order semantics. Operational issues, like strictness, cannot be expressed there. If we were to interpret denotational rules using *applicative* order, however, then laziness could only be expressed using continuations. Then any strictness implied by dollar signs could be expressed in a straightforward manner (without using continuations). So it seems that laziness, rather than strictness, is the difficulty requiring continuations. That is unfortunate because

laziness is so natural under normal order evaluation, where strictness cannot be expressed directly.

3.4 Compiler Domains

C: $D \longrightarrow D$;

(compiler)

$D = EXP \times \mathbf{P} \times ENV \times INT$;

(compilation data)

π : $\mathbf{P} = \$\mathbf{P} + (\mathbf{P} \times \mathbf{P})$;

(strictness patterns)

ρ : $ENV = V \longrightarrow (EXP + \mathbf{nb}) \times \mathbf{P} \times \mathbf{P} \times INT$;

(compiler environment)

ι : INT;

(resource)

ν : $V = ID \times \mathbf{P}$;

(version identifier)

The compile-time environment contains a tuple of binding, synthesized pattern, inherited pattern, and version count. **nb** indicates that a variable is unbound in the source code. A version identifier is represented in the following equations as **id** or **f** subscripted by **P**.

3.5 Compiler Semantic Functions

These equations describe an operational semantics for the abstract compiler. Examples appear after some of the equations. These examples present the code (the other parts of the tuple are omitted) produced by the compiler when it receives an expression, inherited pattern, environment and resource. The environment is assumed to be the initial environment, but the expression, inherited pattern and resource will be described. The pattern

$$\$\text{fix } \lambda\pi.\langle\$\pi \ . \ \pi\rangle$$

is the pattern to be initially propagated by the compiler, however our examples in both this and the next section use a variety of patterns.

We introduce the following notation: $\alpha{\cdot}\pi$ represents a strictness pattern, π, that may or may not be prefixed with \$. If \$ is the prefix, then $\alpha = \$$, otherwise α is the null string.

$\hat{\pi}$ refers to a synthesized pattern, $\check{\pi}$ to an inherited pattern, in analogy with propagation through the parse tree. Expressions surrounded by double brackets are syntactic expressions in the source language. Superscripts indicate new names defined locally within each rule. Unsubscripted $\perp$ stands for $\perp_{\mathrm{P}}$.

$$[\![\$\$\mathbf{e}]\!]\ \pi\ \rho\ \iota = [\![\$\mathbf{e}]\!]\ \pi\ \rho\ \iota. \tag{1}$$

Strictness is idempotent.

$$C\,[\![\$\mathbf{e}]\!]\ \pi\ \rho\ \iota = C\,[\![\mathbf{e}]\!]\ \$\pi\ \rho\ \iota. \tag{2}$$

Expressions marked strict by the programmer may be incorporated into the compilation process.

$$C\,[\![\mathbf{const}]\!]\ \pi\ \rho\ \iota = [\![\$\mathbf{const}]\!]\ \pi\ \rho\ \iota. \tag{3}$$

Constants are atomic, well-defined and thus safe for strict evaluation.

$$C\,[\![\mathbf{e}]\!]\ \pi\ \rho\ \iota = [\![\mathbf{e}]\!]\ \perp_{\mathrm{P}}\ \rho\ \iota \tag{4}$$

where \$ does not occur in **e** or π.

Unless strictness is inherited or specified by the programmer (cf. Rule 2), there is nothing to be compiled. We should note that this is done to keep these equations as simple as possible so that we can communicate our basic approach. It is clearly the case, as Lindstrom [16] observes, that it is safe to compile a function call as if the outer structure of the result is required, as long as the call itself is not marked. These equations only partially take advantage of this idea.

$$C\,[\![\mathbf{head}:\mathbf{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\alpha{\cdot}\mathbf{head}:\$\mathbf{e}^1]\!]\ \perp\ \rho^1\ \iota \tag{5}$$
$$\textbf{where}\ [\![\mathbf{e}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\mathbf{e}]\!]\ \alpha{\cdot}\langle\alpha{\cdot}\pi,\ \perp\rangle\ \rho\ \iota.$$

$$C\,[\![\mathbf{tail}:\mathbf{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\alpha{\cdot}\mathbf{tail}:\$\mathbf{e}^1]\!]\ \perp\ \rho^1\ \iota \tag{6}$$
$$\textbf{where}\ [\![\mathbf{e}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\mathbf{e}]\!]\ \alpha{\cdot}\langle\perp,\ \alpha{\cdot}\pi\rangle\ \rho\ \iota.$$

Patterns inherited by applications of **head** or **tail** are injected into a list pattern to eventually be inherited by a list. For example, if *exp* =

`head:<head:<1 . 2> . tail:<3 . 4>>`

then $C\,[\![exp]\!]$ \$fix $\lambda\pi.(\$\pi\ .\ \pi)\ \lambda\mathrm{id}_\pi.unbound\ 4 =$

`$head:$<$head:$<$1 . $2> . tail:$<$3 . $4>>` .

If $exp =$

`tail:<head:<1 . 2> . tail:<3 . 4>>`

then $C\,[\![exp]\!]$ \$fix $\lambda\pi.(\$\pi\ .\ \pi)\ \lambda\mathrm{id}_\pi.unbound\ 4 =$

`$tail:$<head:$<$1 . $2> . $tail:$<$3 . $4>>` .

$$C\,[\![\texttt{<e0 . e1>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\alpha{\cdot}\texttt{<e0}^1\texttt{. e1}^2\texttt{>}]\!]\ \bot\ \rho^2\ \iota \tag{7}$$

where $[\![\texttt{e0}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\texttt{e0}]\!]\ (\pi{\downarrow}1)\ \rho\ \iota;$

$[\![\texttt{e1}^2]\!]\ \pi^2\ \rho^2\ \iota^2 = C\,[\![\texttt{e1}]\!]\ (\pi{\downarrow}2)\ \rho^1\ \iota.$

The compilation of **cons** passes the head of its inherited pattern to the compilation of its first argument and then the tail of the inherited pattern to the compilation of its second argument. Preorder traversal is implied by an inherited pattern under π_0. For example, if $exp =$

`<head:<1 . 2> . tail:<3 . 4>>`

then $C\,[\![exp]\!]$ \$fix $\lambda\pi.(\$\pi\ .\ \pi)\ \lambda\mathrm{id}_\pi.unbound\ 4 =$

`$<$head:$<$1 . $2> . tail:$<$3 . $4>>` .

$$C\,[\![\texttt{arith-prim:<e0 . e1>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\alpha{\cdot}\texttt{arith-prim:e}^1]\!]\ \bot\ \rho^1\ \iota \tag{8}$$

where

$[\![\texttt{e}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\texttt{<e0 . e1>}]\!]\ \$(\$\bot\ ,\ \$\bot)\ \rho\ \iota.$

Arithmetic primitives are strict in both arguments. For example, if $exp =$

`add:<head:<1 . 2> . tail:<3 . 4>>`

then $C\,[\![exp]\!]$ \$$\bot$ $\lambda id_\pi.unbound\ 4 =$

`$add:$<$head:$<$1 . $2> . $tail:$<$3 . $4>>` .

$$C\,[\![\texttt{if:<e0 e1 e2>}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\alpha^3{\cdot}\texttt{if:\$<}\ \texttt{e0}^0\ \texttt{e1}^1\ \texttt{e2}^2\texttt{>}]\!]\ \bot\rho^3\iota \tag{9}$$

where

$[\![\mathbf{e0}^0]\!]\ \pi^0\ \rho^0\ \iota^0 = C\,[\![\mathbf{e0}]\!]\,\$\bot\ \rho\ \iota;$

$[\![\alpha^1{\cdot}\mathbf{e1}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\mathbf{e1}]\!]\,\alpha{\cdot}\pi\ \rho^0\ \iota;$

$[\![\alpha^2{\cdot}\mathbf{e2}^2]\!]\ \pi^2\ \rho^2\ \iota^2 = C\,[\![\mathbf{e2}]\!]\,\alpha{\cdot}\pi\ \rho^0\ \iota;$

$$\rho^3 = \lambda i_\pi.\begin{cases} \langle \mathbf{nb},\ \bot,\ (\rho^1\ i_\pi)\!\downarrow\!2\!\downarrow\!2\!\downarrow\!1 \sqcap (\rho^2\ i_\pi)\!\downarrow\!2\!\downarrow\!2\!\downarrow\!1,\ 0\rangle \\ \qquad \text{if } (\rho^1\ i_\pi)\!\downarrow\!1 = \mathbf{nb} \text{ and } (\rho^2\ i_\pi)\!\downarrow\!1 = \mathbf{nb}; \\ \langle \mathbf{nb}, \bot, \bot, 0\rangle \\ \qquad \text{if } (\rho^1\ i_\pi)\!\downarrow\!1 = \mathbf{nb} \text{ and } (\rho^2\ i_\pi) = \mathit{unbound} \\ \qquad\quad \text{or} \\ \qquad (\rho^1\ i_\pi) = \mathit{unbound} \text{ and } (\rho^2\ i_\pi)\!\downarrow\!1 = \mathbf{nb}; \\ \rho^1\ i_\pi \\ \qquad \text{if } (\rho^1\ i_\pi) \neq \mathit{unbound}; \\ \rho^2\ i_\pi \quad \text{otherwise;} \end{cases}$$

$$\alpha^3 = \begin{cases} \$, & \text{if } \alpha^1 = \$ = \alpha^2; \\ \mathit{nullstring}, & \text{otherwise.} \end{cases}$$

A conditional expression is strict in its predicate, but not in any of the paths of the predicate's result. Each branch of the **if** may safely be compiled using the pattern inherited by the **if** application as long as the leading $ is stripped off the compiled code when the new application is assembled and returned. This permits the predicate to be evaluated before either of the two branches, and allows the selected branch to be as efficient as possible. The new environment returns the *meet* of the patterns inherited from either branch by a variable bound in a lambda expression, or the appropriate entry. **if** receives special attention in [7], where a more powerful equation is developed and presented.

For example, if *exp* =

```
if:<same?:<head:<1 . 2> .
           tail:<3 . 4>>
      <head:<4 . 5> . 4>
      <2 . 3>
   >
```

then $C\,[\![exp]\!]\ \langle \$\bot\ .\ \$\bot\rangle\ \lambda id_\pi.unbound\ 1 =$

```
if:$<$same?:$<$head:$<$1 . $2> .
             $tail:$<$3 . $4>>
      <$head:$<$4 . $5> . $4>
      <$2 . $3>
   > .
```

$$C\,[\![\texttt{fix:[id e]}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = [\![\alpha{\cdot}\texttt{fix:[id}_{\alpha\cdot\pi}\ \texttt{e}^1\texttt{]}]\!]\ \bot\ \rho^3\ \iota \tag{10}$$

where

$$[\![\texttt{e}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\texttt{e}]\!]\ \alpha{\cdot}\pi\ \rho^2\ \iota;$$

$$\rho^2 = \lambda i_\pi.\begin{cases}\langle \texttt{e}, \bot, \alpha{\cdot}\pi, 0\rangle, \\ \qquad \text{if } i_\pi = \texttt{id}_{\alpha\cdot\pi}; \\ \langle \texttt{e}, \bot, \bot, 0\rangle, \\ \qquad \text{if } i_\pi = \texttt{id}_\bot \textbf{ and } \rho\ \texttt{id}_\bot = \textit{unbound}; \\ \rho\ i_\pi, \text{otherwise};\end{cases}$$

$$\rho^3 = \lambda i_\pi.\begin{cases}\rho\ i_\pi, & \text{if } i_\pi = \texttt{id}_{\alpha\cdot\pi} \textbf{ or } i_\pi = \texttt{id}_\bot; \\ \rho^1\ i_\pi, & \text{otherwise.}\end{cases}$$

For example, if $exp =$

```
fix:[l <1 . 1>]
```

then $C\,[\![exp]\!]\ \langle \$\bot\ .\ \langle \bot\ .\ \text{fix}\ \lambda\pi.\langle \$\bot\ .\ \pi\rangle\rangle\rangle\ \lambda\text{id}_\pi.\textit{unbound}\ 1 =$

```
fix:[l-p1
     <$1 .
       fix:[l-p2
         <$1 . fix:[l <1 . 1>]>]>] ,
```

and $C\,[\![exp]\!]\ \langle \$\bot\ .\ \langle \bot\ .\ \text{fix}\ \lambda\pi.\langle \$\bot\ .\ \pi\rangle\rangle\rangle\ \lambda\text{id}_\pi.\textit{unbound}\ 2 =$

```
fix:[l-p1
     <$1 .
       fix:[l-p2
         <$1 . fix:[l-p3 <$1 . l-p3>]>]>] ,
```

where

p1= $\langle \$\bot\ .\ \langle \bot\ .\ \text{fix}\ \lambda\pi.\langle \$\bot\ .\ \pi\rangle\rangle\rangle$;

p2= $\langle \bot\ .\ \text{fix}\ \lambda\pi.\langle \$\bot\ .\ \pi\rangle\rangle$;

p3= $\text{fix}\ \lambda\pi.\langle \$\bot\ .\ \pi\rangle$.

In the first example, the compiler can make only the first two elements of the output stream strict because the number of versions is too small to permit it to discover the recursive loop. When it is allowed one more version, it is able to make the entire stream strict in its heads. A resource of 3 or more would still produce the result from the second example.

$$C\,[\![\texttt{(fix:[f}\ \lambda\texttt{id.body])}\texttt{:e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \tag{11}$$
$$[\![\alpha{\cdot}\texttt{(fix:[f}_{\alpha\cdot\pi}\ \lambda\texttt{id.body}^1\texttt{])}\texttt{:\$e}^1]\!]\ \bot\ \rho^3\ \iota$$

where

$$[\![\mathbf{body}^1]\!]\ \pi^1\ \rho^1\ \iota^1 = C\,[\![\mathbf{body}]\!]\ \alpha{\cdot}\pi\ \rho^2\ \iota;$$

$$\rho^2 = \lambda i_\pi.\begin{cases} \langle \mathbf{nb}, \bot, \bot, 0\rangle \\ \qquad \text{if } i_\pi = \mathbf{id}_\bot; \\ \langle \lambda \mathbf{id}.\mathbf{body}, \pi^{rec}, \alpha{\cdot}\pi, 0\rangle \\ \qquad \text{if } i_\pi = \mathbf{f}_{\alpha\cdot\pi}; \\ \langle \lambda \mathbf{id}.\mathbf{body}, \bot, \bot, 0\rangle \\ \qquad \text{if } i_\pi = \mathbf{f}_\bot \text{ and } \rho\ \mathbf{f}_\bot = \mathit{unbound}; \\ \rho\ i_\pi, \text{otherwise}; \end{cases}$$

$$\pi^{rec} = (\rho^1\ \mathbf{id}_\bot)\lfloor 2\lfloor 2\lfloor 1;$$

$$[\![\mathbf{e}^1]\!]\ \pi^3\ \rho^3\ \iota^3 = C\,[\![\mathbf{e}]\!]\ \pi^{rec}\rho^4\iota;$$

$$\rho^4 = \lambda i_\pi.\begin{cases} \rho\ i_\pi, & \text{if } i_\pi = \mathbf{id}_\bot \mathbf{or}\ i_\pi = \mathbf{f}_{\alpha\cdot\pi} \mathbf{or}\ i_\pi = \mathbf{f}_\bot; \\ \rho^1\ i_\pi, & \text{otherwise.} \end{cases}$$

The compiler constructs a recursively defined synthesized pattern π^{rec} by fixing the result of the analysis of the λ body. This pattern is then inherited by the argument **e**. (Section 5 on the implementation of the compiler discusses the significant problem that arises when pattern bindings are not maintained explicitly by the compiler. Except for this, these equations provide an executable operational semantics. This point does not arise from equation (10) because no pattern is synthesized there.)

For example, if *exp* =

```
(fix:[l
      \lst.
       <add:<head:lst .
             head:tail:lst>
        . f:tail:lst>]):
 <head:<1 . 2> .
   tail:<3 . 4>>
```

then $C\,[\![exp]\!]$ fix $\lambda\pi.\langle\$\bot\ .\ \pi\rangle\ \lambda \mathrm{id}_\pi.\mathit{unbound}\ 1 =$

```
(fix:[f-p1
      \lst.
       <$add:$<$head:$lst .
               $head:$tail:$lst>
        . f-p1:$tail:$lst>]):
 $<$head:$<$1 . $2> .
   <$tail:$<$3 . $4>>>
```

where **p1**= fix $\lambda\pi.\langle\$\bot\ .\ \pi\rangle$.

$$C\,[\![\mathtt{f}:\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (12)$$

$$\begin{cases} \textbf{Reached-Limit} & \text{if } \rho\ \mathtt{f}_{\alpha\cdot\pi} = \textit{unbound} \textbf{ and } \textit{v-count} = \iota; \\ \textbf{Compile-Binding} & \text{if } \rho\ \mathtt{f}_{\alpha\cdot\pi} = \textit{unbound} \textbf{ and } \textit{v-count} < \iota; \\ \textbf{Retrieve-Syn-Pattern}, \text{otherwise}; & \end{cases}$$

$$\textbf{where} \langle [\![\mathtt{binding}]\!], \hat{\pi}, \check{\pi}, \textit{v-count} \rangle = \rho\ \mathtt{f}_{\perp};$$

Reached-Limit is $[\![\alpha{\cdot}(\mathtt{fix}:[\mathtt{f\ binding}]):\mathtt{e}]\!]\ \perp\ \rho\ \iota;$

Compile-Binding is

$$C\,[\![(\mathtt{fix}:[\mathtt{f\ binding}]):\mathtt{e}]\!]\ \alpha{\cdot}\pi\ \rho^1\ \iota$$

$$\textbf{where} \rho^1 = \lambda i_\pi . \begin{cases} \langle \mathtt{binding}, \hat{\pi}, \check{\pi}, \textit{v-count}+1 \rangle, & \text{if } i_\pi = \mathtt{f}_\perp; \\ \rho\ i_\pi, & \text{otherwise}; \end{cases}$$

Retrieve-Syn-Pattern is

$$[\![\alpha{\cdot}\mathtt{f}_{\alpha\cdot\pi}:\$\mathtt{e}^1]\!]\ \perp \rho^2\ \iota$$

$$\textbf{where } [\![\mathtt{e}^1]\!]\ \pi^2\ \rho^2\ \iota^2 = C\,[\![\mathtt{e}]\!]\ (\rho\ \mathtt{f}_{\alpha\cdot\pi})\downarrow 2 \downarrow 1\ \rho\ \iota.$$

If $\mathtt{f}_{\alpha\cdot\pi}$ does not refer to a version of **f** already compiled with $\alpha{\cdot}\pi$ as its inherited pattern, then there are two possibilities. Either the bound on versions of **f** forbids the creation of a new one, or a new one is created, raising **f**'s version count. If $\mathtt{f}_{\alpha\cdot\pi}$ does refer to a compiled version, then the function argument is compiled with the version's synthesized pattern. See (11) for an example.

$$C\,[\![\mathtt{id}]\!]\ \alpha{\cdot}\pi\ \rho\ \iota = \qquad (13)$$

$$\begin{cases} \textbf{Variable} & \text{if } \mathtt{binding} = \mathtt{nb}; \\ \textbf{Reached-Limit} & \text{if } \rho\ \mathtt{id}_{\alpha\cdot\pi} = \textit{unbound} \textbf{ and } \textit{v-count} = \iota; \\ \textbf{Compile-Binding} & \text{if} \rho\ \mathtt{id}_{\alpha\cdot\pi} = \textit{unbound} \textbf{ and } \textit{v-count} < \iota; \\ \textbf{Mark-With-Pattern}, \text{otherwise}; & \end{cases}$$

$$\textbf{where} \langle [\![\mathtt{binding}]\!], \hat{\pi}, \check{\pi}, \textit{v-count} \rangle = \rho\ \mathtt{id}_{\perp};$$

Variable is

$$[\![\alpha{\cdot}\mathtt{id}]\!]\ \perp\ \rho^1\ \iota$$

where

$$\rho^1 = \lambda i_\pi . \begin{cases} \langle \mathtt{binding}, \hat{\pi}, \\ \quad (\alpha{\cdot}\pi \sqcup \check{\pi}) \sqcap \$\mathrm{fix}\ \lambda\pi . \langle \$\pi\ .\ \pi \rangle, \textit{v-count} \rangle, \\ \qquad \text{if } i_\pi = \mathtt{id}_\perp; \\ \rho\ i_\pi, \text{otherwise}; \end{cases}$$

Reached-Limit is $[\![\alpha{\cdot}\texttt{fix:[id}_\perp\ \texttt{binding]}]\!]\ \perp\ \rho\ \iota$;

Compile-Binding is

$C\ [\![\texttt{fix:[id binding]}]\!]\ \alpha{\cdot}\pi\ \rho^1\ \iota$

where $\rho^1 = \lambda i_\pi.\ \begin{cases} \langle \texttt{binding}, \hat{\pi}, \check{\pi}, \textit{v-count}+1\rangle, & \text{if } i_\pi = \texttt{id}_\perp; \\ \rho\ i_\pi, \text{otherwise}; \end{cases}$

Mark-With-Pattern is $[\![\alpha{\cdot}\texttt{id}_{\alpha\cdot\pi}]\!]\ \perp\ \rho\ \iota$.

Identifiers may be recursively bound to a value, in which case they are treated similarly to recursive functions, or they may be bound in a lambda expression, in which case the pattern currently being inherited is combined with the previously inherited patterns. See (11) for an example.

4. Three Examples

The following is a simple and common program, in which a filter passes on certain elements of its argument stream.

```
(fix:[G
     \l. <head:l . G:tail:tail:l>]):
  (fix:[F
       \x. if:<eq?:<head:x . 1>
                  <head:x . F:<tail:x . head:x>>
                  <(fix:[Bad \y. Bad:y]):
                     head:x
                   . F:<tail:x . head:x>>>]):
    <1 . 0>
```

F produces a stream of alternating 1's and $\perp_S$. G selects odd elements of F's result, avoiding the divergent elements. The compiler produces the following compiled expression, given π_0, the initial environment $\lambda \text{id}_\pi.\textit{unbound}$, and the resource 5;

```
(fix:[G-printer
     \l.
      $<$head:$l .
        G-printer:$tail:$tail:$l>]):
  (fix:[F-p1
       \x.
       $if:$<$eq?:$<$head:$x . $1>
              <$head:$x .
                F-p2:$<$tail:$x .
                          head:$x>>
```

```
                <$(fix:[Bad-p4
                        \y. $Bad-p4:$y]):
                   $head:$x
                  . F-p2:$<$tail:$x .
                             head:$x>>>]):
       $<$1 . $0>
where
F-p2 =
fix:[F-p2
    \x.
     if:$<$eq?:$<$head:$x . $1>
           <head:x .
            F-p3:$<$tail:$x .
                      head:$x>>
           <(fix:[Bad
                 \y. Bad:y]):
              head:x
             . F-p3:$<$tail:$x .
                        head:$x>>>]
F-p3 =
(fix:[F-p3
     \x.
     if:$<$eq?:$<$head:$x . $1>
           <$head:$x .
             F-p2:$<$tail:$x .
                       head:$x>>
           <$(fix:[Bad-p4
                  \y. $Bad-p4:$y]):
              $head:$x
             . F-p2:$<$tail:$x .
                        head:$x>>>]):
    $<$1 . $0>
and where
```

$\mathtt{p1} = \$\mathrm{fix}\lambda\pi.\langle\$\pi_0 \,.\, \langle\bot \,.\, \pi\rangle\rangle$

$\mathtt{p2} = \mathrm{fix}\lambda\pi.\langle\bot \,.\, \langle\$\pi_0 \,.\, \pi\rangle\rangle$

$\mathtt{p3} = \mathrm{fix}\lambda\pi.\langle\$\pi_0 \,.\, \langle\bot \,.\, \pi\rangle\rangle$

$\mathtt{p4} = \$\pi_0$

The versions of F produce a stream that is alternately strict and lazy in its heads, and G is strict in all elements it accesses, but produces a stream strict in all heads and lazy in the tails. Notice that the first three patterns, those patterns which distinguish between versions

of **F**, have

$$\mathrm{fix}\lambda\pi.\langle\bot\ .\ \langle\bot\ .\ \pi\rangle\rangle$$

as their greatest lower bound. If we had decided not to produce different versions of **F** here, then we could have found no strictness in **F**. Versions **F-p1** and **F-p3** are similar and could be coalesced into one version inheriting the *meet* of the two patterns. **Bad-p4** produces a synthesized pattern that is $\bot$. The effect of this pattern would be obvious if **Bad-p4** was applied to an expression using list syntax, such as **<a . b>**.

The next two examples are hard to read using fix notation, so we will use *letrec* instead. The implemented compiler uses *letrec*, a recursive binding function that permits all versions to be gathered into the same scope, producing less object code. Formals are grouped together, as are actuals, rather than pairs of formals and actuals. We also introduce an *if* with more than two branches. Predicates following the first one may be marked, but since the mark is covered by the tail of the *if* argument, it won't be evaluated until the branch in which it appears is selected. Function formal arguments are now destructured into a flat list of bound variables, however the corresponding actuals are written as dotted pairs. The functions **odd?, number?, identifier?, Fn, Args, Body, Formals** and **nil?** are all assumed to produce the synthesized pattern **$⊥**. **eq?** and **same?** are treated as in equation (8).

The first is a function that prints the even Fibonnacci numbers, seeded with two values from anywhere in the series.

```
\[a b].
rec:[[h Addall Skip]
     <
h =
     <a . <b . Addall:<h . <tail:h>>>>
Addall =
     \[a b]. <add:<head:a . head:b> .
               Addall:<tail:a . <tail:b>>>
Skip =
     \[stream].
      if:<odd?:head:stream
           Skip:<tail:stream>
           <head:stream .
              Skip:<tail:stream>>
         >
     >
in
   Skip:<h>
```

```
]
```

When compiled with the pattern fix $\lambda\pi.\langle\$\bot\ .\ \pi\rangle$, and a resource of 4, our compiler produces the following output;

```
\[a b].
rec:[[h-p2 h-p3 Addall-p1 Skip-p1]
     <
h-p2 =
 $<$a . <$b .
    Addall-p1:$<$h-p2 . <$tail:$h-p3>>>>
h-p3 =
 $<a . $<$b .
    Addall-p1:$<$h-p2 . <$tail:$h-p3>>>>
Addall-p1 =
 \[a b].
  <$add:$<$head:$a . $head:$b> .
    Addall-p1:$<$tail:$a . <$tail:$b>>>
Skip-p1 =
 \[stream].
  if:$<$odd?:$head:$stream
        Skip-p1:$<$tail:$stream>
        <$head:$stream .
          Skip-p1:$<$tail:$stream>>>
     >
in
 Skip-p1:$<$h-p2>
 ]
```

p1 = fix $\lambda\pi.\langle\$\bot\ .\ \pi\rangle$

p2 = \$fix $\lambda\pi.\langle\$\bot\ .\ \pi\rangle$

p3 = $\$\langle\bot\ .\ \$\text{fix}\ \lambda\pi.\langle\$\bot\ .\ \pi\rangle\rangle$

Skip-p1 is strict in all the heads of its stream argument, and passes this pattern to the recursive data structure **h-p2**. **Addall-p1** inherits the same pattern. Note that two versions of **h** are created because one inherits the cyclic pattern passed on from **Addall-p1** while the other's inherited pattern is lazy in its head but inherits the cyclic pattern in the tail.

The second function compiled using *rec* is a simple interpreter. Its language is restricted to constants, identifiers, head, tail, cons, and functions of one argument.

Note that **cons** is used to collect the *rec* actuals. Outer strict marks on actuals are included here, but would normally be removed.

```
\[input].
```

```
rec:[[EVAL APPLY LOOKUP MKENV]
<
EVAL =
\[exp env].
 if:<number?:exp
      exp
     identifier?:exp
      LOOKUP:<exp . <env>>
      APPLY:<Fn:exp .
             <EVAL:<Args:exp . <env>> .
               <env>>>
    >
APPLY =
\[fn args env].
 if:<nil?:args
      [error]
     same?:<^head . fn>
      head:args
     same?:<^tail . fn>
      tail:args
     same?:<^cons . fn>
      <head:args . tail:args>
     identifier?:fn
      APPLY:<EVAL:<fn . <env>> .
              <args .
               <env>>>
      EVAL:<Body:fn .
             <MKENV:<Formals:fn .
                      <args .
                        <env>>>>>
    >
LOOKUP =
\[exp env].
 if:<same?:<exp . head:head:env>
      tail:head:env
      LOOKUP:<exp . <tail:env>>
    >
MKENV =
\[formals actuals env].
 if:<nil?:actuals
      env
      <<head:formals . head:actuals> .
        MKENV:<tail:formals .
                <tail:actuals env>>>
    >
 in
```

```
EVAL:<input . <[]>>
  ]
```

When the interpreter is compiled with the pattern $\perp$, and the resource 3, the following output is produced;

```
$\[input].
$rec:[[EVAL-p1 APPLY-p1 LOOKUP-p1 MKENV]
<
EVAL-p1 =
$\[exp env].
 $if:$<number?:$exp
        exp
       $identifier?:$exp
        LOOKUP-p1:$<$exp . <$env>>
        APPLY-p1:$<$Fn:$exp .
                  <$EVAL-p1:$<$Args:$exp .
                              <env>>
                    . <env>>>
    >
APPLY-p1 =
$\[fn args env].
 $if:$<$nil?:$args
        [error]
       $same?:$<$^head . $fn>
         head:$args
       $same?:$<$^tail . $fn>
         tail:$args
       $same?:$<$^cons . $fn>
         <head:args . tail:args>
       $identifier?:$fn
          APPLY-p1:$<$EVAL-p1:$<$fn .
                              <env>> .
                    <$args .
                     <env>>>
          EVAL-p1:$<$Body:$fn .
                   <MKENV:<Formals:fn .
                             <args .
                               <env>>>>>
     >
LOOKUP-p1 =
$\[exp env].
 $if:$<$same?:$<$exp . $head:$head:$env>
        tail:$head:$env
        LOOKUP-p1:$<$exp . <$tail:$env>>
     >
```

```
MKENV =
\[formals actuals env].
 if:<nil?:actuals
      env
      <<head:formals . head:actuals> .
        MKENV:<tail:formals .
               <tail:actuals env>>>
    >
>
 in
EVAL-p1:$<$input . <$[]>>
  ]
```

$\texttt{p1} = \$\bot$

EVAL-p1 is strict only in its first argument, and **APPLY-p1** is strict only in its first two arguments. **LOOKUP-p1** is strict in all of its arguments. **MKENV** is never reached by the propagation of a strict pattern, so it remains unchanged.

5.1 Representation of $\bot_P$

The equations shown here are not directly executable with conventional binding mechanisms. For example, if we were to analyse the expression

```
(fix:[f \n.f:n]):3
```

we would find within equation (11) that π_{rec} is bound to "the value of π_{rec}" and our compiler would loop indefinitely. Actually $\pi_{rec} = \bot_P$, but conventional binding machinery denies us the ability to detect this case. However, the compiler maintains a table of pattern bindings, permitting it to detect such a binding. Since only *rational patterns* arise here, every potential divergence manifests itself in such a cycle. Therefore, any binding that would diverge because of indirect self-dependence must cycle through some binding in the table. It is the *second* visit to such an entry in the table (of bounded size) which determines that the value of π_{rec} is $\bot_P$.

5.2 Version Reduction

The ideal strictness compiler would produce as many versions as needed, subject to a reasonable resource, but would then coalesce versions according to certain criteria. Versions that are identical, yet inherit different patterns, are really one version. Versions inheriting

patterns of comparable power should be coalesced, but the user should also have the ability to define classes of patterns which are to be regarded as comparable in power. This seems straightforward, although these techniques have not yet been implemented. More experience is needed with versions before the problem of combining them is fully understood.

6. Compiler safety and termination

Any implementation of a lazy stream language that ameliorates performance through an analysis of strictness *must* sacrifice some semantic strength when printing a list containing $\perp_S$. This fact is assumed in current work on the strictness of lists [16, 22]. The problem is that some element of a list may be $\perp_S$ and it might occur in a position that has been analysed as "strict." Thus, an enveloping portion of the list may "diverge" (*i.e.* evaluate to $\perp_S$), even though a truly lazy implementation would have no difficulty with the structure of the envelope. In the simplest case, Landin's $\langle \$\perp_S .\ b \rangle$, this divergence causes the printing operation to lose even the outer left parenthesis. A more complicated case, such as $F{:}\langle \perp_S\ 1 \rangle$, where $F = \lambda a\ b.\langle \$b\ .\ \langle \$a\ .\ \langle\rangle\rangle\rangle$, would cause the loss of the output prefix '⟨1'.

This problem is inherent in streams, but streams create yet another problem. In some special cases, the printer will not print some prefix of the output even though none of the elements are $\perp_S$. For example, a stream of natural numbers can be created by the following expression:

```
F:0 where F = \n. <$n ! F:inc:n>
```

It can also be created as follows:

```
F:0 where F = \n. <$F:inc:n ! n>
```

which, of course, has no printable prefix. Traversal of the stream of naturals constructed the second way, with recursion in the left of the resulting list, will cause the loss of the initial parentheses produced by traversal over a fully lazy expression. For these reasons, we define an *admissable* answer and show that for such results, the printer produces the same output as the traversal of a fully lazy expression.

We presume that any resulting stream structure, which would be fed to the printer/output device, must be "maximal" in the domain of streams. $\perp_S$ cannot occur anywhere within its result. This stipulation does not require that the result be isolated (e.g. the stream of ascending natural numbers is maximal, but not isolated.) Specifically, we *do* desire non-

isolated points to be admissable answers, but we need to exclude non-isolated values from some positions within the answer structure. Those positions are to be determined by the printer pattern and, having determined that the desired printer traverses the result in *preorder*, we now assert that any *admissable* result, σ, must be maximal in S and $\sigma \downarrow 1$ must be isolated at all levels.

The following reflexive equations develop admissable results. The original value domain is

$$S = (S \times S + A)_{\perp}.$$

We first remove $\perp_S$ from the lifted value domain, producing

$$S = S \times S + A.$$

However, we wish to alter this domain so that streams contain only finite lists in their heads. So, we instead restrict S to an incomplete partial order by applying a partial identity function f which is defined only on isolated points as follows:

$$S = f(S) \times S + A.$$

S is no longer a domain in the sense of Scott [19], but that fact does not matter because we do not intend it as a domain of computation. It is presented here *only* as an object of discourse describing the results that support the following theorem, where strictness compilation works as well as can be expected.

THEOREM 2. *The compiler preserves meaning.*

OUTLINE OF PROOF: Strictness patterns are propagated in a leftmost-outermost manner, so that by structural induction over the compiler rules a strictness pattern will be compiled with the appropriate expression. ∎

COROLLARY: *In its preorder traversal of a program's result, a printer proceeds exactly as far interpreting source code as interpreting object code when it prints an admissable value.*

LEMMA 1. *The compiler propagates only finite representations of strictness patterns.*

PROOF by structural induction on the compiler rules. There exists a finite representation for the initial inherited pattern, π_0. Rules 1-6, 8-10, 12 and 13 propagate a pattern that is at most finitely increased in length. Rule 7 is distributive, and thus propagates a pattern

shorter than that it receives. Rule 11 propagates either its inherited pattern, or a circularly defined pattern synthesized by the compilation of the lambda body. The circular pattern may represent an infinite pattern, but it can be represented with cycles, and so has a finite representation. ■

LEMMA 2. *The compiler executes a finite number of rules.*

PROOF: Rules 1 through 11 recursively invoke the compiler on proper subexpressions or not at all. A simple induction on the structure of the expression shows that it terminates in a finite number of steps. Rules 12 and 13 decrement the resource to limit the possible expansion of the source code. Thus the compiler applies the rules finitely often. ■

THEOREM 3. *The compiler terminates.*

PROOF: Finite cyclic graphs may be compared or combined in finite time. Thus, *meet*, *join*, and environment-lookup all terminate. By Lemmas (1) and (2), the compiler terminates. ■

7. Comparisons with other work

The fundamental work on strictness detection in first-order, flat domains is Mycroft's [18]. This has been extended to first order functions [4, 9]. Lists may be represented using functionals, but unrestricted higher-order analysis makes strictness detection intractable [9] . Our approach to the strictness problem is most similar to that of Hughes [10], who builds up strictness information with sets of continuations, or contexts. He independently restricts infinite strictness functions to "rational" functions similar to our "finite representations", so that a practical compiler is conceivable using his theory, but he doesn't claim one.

Lindstrom [16] proposes a domain that contains typing information as well as strictness information. However, he cannot represent patterns that are internally strict but externally lazy, such as $\langle \bot \ .\$\langle\langle \bot \ . \ \$\bot\rangle \ . \ \$\bot\rangle\rangle$. He treats streams as a future problem.

Wadler [22] is able to determine certain kinds of list strictness, such as strictness in all heads and tails, all tails or just the outer structure of the list. While this approach is useful in analysing some finite lists, he cannot handle streams , since some stream tails must remain lazy.

Jones [12] restricts strictness analysis only to information synthesized from application of

primitive operations (like **head** and **tail**). He does not consider "need" indirectly inherited from the output device and so arrives at an incorrectly strict **append** in the example **(head (append (cons 1 (bottom)) (2 3)))**.

Much work has been done on the strictness of higher-order functions [1, 3, 9, 12, 13, 17]. However, we regard the syntactic identification of **cons** to be crucial if a practical compiler is to result.

Conclusion

We have automatically analysed several programs using this approach, and regard it as a promising start on the problem of improving speed in circuit specifications and programming environments. Much remains to be done. There are many circumstances in which versions can and should be coalesced, and this problem needs some thought. In many cases, finite lists can be detected and should be marked as strict. For example, function arguments are often collected by a finite list which can safely be marked, although our examples do not take advantage of this fact. In addition, the notion of multiple buffering may be introduced, allowing more than just one element of a stream to be evaluated at a time. This would require that the user accept the loss of a buffer load of stream elements if any of those elements is $\perp_S$, a situation accepted by users of most conventional operating systems. This can easily be done by expanding the printer pattern appropriately. Work currently under way includes expanding this approach to handle first class functions and a more powerful "if" equation that detects such special cases as nil tests and accumulator style programming.

Acknowledgements

We would like to thank Steve Johnson and John Hughes for their remarks on an earlier draft, especially for some very useful advice on presenting the compiler equations. This work was supported in part by the National Science Foundation under a grant numbered DCR 84-05241.

References

1. S. Abramsky. Strictness analysis and polymorphic invariance. In Ganzinger and Jones (eds.), *Programs as Data Objects*, Springer, Berlin (October 1985), 1-23.
2. A. Bloss and P. Hudak. Variations on Strictness Analysis. *ACM Conf. on Lisp and Functional Programming*, (August 1986), 132-142.
3. G. L. Burn, C. L. Hankin and S. Abramsky. The theory of strictness analysis for higher order functions. In Ganzinger and Jones (eds.),*Programs as Data Objects*, Springer, Berlin (October 1985), 42-61.
4. C. Clack and S. L. Peyton Jones. Strictness analysis—a practical approach. *Functional Programming Languages and Computer Architecture; Lecture Notes in Computer Science* **201**, Springer, Berlin (1985), 35-49.
5. D. P. Friedman, and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner (eds.), *Automata, Languages and Programming*, Edinburgh University Press (1976), 257-284.
6. D. P. Friedman, and D. S. Wise, Unbounded Computational Structures, *Software: Practice and Experience* **8** (1976), 407-416.
7. C. V. Hall. *Compiling strictness into lazy lists.* Ph.D. Dissertation, Indiana University (expected 1987).
8. P. Henderson and J. H. Morris, Jr. A lazy evaluator. *Conf. Rec. 3rd ACM Symp. on Principles of Programming Languages* (January, 1976), 95-103.
9. P. Hudak and J. Young. Higher order strictness analysis for untyped Lambda Calculus. *Conf. Rec. 13th ACM Symp. on Principles of Programming Languages* (January, 1986), 97-109.
10. R.J.M. Hughes. Analysing strictness by abstract interpretation of continuations. In Abramsky and Hankin (eds.), *Abstract Interpretation*, Ellis-Horwood (to appear).
11. S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations.* M.I.T. Press, Cambridge, MA (1984).
12. N. D. Jones. Flow analysis of lazy higher order functional programs (April, 1986).
13. R. Kieburtz and M. Napierala. A studied laziness [extended abstract] (August, 1985).

14. A. T. Kohlstaedt. Daisy 1.0 reference manual. Tech. Rept. 119, Computer Science Dept., Indiana University, Bloomington (November 1981).

15. P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Comm. ACM* **8**, 2 (February, 1965), 89-101.

16. G. Lindstrom. Static evaluation of functional programs. *Proc. of the SIGPLAN '86 Symp. on Compiler Construction, SIGPLAN Notices* **21**, 7 (July 1986), 196-206.

17. D. Maurer. Strictness computation using lambda expressions. In Ganzinger and Jones (eds.), *Programs as Data Objects*, Springer, Berlin (October 1985), 136-155.

18. A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. *Proc. of Intl. Symp. on Programming, Lecture Notes in Computer Science* **83**, Berlin, Springer (1980), 269-281.

19. D. A. Schmidt. *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Newton, MA (1986).

20. D. A. Turner. Recursion equations as a programming language. In Darlington, Henderson and Turner (eds.), *Functional Programming and its Applications*, Cambridge University Press (1981).

21. J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. *J. Comput. System Sci.* **9** (1974), 332-354.

22. P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains), (November, 1985).