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:

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

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;

- $\pi = \bot$ 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: $\text{fix } \lambda\pi.\$\langle \pi \ . \ \pi \rangle \ = \ \top_{\mathbf{P}}$.

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.

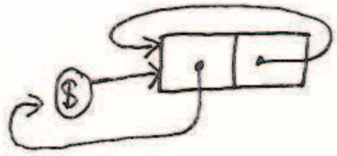
THEOREM 1. $\mathbf{P}$ is a complete lattice.

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:



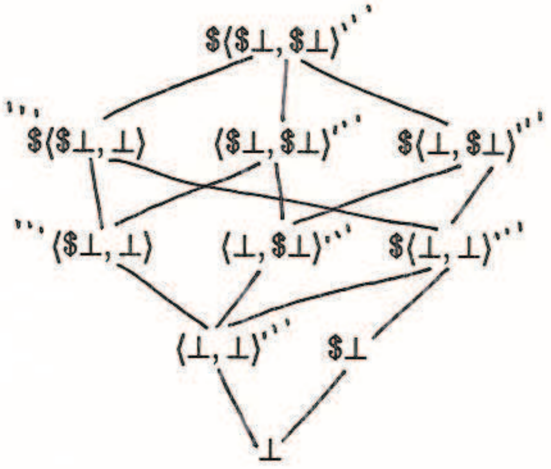


Figure 1. Partial lattice **P** of strictness patterns

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 ∈ *IDE* ; (identifiers)
e ∈ *EXP* ; (expressions)
const ∈ 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