GENERATING FUNCTION VERSIONS WITH RATIONAL STRICTNESS PATTERNS*

Cordelia V. HALL
Computing Science Department, University of Glasgow, Glasgow G12 8QQ, United Kingdom

David S. WISE
Department of Computer Science, Indiana University, Bloomington, IN 47401, U.S.A.

Communicated by M. Sintzoff
Received April 1988

Abstract. Expression evaluation in lazy applicative languages is usually implemented by an expensive mechanism requiring time and space which may be wasted if the expression eventually needs the values anyway. Strictness analysis, which has been successfully applied to flat domains and higher order functions, is used here to annotate programs in a first order language containing lazy list constructors so that they retain their original behavior, but run more efficiently. In practice, the strictness in fields within these constructors often follows regular patterns that can be finitely represented, especially in programs that manipulate such useful structures as finite or infinite trees. The approach presented here typically generates efficient, mutually recursive function versions for these programs. Weak and strong safety are defined and discussed, and the compiler is shown to be weakly safe. Termination is guaranteed by several factors, including a finite resource which controls the increase in code size, and a regularity constraint placed upon the strictness patterns propagated during compilation.

1. Introduction

Applicative lazy languages, such as Miranda [31], SASL [30], LML [3], Ponder [9], or Daisy [19], have many properties worth exploring. They have no side-effects, a fact that makes them interesting candidates for general-purpose parallel programming languages because control-flow problems are removed, leaving only the problem of reducing data dependencies. They facilitate the construction of infinite lists and produce values where applicative order, or call-by-value, languages loop forever. Finally, they permit expressions to be substituted for equivalent expressions, providing programs which are easier to reason about, thus supporting automatic proofs of correctness.

Unfortunately, implementations of these languages tend to be slow. "Lazy" or "delayed" evaluation provides the semantic power of these languages by permitting any given computation to avoid calculating values which are not required in computing the final value. This is generally implemented by a mechanism similar to Algol's

* This work was supported in part by the National Science Foundation grant number DCR 84-05241 and a SERC Research Fellowship.
call-by-name, except that instead of computing the value each time it is required (necessary in a language with side-effects), the value is computed only once. Usually, this mechanism, referred to here as a suspension, is implemented in a general way that does not distinguish between values that will eventually be required and values that are never needed. It saves a pointer to the environment current when the suspension was created, prolonging the existence of environments that might otherwise be garbage-collected, and also saves information needed to perform an expensive context swap if and when the suspended code is evaluated.

The essential idea motivating strictness analysis is that it is worthwhile avoiding the construction of suspensions that represent values required by the computation of the program's result.

1.1. Strictness analysis

Strictness analysis calculates the relationship between a function's arguments and its result. A function is said to be strict in argument \( n \) if the value of the function application is \( \bot \) when argument \( n \) is \( \bot \), where \( \bot \) represents an infinite loop in the domain of S-expression values.

The relationship between strictness analysis and the safe removal of suspensions is straightforward. If a function is strict in an argument \( n \), and if the function's result will be required by the whole computation, then there is no point in suspending the evaluation of either \( n \) or the function's result. Moreover, there is no point in suspending values upon which the computation of the value of \( n \) similarly depends. Since the whole computation would be \( \bot \) if any of these values were \( \bot \), there is no harm in letting \( \bot \) occur at a different point in the computation than it would have if the computation had been completely lazy. Of course, if none of these values were \( \bot \), then there is no point in suspending them anyway.

The identification of expressions that need not be suspended can also aid the implementation of lazy languages on parallel architectures. Functions, such as \texttt{add}, may be strict in more than one argument. Since there are no side-effects in the language and the values of the arguments are known to be necessary to the computation of the final result, these arguments may be evaluated simultaneously by processors that can be fully committed to their evaluation.

Interest in strictness analysis has steadily grown since Mycroft [25] first used abstract interpretation to determine strictness for flat domains (programs producing atomic values) in 1980. Clack and Peyton Jones [7] provide a useful clarification of Mycroft's work on flat domains, and provide measurements of the degree of parallelism achieved by applications of an algorithm similar to that of Mycroft.

Recent work has centered upon higher order functions in both the typed and untyped lambda calculus. Work on strictness analysis of higher order functions is directed towards a variety of problems. Like Clack and Peyton Jones, Maurer's work, an extension of Mycroft's result to typed higher-order functions [24], is motivated by an interest in exploiting possible parallelism in functional languages.
Hudak and Young [14] develop an algorithm for performing higher order strictness analysis in the untyped lambda calculus, based upon a set-theoretic description of strictness. Wray [34] extends Mycroft's result with an algorithm that annotates strict expressions in lazy higher order combinators. Kuo and Mishra [21] show that strictness analysis can be shown to be a particular case of type inference for the typed lambda calculus. Nielson [26] develops a theory of abstract interpretation for the typed lambda calculus, which is shown to also be suitable for strictness analysis. Burn, Hankin and Abramsky [5] use abstract interpretation to analyze higher order functions in the typed lambda calculus; Abramsky [2] extends these results to polymorphic types.

Two forms of analysis are currently applied to first order functions in non-flat domains; abstract interpretation and "backwards analysis" [16]. Wadler [32] uses abstract interpretation 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. Kieburtz and Napierala [20] use abstract interpretation to develop total interpretations, which can be used by a compiler without risk of non-termination; they then develop such an interpretation for strictness analysis. Hughes [17] analyzes a first order functional language containing only variables, function applications and case expressions, using a simple domain of contexts which carry strictness. He presents a theoretical framework for strictness analysis [15], using continuations to represent contexts. Wadler and Hughes [33] present another theoretical treatment of contexts using a finite domain of retracts, or projections for analysis of a monomorphic first order language. Hall and Wise [11] describe a compiler that selects an infinite domain of strictness patterns (analogous to contexts) and that demonstrates the value of creating versions to take full advantage of this augmented information.

Finally, some work has been done on combining the analysis of first order functions on non-flat domains and higher order functions. Hughes [16] compares two approaches to strictness analysis. One is based upon abstract interpretation and the other involves reasoning from information about the strictness of an expression to deduce information about a sub-expression, or "backwards" analysis. He argues that backward analysis is likely to be more efficient than forward analysis and that it can be extended to provide strictness information about lists and higher-order function in typed languages. Hall extends a form of "backwards" analysis [11] to a restricted set of higher-order functions [12].

1.2. Overview of approach presented here

These results are a substantial extension and revision of earlier work [11]. The general technique is discussed in detail, along with several short programs that explain why the compiler has developed along these lines. The compiler rules have been improved and corrected. They are now written in a simpler notation, and annotate only cons expressions. The compiler is shown to be weakly safe under an
original definition of weak safety that is relevant to list strictness in general, and proofs of termination and safety have been expanded.

The untyped source and target language of the compiler presented here is a first-order subset of Daisy, an applicative statically-scoped lazy language developed from the original Lisp interpreter. Delayed evaluation is achieved by altering only the behavior of \texttt{cons}, a few list predicates, and list accessing functions such as \texttt{car} and \texttt{cdr}. By delaying the evaluation of its arguments, \texttt{cons} provides the only laziness required to produce normal order semantics. Functions receive a single argument, which may be a list. Often, the function treats sub-structures of this argument list as if they were distinct arguments; it is this convention which guarantees that evaluation of function arguments (sub-structures) will be delayed until the argument value is required by the computation.

Expressions in Daisy are reduced to weak head normal form \cite{27}, and the resulting expression is then given to the output device, or \texttt{printer}, which traverses it in order to produce the final value. Thus the printer may be said to create the initial demand for a value at run time. The compiler uses an abstraction of this demand to determine which nodes in the code tree will, when evaluated at run time, produce values which are required. It does this by recursively traversing the abstract syntax tree, associating a demand or \textit{strictness} pattern with each node, which is said to \textit{inherit} this pattern. If the node being compiled is the application of a lambda abstraction, then it is convenient to combine patterns inherited by instances of the formal variable in some way, and this combination is said to represent the \textit{synthesized} pattern of the lambda abstraction.

Strictness patterns are presented as elements in a lattice for two reasons. Instances of a formal variable may inherit different patterns, and both of these instances may represent values demanded by the printer. When this occurs, it is convenient to combine the strictness patterns inherited by each instance of such a variable so that the strictness information contained in each pattern is preserved. Instances of a variable may also appear in both arms of a conditional expression, only one of which will be executed. Here, it is necessary to be able to combine strictness patterns so that the resulting pattern contains only the information that all of the patterns inherited by these instances have in common.

Strictness patterns allow the compiler to determine which portions of the code tree can be evaluated early. When associated with a given node $n$, a pattern has the following meaning:

1. $\bot$ means that the compiler has not determined whether the evaluation of $n$ will produce a value that will be required by the printer;
2. $\$\bot$ means that $n$ will produce a value that will be required at run-time; if the value is a ‘flat’ value, such as a number or character, then it will be required, otherwise the outer structure of the \texttt{cons} cell value will be needed, but the fields of the cell may not be and nothing is known about them. For example, if $\$\bot$ is inherited by \texttt{(add (list a b))}, then the compiler has recognized that the result of the addition will be required, and so the arguments to the addition can be evaluated early. If
$\bot$ is inherited by (cons a b), then the compiler has recognized that the outer structure of the cons cell will be required (so the expression representing it must be evaluated until at least the cons cell exists), but cannot determine that the fields within the cell will be evaluated, and so leaves them alone;

(3) $\langle p1, p2 \rangle$ has the same meaning as $\bot$, except that $p1$ and $p2$ may provide more information about the fields of $n$'s value if it is a cons cell. For example, if $\langle \bot, \bot \rangle$ is associated with (cons a b), then the compiler has determined that both the outer structure of the cons cell and that of the first field is required.

(4) The meaning of $\langle p1, p2 \rangle$ is similar to that of $\langle p1, p2 \rangle$, but will be described further when a constraint preserving stream output is presented.

When a cons expression inherits a list pattern, such as $\langle p1, p2 \rangle$, the compiler marks the argument corresponding to $p1$ with a strictness mark $\bot$ if that is indicated by $p1$, and treats the other argument symmetrically. These marked expressions are then evaluated early at run-time.

Initially, it seems reasonable to associate the infinite pattern

$$\text{fix}(\lambda \pi.\bot(\pi, \pi))$$

with the root of the code tree, as it indicates a demand for any possible element (the outer structure and all heads and tails) in the final value of the program. During the following discussion, this initial pattern will be assumed until it is modified to preserve stream output.

The compiler uses the pattern inherited by a given node to deduce the pattern to be associated with each sub-tree of that node. This is done in the following way:

- Constants do not loop, so may always be marked safely.
- Any node inheriting a pattern without strictness marks is essentially left alone, as there are no strictness marks to be transferred from the pattern to a cons expression within the code.
- If $\bot$ or $\langle p1, p2 \rangle$ is inherited by (head e), then e inherits a new list pattern, where the original pattern is embedded within it in the appropriate position. (It is assumed that head is eventually given a cons cell as its argument.) For example, if (head (cons a b)) inherits $\bot$, then (cons a b) inherits $\langle \bot, \bot \rangle$. The object code produced is (head (cons a b)). Applications of tail are treated symmetrically.
- When $\bot$ is inherited by (cons a b), the cons expression itself is not explored further, however it may be marked if it is the argument of an outer cons expression. If $\langle p1, p2 \rangle$ is inherited by (cons a b), then $p1$ is inherited by a and $p2$ is inherited by b.
- When either $\bot$ or $\langle p1, p2 \rangle$ is inherited by the application of a primitive that produces a flat value, such as a number, boolean or character, $\bot$ is inherited by each argument in the cons expression that forms the actual parameter of the application. Naturally, the compiler rules for these primitives may vary, according to the use these primitives make of their arguments when the value of their application is required. For example, or does not always evaluate its second argument, and so a compiler rule for or would associate $\bot$ with that argument.
- The application of a lambda abstraction often contains more than one reference to the formal variable in the lambda body. The compiler uses a special function, the compile-time environment, to accumulate and combine the patterns inherited by instances of the formal variable. The body of the lambda abstraction inherits the pattern originally inherited by the application, and the pattern associated with the formal variable by the compile-time environment after the body has been explored is then inherited by the application argument.

- The application of a recursively defined lambda abstraction allows functions to be named. These names may inherit strictness patterns requiring that the corresponding definitions contain cons expressions whose arguments are not marked in the same way. The approach presented here assumes that each function definition is to be made as efficient as possible. If the compiler can only produce one definition, or version, per function, then the pattern used to compile this definition must include only the information contained in the intersection, or meet, of the patterns inherited by instances of the function. This new pattern may contain very little of the strictness information contained in some of these patterns. The alternatives are either to introduce no marks into the function definition or to use a marked definition which would be unsafe in some cases. However, if the compiler is permitted to create several versions for a particular function, then it is possible to identify cases in which little space is consumed by an additional version which will be executed many times. Examples of such cases appear in Section 2.4. The compiler uses the compile-time environment to maintain information about the versions it creates for each function definition. When a definition is compiled, the environment contains (1) the original, unmarked definition so that it may be copied to create new versions, (2) a function from inherited to synthesized patterns, used to determine whether an instance of this function has inherited a given pattern elsewhere, and if so, that version’s synthesized pattern, and (3) the number of versions created so far.

- Data may also be defined recursively, and versions are created for these definitions as well. For each recursive data definition, the compile-time environment maintains an entry similar to that for a function.

The lattice of strictness patterns contains infinite patterns that may be divided into two categories, those that can and those that cannot be represented by finite cyclic graphs. An irrational pattern can be said to correspond to the decimal expansion of an irrational number, such as pi. The compiler is constructed so that it propagates only rational patterns, or those patterns that can be represented by a finite graph. (Hughes independently recognized the importance of using rational contexts to guarantee termination [17].) The reasons for this restriction are as follows: the meet or join of two patterns, at least one of which is irrational, will not terminate, and the construction of versions may not terminate. For example, a simple recursive definition such as

\[ f = (\text{lambda} \, (\, \text{cons} \, (\text{add} \, (\text{list} \, 1 \, \text{head} \, l)) \, (f \, \text{tail} \, l))) \]

would not terminate if the initial application of f inherited a pattern corresponding...
Generating function versions with rational strictness patterns

to the decimal expansion of \( \pi \), because all of the suffixes of this pattern are unique.
As the recursive call to \( f \) was compiled (Fig. 1), its inherited pattern would never
match that of a previously compiled call. However, the compilation of \( f \)'s application
would terminate if its inherited pattern were a rational pattern, such as a cyclic
pattern requiring every other element of an infinite list. The recursive call would
then inherit a suffix of the original pattern, and the recursive call in the ensuing
version would then inherit a suffix which is identical to the pattern inherited by the
original application of \( f \). This allows the compiler to create a reference back to the
version inheriting the original pattern. It then terminates after creating two mutually
recursive versions for \( f \) (Fig. 2), allowing every other element in the stream produced
at run-time to be evaluated early.

One might think that the compiler will now terminate in general, since it propagates
only rational patterns (and since the initial pattern inherited by the root of the code
tree is rational). However, it is still possible to construct a pathological function
that, when compiled, causes an infinite number of versions to be created, all of

---

![Fig. 1.](image-url)
which are associated with unique rational patterns. For example, if an application of a perverse function such as \( g \) is compiled with the pattern \( S_\perp \), where

\[
g = (\lambda (a) (\text{tail} (g a))),
\]

then compilation of the inner recursive call to \( g \) becomes an infinite process. The call initially inherits a pattern in which \( S_\perp \) is embedded in the pattern tail, and since no version currently exists associated with this pattern, it is necessary to create a new one, whose body must also be compiled. During this process (Fig. 3), it becomes necessary to embed \( S_\perp \) in longer and longer finite patterns in order to ensure that it is inherited by compilation of the appropriate code. For this reason, and to allow the user to control the number of versions created for each function, the user passes a \textit{resource} to the compiler, which limits the number of versions created.

The initial strictness pattern modelling the behavior of the printer has been described as demanding all of the heads and tails of the value produced at run-time.
As it stands, this pattern will cause the loss of stream output. For example, if a function application producing a stream of integers, such as

\[(\text{ints} \, 1)\]

where \(\text{ints} = (\lambda \, i. \, (\text{cons} \, i \, (\text{ints} \, (\text{add} \, 1 \, i))))\)

inherits this initial strictness pattern, the \(\text{cons}\) expression will be marked strict in both of its arguments and none of the stream elements can be printed before the next is evaluated, causing an infinite loop which produces no output rather than one that does produce output. However, if this initial pattern does not demand the outer structure of any tail, then the recursively constructed tail of this stream of integers will be constructed lazily, permitting the head of the stream to be output before further evaluation of the tail takes place, even though there may be \(\text{cons}\) expressions within this tail that evaluate one or both arguments early. The compiler receives this modified initial pattern, and uses it to constrain any cyclic pattern created by the exploration of a recursively defined function.

All synthesized patterns are now created by taking the \textit{meet} of any least upper bound with the printer pattern itself. Otherwise, the \textit{join} of two patterns might produce a pattern such as

\[\text{fix}(\lambda \, n. \, $(\bot, \, n))\],

which is strict in all tails. Aside from the printer pattern, synthesized patterns are the only patterns that need to be controlled in this way, because only they represent potentially new cyclic patterns that may cause recursive versions to be created as the patterns are unrolled during compilation.

The compiler rules for \textbf{head} and \textbf{tail} applications are extended to handle the new class of patterns, \(\langle p_1, p_2 \rangle\), introduced by the modified initial pattern, as follows:
the outer mark of the pattern inherited by the application is preserved in the list pattern inherited by the argument. If there is no outer mark on the original pattern, then the list pattern has no outer mark either. No other rules are changed, although it is interesting to note why applications of primitives returning flat values are compiled in the same way as they were. The reason is that although the new pattern does not have an outer mark, it does contain an internal mark, and under the previous initial pattern would have had an outer mark as well. There is no danger of its result generating an infinite stream with output that would be lost, since its result is a flat value.

The compiler is defined as being weakly safe, meaning that the values produced by evaluation of the object code produced by this compiler may lose some output if they are lists that contain an infinite loop. Weak and strong safety are discussed in more detail in Section 2.7.

1.3. Outline of paper

The compiler and a lattice of strictness patterns are presented in Section 2. Section 3 compares this approach with related work, suggests possible future research and summarizes the contribution of this work.

2. The lattice of strictness patterns and the compiler

Section 2.1 defines strictness patterns more formally, and Sections 2.2-2.3 present the compiler domains, equations and some small examples. Section 2.4 contains two extended examples, Section 2.5 discusses the technique used to find pattern fixed points, and Sections 2.6-2.7 describe termination and safety properties of the compiler.

2.1. Strictness patterns

All functions in Daisy take one argument. This argument is similar to a Lisp S-expression. If the argument is a binary tree, then different fields within it may be regarded as the function's arguments and the entire structure is then called the argument collection. For this reason, the definition of a strict function is expanded to specify an index for each part of the argument collection in which the function is strict. Consider the conventional labelling of a binary tree with root labelled '1', right children successively labelled with '1' and left children labelled with '0'. The index of each node in this tree is the number represented by the concatenation of bits labels along the path from the root to its location. The following displays the indexing of a tree:

\[ 1(2(4\ldots,5\ldots),3(6\ldots,7\ldots)). \]
In the following definition, \((n \, ac)\) selects an argument at index \(n\) in the argument collection \(ac\).

**Definition.** A function \(f\) is strict in an argument \(a\) at index \(n\) of its argument collection \(ac\) if \(((n \, ac) = \bot) \Rightarrow (f \, ac = \bot)\).

A tree marked with \$ at any indexed subtree is to be evaluated by an interpreter using call-by-value for the marked field. 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 the program
\[
(\text{cons } \$(\text{cons } b \, c) \, d)
\]
is lazy in the values of \(b, c, d\) but strict in the external structure of the pair \((\text{cons } b \, c)\).

An expression is strict in a given sub-expression when the sub-expression and all containing structures are marked. For example, the program
\[
(\text{cons } (\text{cons } \$(\text{cons } a \, b) \, \text{nil}) \, \text{nil})
\]
is not strict in \(a\), but
\[
(\text{cons } \$(\text{cons } \$(\text{cons } \$a \, b) \, \text{nil}) \, \text{nil})
\]
is strict in \(a\).

### 2.1.1. Definition of lattice \(P\)

The domain \(P\) is defined by the reflexive equation
\[
P = \$P + (P \times P)_{\bot}
\]
where \(\$P = \{\pi \mid \pi \in P\}\), \(+\) is coalesced sum and all lifting is explicit \([28]\), subject to the constraint that \(\$\pi = \$\$\pi\). This domain is sketched in Fig. 4. \(P\) is a complete
lattice [29], with a top element,

\[ \text{fix}(\lambda \pi.\$\langle \pi, \pi \rangle) = T_P. \]

An important element of \( P \) is the \textit{printer} pattern

\[ \pi = \$\text{fix}(\lambda \pi.\langle \pi \rangle) \neq T_P, \]

which can be abstractly represented as a finite cyclic graph (Fig. 5). (Note that this pattern contains a mark outside the scope of the \textit{fix} expression defining it; this is not the same pattern as

\[ \text{fix}(\lambda \pi.\$\langle \pi \rangle), \]

which represents a pattern with marked tails.)

The \textit{meet}, \textit{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, all patterns belong to the set of finitely representable elements in \( P \).

\[ \text{Fig. 5.} \]

2.2. The compiler

The compilation of a program inherits the \textit{printer} pattern, \( \pi \), which is strict in its outer structure as well as the heads of all trees and sub-trees. This strictness pattern assumes a leftmost-outermost evaluation order, and allows the compiler to find strictness in programs that generate trees, including infinite trees. As it recursively traverses the abstract syntax tree, the compiler builds up strictness information about identifiers and functions in a special symbol table referred to here as the compile-time environment. It also receives an integer resource that bounds the number of different versions that can be created for any given function or data recursion.

The next two sections present syntax domains and a grammar for a restricted form of Daisy.
2.2.1. Restricted Daisy syntax domains

const $\in$ CONST; \hspace{1cm} (constants)
$id \in$ IDE; \hspace{1cm} (identifiers)
e $\in$ EXP. \hspace{1cm} (syntactic expressions)

2.2.2. Restricted Daisy syntax

e ::= expr | $expr$
expr ::= 
const | constants
[ ] | nil
(exprs) | lists
prim:(e e) | primitives with 2 arguments
head:e | head application
tail:e | tail application
if:(e e e) | conditional application
(\lambda id. e):e | lambda application
(fix :[id \lambda id. e]):e | application of a recursive function
fix :[id e] | data recursion
id:e | function application
id | identifiers

exprs ::= e exprs | e.e | empty

Expressions surrounded by double brackets are syntactic expressions in the source and target language. Syntactic expressions will often contain numbered sub-expressions, so that they are easier to discuss. For example, $[[\text{prim}:(e e)]]$ becomes $[[\text{prim}:(e1 \cdot e2)]]$. Application associates to the right, so that head:tail:a can also be written as head:(tail:a).

2.2.3. Restricted Daisy value domains and semantic functions

\[ A; \hspace{1cm} \text{(atoms)} \]
\[ S = A + (S \times S) + (S \to S). \hspace{1cm} \text{(structures)} \]

Johnson presents a denotational semantics for Daisy [18]. A formal operational semantics for Daisy might permit discussion of the relative performance of the compiler’s source and object code, however such a discussion is beyond the scope of this work.

2.2.4. Compiler domains

The compiler is given a syntactic expression, a strictness pattern, a compile-time environment that performs some bookkeeping, and a natural number that limits the
number of versions to be created for any one function. The rules alter only the expression and compile-time environments given them. The domain of strictness patterns has already been defined, but the domain of environments has interesting structure which is described in more detail in the following section.

\[ C: D \rightarrow D; \quad \text{(compiler)} \]
\[ D = EXP \times P \times ENV \times INT; \quad \text{(structure of compiler domain)} \]
\[ \pi: P = SP + (P \times P); \quad \text{(strictness patterns)} \]
\[ \rho: ENV = V \rightarrow (BEXP \times PF \times INT \times BTAG) + unbound; \quad \text{(compile-time environment)} \]
\[ \upsilon: NAT. \quad \text{(resource)} \]

2.2.5. Domain of compile-time environments

The compile-time environment allows the compiler to predict the scope in which expressions will be evaluated at run-time. A distinction is made between variables that are lambda bound and variables, such as functions and data recursions, that are recursively bound. The tags \texttt{lambda} and \texttt{fix} respectively identify entries for these two types of variables. However, each entry is padded, if necessary, so that entries for both types have the same general structure. An entry for a lambda bound variable has the following structure;

(1) A dummy syntactic expression;
(2) A pattern representing the cumulative strictness information inherited by all instances of the variable seen so far;
(3) A dummy count;
(4) The tag \texttt{lambda}.

An entry for a recursively bound variable has the following structure;

(1) A copy of the definition of the function or data recursion represented by the variable, permitting the compiler to unfold recursive references and mark them appropriately when creating versions;
(2) A function from inherited patterns to synthesized patterns, which allows the compiler to distinguish between the patterns inherited by instances of the function (or data recursion) represented by the variable, and retrieve the pattern synthesized for that version when needed. Since data recursions are not applications, the synthesized pattern for each version of a data recursion is always \(\bot\);
(3) A version count, which allows the compiler to keep track of the number of versions created for a given function or data recursion and cease creating versions when the count is equal to the resource;
(4) The tag \texttt{fix}.

The domain of environments, \(ENV\), contains only environments with pattern entries (second elements of the entry tuple) that are at most \(\Pi\), and that, if functions,
map elements from $P$ into $P_\Pi$, a sub-lattice of $P$ with the top element $\Pi$.

\[ \nu: \quad V = \text{ID} + (\text{ID} \times P); \]

\[ B\text{EXP} = [\text{}][\text{}][\text{fix :[id1 \ lambda id2. e]}] + [\text{fix :[id e]}] \]

\[ \text{pa:} \quad PF = (P \to (P_\Pi + \text{unbound})) + P_\Pi \]

(inherited and synthesized pattern entries in environment)

\[ P_\Pi = \{ \pi \in P \mid \pi \subseteq \Pi \}; \]

\[ B\text{TAG} = \lambda e. \text{fix}. \]

(binding tags in environment)

The following functions are projections on environment entries:

\[ \text{Binding} = \lambda e. \text{e} \downarrow 1, \]
\[ \text{Pat-fun} = \lambda e. \text{e} \downarrow 2, \]
\[ \text{Binding-type} = \lambda e. \text{e} \downarrow 4. \]

2.3. 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, compile-time environment and resource. The pattern $\Pi$ is the pattern to be initially propagated by the compiler, however the examples in both this and the next section use a variety of patterns.

2.3.1. Notation

The following notation is introduced:

- $\alpha \cdot \pi$ represents a strictness pattern, $\pi$, that may or may not be prefixed with $\$. The pattern matching convention is that if $\$ is the prefix, then $\alpha = \$, otherwise $\alpha$ is empty. For example, if $\alpha \cdot \pi = \text{\textless }\$, then $\alpha = \$ and $\pi = \text{\textless }$. However, if $\alpha \cdot \pi = \text{\textless }$, then $\alpha = \text{empty}$ and $\pi = \text{\textless }$.

- A single vertical bar indicates concatenation of an identifier with a strictness pattern; this forms a new version name.

- Unsubscripted $\text{\textless }$ stands for $\text{\textless }P$.

- If $(\$ \in \alpha \cdot \pi)$, then $\alpha \cdot \pi$ is a strictness pattern in which a strictness mark appears.

- The list projection functions $\downarrow 1$ and $\downarrow 2$ select the appropriate sub-pattern and ignore any pattern prefix. For example, $\$\langle p1, p2 \rangle 1$ is $p1$, as is $\langle p1, p2 \rangle 1$.

- $[[a]]/[e']$ refers to the substitution of $[[a]]$ for all instances of $[[e']]$ in $[[e']]$.

- If a syntactic expression $[[e']]$ is a subexpression of another expression $[[e]]$, then this relationship is denoted as $[[e']] \subseteq [[e]]$.

- $x$ contains $\downarrow s$ if $\downarrow s \in x$.

2.3.2. Compiler rules

The insertion of strictness marks during the compilation process is idempotent, and expressions marked by the programmer may be compiled.

\[ C[[\text{const}]] \alpha \cdot \pi \rho i = [[\text{const}]] \alpha \cdot \pi \rho i. \]
Constants cannot cause an infinite loop when they are evaluated, so they can always be safely marked. Nil is treated as a constant where it appears explicitly.

\[(C2) \quad C[e] \alpha \cdot \pi \rho \iota, \text{ such that } (\$ e \alpha \cdot \pi) = \begin{cases} \llbracket \exp \rrbracket \alpha \cdot \pi \rho \iota \\ \text{where} \llbracket \exp \rrbracket = \llbracket e(Binding(\rho[e]))/\llbracket e \rrbracket \rrbracket \\ \text{if } \exists [e'] \in [e] \text{ such that } [e'] \in ID \\ \& (Binding-type(\rho[e'])) = \text{fix}, \\ \llbracket e \rrbracket \alpha \cdot \pi \rho \iota \text{ otherwise.} \end{cases}\]

During compilation of recursively defined expressions, the compiler may associate a pattern without any strictness marks in it with a recursive reference, such as a function call or data recursion. The versions enclosing this reference may be labelled differently, and so a locally defined lazy version is inserted. Except for this special case, compilation stops once the inherited strictness pattern cannot improve the source code.

\[(C3) \quad C[\llbracket \text{head}: e \rrbracket] \alpha \cdot \pi \rho \iota = \llbracket \text{head}: e \rrbracket \alpha \cdot \pi \rho_1 \iota \\ \text{where} \llbracket e_1 \rrbracket \alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[e] \alpha \cdot (\alpha \cdot \pi, \perp) \rho \iota. \]

\[(C4) \quad C[\llbracket \text{tail}: e \rrbracket] \alpha \cdot \pi \rho \iota = \llbracket \text{tail}: e \rrbracket \alpha \cdot \pi \rho_1 \iota \\ \text{where} \llbracket e_1 \rrbracket \alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[e] \alpha \cdot (\perp, \alpha \cdot \pi) \rho \iota. \]

Patterns inherited by applications of head or tail are injected into a list pattern to eventually be inherited by an expression that produces a list when evaluated.

\[(C5) \quad C[\llbracket \langle e_1 . e_2 \rangle \rrbracket] \alpha \cdot \pi \rho \iota = \begin{cases} C[\llbracket \langle e_1 . e_2 \rangle \rrbracket] \alpha \cdot \pi \rho \iota, \text{ such that } (\$ e \alpha \cdot \pi) \text{ if } (\$ \notin \pi); \\ \llbracket \langle \alpha_1 . e_1 . \alpha_2 . e_2 \rangle \rrbracket \alpha \cdot \pi \rho_2 \iota, \text{ otherwise;} \end{cases} \\ \text{where} \begin{align*} \alpha_1 \cdot \pi_1 &= (\pi \downarrow 1) \\
\alpha_2 \cdot \pi_2 &= (\pi \downarrow 2) \\
\llbracket e_1 \rrbracket \alpha_1 \cdot \pi_n \rho \iota_1 &= C[e_1](\pi \downarrow 1) \rho \iota; \\
\llbracket e_2 \rrbracket \alpha_1 \cdot \pi_1 \rho_2 \iota &= C[e_2](\pi \downarrow 2) \rho_1 \iota. \end{align*} \]

The compilation of cons first 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. The returned object code is then marked if the corresponding sub pattern is marked. When \(\pi\) doesn't contain a strictness mark, the second compiler rule, \((C2)\), is executed; if this expression appears in an outer cons expression and \(\alpha = \$\), the compilation of that outer cons will then mark it.

For example, if \(\exp = \llbracket \text{head}: (\text{tail}: (a . b) . c) \rrbracket\) where the free variables are lambda bound, then \(C[\llbracket \exp \rrbracket \text{fix}(\lambda \pi. (\$ \pi, \pi)) \rho 4 = \llbracket \text{head}: (\text{tail}: (a . b) . c) \rrbracket\).

In this next rule, a slightly different strictness notation for lists is introduced. The list syntax is converted to dotted notation when compiled. The list \(\langle x \, y \rangle\) is strict
in both $x$ and $y$—the meaning is the same as $\langle x . \langle y \rightarrow \rangle \rangle$.

(C6) \[ C[\text{prim} : (e1 e2)] \alpha \cdot \pi \rho_1 \iota = C[\text{prim} : e_1] \alpha \cdot \pi \rho_1 \iota \]

where \[ e_i \alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[\text{add} : (a \cdot b)] C[\text{add} : (c \cdot d)] \rho \iota. \]

Primitives (arithmetic and logical) are assumed to be strict in both arguments. Other rules can be constructed to handle primitives such as or, which is strict only in its first argument.

For example, if

\[ \text{exp} = \text{add} : (\text{head} : (a \cdot b) \cdot \text{tail} : (c \cdot d)) \]

and the free variables are lambda bound, then

\[ C[\text{add} : (\text{head} : (a \cdot b) \cdot \text{tail} : (c \cdot d))] \]

are

(C7) \[ C[\text{if} : (e1 e2 e3)] \alpha \cdot \pi \rho_4 \iota =\]

\[ C[\text{if} : (e1 \cdot e2 \cdot e3)] \alpha \cdot \pi \rho_4 \iota \]

where

\[ e_1 \alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[e1] \rho \iota; \]
\[ e_2 \alpha_2 \cdot \pi_2 \rho_2 \iota_2 = C[e2] \rho \iota; \]
\[ e_3 \alpha_3 \cdot \pi_3 \rho_3 \iota_3 = C[e3] \rho \iota; \]

\[ \rho_4 = \lambda a. \begin{cases} \text{unbound} & \text{if } \rho_2 \cdot i = \text{unbound} & \text{& } \rho_3 \cdot i = \text{unbound}; \\ \langle \text{binding}_2, (pa_2 \setminus pa_3), 0, b \cdot \text{type}_2 \rangle & \text{if } b \cdot \text{type}_2 = \lambda \text{a}; \\ \langle \text{binding}_2, pa_4, \nu \cdot \text{count}_2 \cup \nu \cdot \text{count}_3, b \cdot \text{type}_2 \rangle & \text{if } b \cdot \text{type}_2 = \text{fix}; \end{cases} \]

where

\[ \langle \text{binding}_2, pa_2, \nu \cdot \text{count}_2, b \cdot \text{type}_2 \rangle = \rho_2 \cdot i \]
\[ \langle \text{binding}_3, pa_3, \nu \cdot \text{count}_3, b \cdot \text{type}_3 \rangle = \rho_3 \cdot i \]
\[ pa_4 = \lambda \text{pat}. \begin{cases} (pa_3 \text{ pat}) & \text{if } (pa_2 \text{ pat}) = \text{unbound}; \\ (pa_2 \text{ pat}) & \text{otherwise.} \end{cases} \]

The compile-time environment returned by this rule is complicated because it is necessary to constrain the strictness information associated by $\rho_2$ and $\rho_3$ with instances of formal variables in each branch. In addition, some bookkeeping is performed to handle any new versions that may have been created in each branch. After compilation of the first branch is completed, $\rho_2$ contains an approximation of the demand made by the predicate and first branch upon expressions bound to the formal variables defined in enclosing scopes. After compilation of the second branch is completed, $\rho_3$ contains similar information. The environment returned by this rule, $\rho_4$, inserts the meet of two patterns, contributed by $\rho_2$ and $\rho_3$, into the entry created for each of these formal variables. New versions may be created in either branch, so $\rho_4$ also contains updated entries for recursively defined variables encountered in outer scopes. For each such entry, the function mapping inherited to synthesized patterns must contain all of the mappings in both $\rho_2$ and $\rho_3$. There
may or may not be a set of versions common to both branches; the addition of the version counts from \( \rho_2 \) and \( \rho_3 \) covers both possibilities.

\[(C8)\]  
\[
C[(\lambda \text{id. body}):e]\alpha \cdot \pi \rho \iota = \\left\{ (\lambda \text{body},):e_1]\alpha \cdot \pi \rho_4 \iota \right. 
\]

\[\text{where}\]
\[
[\text{body}_1]\alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[\text{body}]\alpha \cdot \pi \rho_2 \iota \\
\rho_2 = \lambda i.\left\{ \begin{array}{ll}
\rho i & \text{if } i = \text{[id]} \\
\rho_1 i & \text{otherwise;}
\end{array} \right. \\
\rho_3 = \lambda i.\left\{ \begin{array}{ll}
\rho i & \text{if } i = \text{[id]} \\
\rho_1 i & \text{otherwise;}
\end{array} \right. \\
[e_1]\alpha_2 \cdot \pi_2 \rho_4 \iota_2 = C[e] (\text{Pat-fun} (\rho_1 \text{[id]})) \rho_3 \iota.
\]

The body of the lambda abstraction must be compiled before the effect of its use on the expression bound to the formal variable can be determined. The compilation of the body is passed a new compile-time environment extended to include an initial entry for the formal variable which is structured as follows; a (meaningless) syntactic expression, an initial inherited pattern indicating that this variable is not yet known to represent a required value, a version count (again meaningless), and a tag which indicates that [id] was bound in a lambda environment. As the compiler explores the body of the lambda expression, the pattern inherited by [id] is updated. When analysis of the body is complete, the formal variable has inherited a composite pattern which becomes the synthesized pattern for this lambda expression. The projection function, \( \text{Pat-fun} \), retrieves this pattern so that it can be propagated to the compilation of the expression bound to the formal variable, \( [e] \).

For example, if

\[\text{exp} = [(\lambda a. (\text{head:a . head:a})): (b . c)]\]

and the unbound variables are lambda bound, then

\[(C9)\]
\[
C[\text{exp}] (\$\bot, \$\bot) \rho 1 = [(\lambda a. (\text{head:a . head:a})): (\text{Sb . c})] \\
\]

\[
\left\{ (\text{fix: } [f \lambda \text{id. body}]):e]\alpha \cdot \pi \rho \iota = \left( (\text{fix: } [f \lambda \text{id. body}]):e_1]\alpha \cdot \pi \rho_4 \iota \right. 
\]

\[\text{where}\]
\[
[\text{body}_1]\alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[\text{body}]\alpha \cdot \pi \rho_2 \iota \\
\rho_2 = \lambda i.\left\{ \begin{array}{ll}
\rho i & \text{if } i = \text{[fix]} \\
\rho_1 i & \text{otherwise;}
\end{array} \right. \\
\rho_3 = \lambda i.\left\{ \begin{array}{ll}
\rho i & \text{if } i = \text{[id]} \text{ or } i = \text{[fix]} \\
\rho_1 i & \text{otherwise;}
\end{array} \right. \\
\rho_{\text{pat}} = \text{Pat-fun} (\rho_1 \text{[id]}) \\
\rho \iota_2 = \text{[e]} \rho_3 \iota.
\]
The compiler constructs a synthesized pattern $rec-p$ by recursively defining the result of the analysis of the $\lambda$ body. This pattern is then inherited by the argument $[x]$. (Section 2.5 on the implementation of the compiler discusses the actual construction of these recursively defined patterns as well as the significant problem that arises when pattern bindings are not maintained as explicitly labelled objects by the compiler.)

In the following example, and those throughout the rest of the paper, identifiers are labelled with pattern names, rather than patterns. The patterns represented by these names are listed after the example.

If

$$exp = \left[ (\text{fix}: [f \lambda lst. (\text{add}: (\text{head}: lst 2). f: \text{tail}: lst)]: a) \right]$$

and the free variables are lambda bound, then

$$C[exp] \text{fix}(\lambda \pi. (\pi \lambda \pi))(\rho_1) = \left[ (\text{fix}: [f \rho_1 \lambda lst. (\text{add}: (\text{head}: lst \pi 2). f: \text{tail}: lst)]: a) \right]$$

where $\rho_1 = \text{fix}(\lambda \pi. (\pi \lambda \pi))$.

The compilation of $[a]$ then inherits the pattern $\$\text{fix}(\lambda \pi. (\pi \lambda \pi))$.

(C10)  \[
C[\text{fix}: [id e]] \alpha \cdot \pi \rho i = \left[ \text{fix}: [id | \alpha \cdot \pi e_1] \right] \alpha \cdot \pi \rho_3 \ i_1
\]

where

$$\left[ e, \right] \alpha \cdot \pi \rho_1 \ i_1 = C[\left[ e \right]] \alpha \cdot \pi \rho_2 \ i_2;$$

$$\rho_2 = \lambda i \left\{ \begin{array}{ll}
\rho i & \text{if } i = [id]; \\
\rho & \text{otherwise};
\end{array} \right.$$

$$\rho_3 = \lambda i \left\{ \begin{array}{ll}
\rho i & \text{if } i = [id]; \\
\rho & \text{otherwise};
\end{array} \right.$$

Equation (C10) permits the construction of recursively defined lists.

For example, if

$$exp = \left[ \text{fix}: [1 \ a \ . \ 1] \right]$$

and the free variable is lambda bound, then

$$C[exp] (\pi \lambda \pi) (\pi \lambda \pi \lambda \pi) (\rho_1) = \left[ \text{fix}: [1 | \rho_1 (\pi \lambda \pi \lambda \pi)] \right]$$

and

$$C[exp] (\pi \lambda \pi) (\pi \lambda \pi \lambda \pi) (\rho_2) = \left[ \text{fix}: [1 | \rho_2 (\pi \lambda \pi \lambda \pi)] \right]$$

where

$$\rho_1 = (\pi \lambda \pi) (\pi \lambda \pi \lambda \pi);$$

$$\rho_2 = \lambda i \left\{ \begin{array}{ll}
\rho i & \text{if } i = [id]; \\
\rho & \text{otherwise};
\end{array} \right.$$
In the first example, the compiler can make only the first element of the output
strict because the bound on the number of versions is too small to permit it to
discover the loop. When it is allowed to create two more versions, it is able to make
the entire output value strict in every head, except for the second head, which was
explicitly omitted by the inherited pattern. A resource permitting the construction
of more than three versions would still produce the result from the second example.

\[
(C11) \quad C[f; e_1] \alpha \cdot \pi \rho \iota = \begin{cases} 
\text{Reached-Limit} & \text{if } (pa \alpha \pi) = \text{unbound} \& v\text{-count} \equiv \iota; \\
\text{Compile-Binding} & \text{if } (pa \alpha \pi) = \text{unbound} \& v\text{-count} < \iota; \\
\text{Mark-With-Pattern} & \text{otherwise;}
\end{cases}
\]

\[\text{where } \langle \langle \text{fix: } [f \lambda \text{id. body}] \rangle, pa, v\text{-count}, \text{fix} \rangle = \rho[f];\]

\[\text{Reached-Limit} \quad \langle \langle \text{fix: } [f \lambda \text{id. body}] \rangle : e \rangle \alpha \cdot \pi \rho \iota;\]

\[\text{Compile-Binding} \quad \langle \langle \text{fix: } [f \alpha \cdot \pi \lambda \text{id. body}] \rangle : e_1 \rangle \alpha \cdot \pi \rho_4 \iota \]

\[\text{where} \quad \langle \text{body} \rangle \alpha_1 \pi_1 \rho_1 \iota_1 = C[\text{body}] \alpha \cdot \pi \rho_2 \iota \]

\[\langle \langle \text{fix: } [f \lambda \text{id. body}] \rangle, pa_1, v\text{-count} + 1, \text{fix} \rangle \quad \text{if } i = [f];\]

\[\rho_2 = \lambda i \begin{cases} 
\rho i, & \text{if } i = [id]; \\
\rho_1 i, & \text{otherwise;}
\end{cases}
\]

\[pa_1 = \lambda pat. \begin{cases} 
\text{rec-p} & \text{if } pat = \alpha \cdot \pi; \\
p a & \text{otherwise;}
\end{cases}
\]

\[\rho_3 = \lambda i. \begin{cases} 
\rho i, & \text{if } i = [id] \text{ or } i = [f]; \\
\rho_1 i, & \text{otherwise;}
\end{cases}
\]

\[\text{rec-p} = \text{Pat-fun} (\rho_3 [id])\]

\[\langle e_1 \rangle \alpha_2 \cdot \pi_2 \rho_4 \iota_2 = C[e] \text{rec-p} \rho_3 \iota\]

\[\text{Mark-With-Pattern} \quad \langle f \alpha \cdot \pi : e_1 \rangle \alpha \cdot \pi \rho_1 \iota \]

\[\text{where} \quad \langle e_1 \rangle \alpha_1 \cdot \pi_1 \rho_1 \iota_1 = C[e](pa \alpha \cdot \pi) \rho \iota\]

There are three possible ways in which recursive function applications can be
compiled.

- If the version count for this particular function has been exhausted and the
  combination of this function call and the strictness pattern currently inherited has
  not been seen before, then the compiler expands the expression once, guaranteeing
  that the lazy call refers to the correct name, and stops exploring the source code.
- If the version count for this particular function has \textit{not} been exhausted and the
  combination of this function call and the strictness pattern currently inherited has
  not been seen before, then a new version is compiled. The current compile-time
  environment is updated so that the function mapping an inherited pattern to a
synthesized pattern for each version created so far now has an entry for this new version.

- Otherwise, the compiler is currently compiling a version whose compilation has already inherited the pattern propagated to the current function application. In this case, the synthesized pattern for this function is (circularly) present in the environment entry for this function and can be used in compiling the argument \[v]. See the previous examples.

\[(C12) \quad C]α \cdot π ρ v =
\begin{align*}
\text{Variable} & \quad \text{if } b\text{-type} = \text{lambda;} \\
\text{Reached-Limit} & \quad \text{if } (pa \cdot α) = \text{unbound} & v\text{-count} \geq i; \\
\text{Compile-Binding} & \quad \text{if } (pa \cdot α) = \text{unbound} & v\text{-count} < i; \\
\text{Mark-With-Pattern} & \quad \text{otherwise};
\end{align*}
\]
\[\text{where } ([binding], pa, v\text{-count}, b\text{-type}) = ρ[ id ];\]
\]

\[\text{Variable} \]
\[[(binding), \alpha \cdot π ρ_1 v]
\]
\[\text{where}
\]
\[ρ_1 = \lambda i. \begin{cases} ([binding], (α \cdot π pa) \cap_1 \Pi, 0, b\text{-type}) & \text{if } i = \|id\|; \\
\rho i & \text{otherwise}; \end{cases}\]

\[\text{Reached-Limit } ([binding]α \cdot π ρ_1 v;\]

\[\text{Compile-Binding } \]
\[[(\text{fix: } [id | α \cdot π e_1])α \cdot π ρ_3 v]
\]
\[\text{where}
\]
\[\text{fix: } [id e] = [binding];
\]
\[[(e_1)α \cdot π ρ_1 v_1 = C[e]α \cdot π ρ_2 v;\]
\[ρ_2 = \lambda i. \begin{cases} \text{fix: } [id e], pa_1, v\text{-count} + 1, \text{fix} & \text{if } i = \|id\|; \\
ρ i & \text{otherwise}; \end{cases}\]
\[pa_1 = \lambda \text{pat}. \begin{cases} \bot & \text{if } \text{pat} = α \cdot π; \\
pa & \text{otherwise}; \end{cases}\]
\[ρ_3 = \lambda i. \begin{cases} ρ i & \text{if } i = \|id\|; \\
ρ_1 i & \text{otherwise}. \end{cases}\]

\[\text{Mark-With-Pattern } [(id | α \cdot π)α \cdot π ρ v;\]

Identifiers may be recursively bound to a value, in which case they are treated like recursive data (but in cases which are similar to the discussion above) or they may be bound in a lambda expression, in which case the pattern currently being inherited is combined with the pattern accumulated by earlier compilation of instances of the identifier in a lambda body.

In the following example, the updated environments are included. The initial environment is \(ρ_1\), defined as follows;

\[ρ_1 = \lambda i. \text{unbound}[a1/a][b1/b][c1/c]\]
where

\[
\begin{align*}
a_1 &= [\text{nil}, \bot, 0, \lambda], \\
b_1 &= [\text{nil}, \bot, 0, \lambda], \\
c_1 &= [\text{nil}, \bot, 0, \lambda]. \\
\end{align*}
\]

(C7) \( \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \| \text{if: (a b c)} \)

\[
\begin{align*}
\rho_{2} &= \rho_{1}[a2/a], \\
\rho_{3} &= \rho_{2}[b2/b], \\
\rho_{4} &= \rho_{2}[c2/c], \\
\rho_{5} &= \rho_{2}, \\
a2 &= [\text{nil}, \bot, 0, \lambda], \\
b2 &= [\text{nil}, \bot, 0, \lambda], \\
c2 &= [\text{nil}, \bot, 0, \lambda].
\end{align*}
\]

Each time (C12) is called, the initial pattern, \( \bot \), is updated with the higher pattern, \( \bot \), which has an outer strictness mark preserved by the meet with the printer pattern. The environment formed by (C7) after both branches of the conditional have been compiled must take the meet of \( \bot \) and \( \bot \) for each of the variables \( b \) and \( c \), as each variable appears in only one branch.

2.4. Two extended examples

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

\[
\begin{align*}
\text{if: (eq?: (head:\text{x} 1))}
&\quad ((\text{fix: [Bad \text{x}. Bad:y]}):\text{head:\text{x}}.
\quad \text{F:(\text{tail:\text{x} . head:\text{x}}))})
\quad \langle\text{tail:\text{x} . head:\text{x}}\rangle)
\end{align*}
\]

\( F \) produces a stream of alternating 1's and \( \bot \). \( G \) selects odd elements of \( F \)'s result, avoiding the divergent elements. The compiler produces the following compiled expression, given \( \text{fix(\lambda x. (\text{tail:x} . head:x))} \), the initial environment \( \text{lid.unbound} \), and the rsource, \( S \):

\[
\begin{align*}
\langle G | p1:F | p2:(\text{\$1 . \$0}) \rangle
\end{align*}
\]

where

\[
\begin{align*}
\langle G | p1 = \lambda l. (\text{\$head:l} . G | p1:tail:tail:l) \rangle
\end{align*}
\]
and where the following strictness patterns identify the versions created;

\[ p1 = \text{fix}(\lambda \pi. (\pi, \pi)) ; \]
\[ p2 = \text{fix}(\lambda \pi. (\mathit{II}, (\perp, \pi))) ; \]
\[ p3 = \text{fix}(\lambda \pi. (\perp, (\mathit{II}, \pi))) ; \]
\[ p4 = \text{fix}(\lambda \pi. (\mathit{II}, (1, \pi))) ; \]
\[ \pi0 = \mathit{II} . \]

The versions of \( \mathcal{F} \) produce a stream that is alternately strict and lazy in its heads, and \( \mathcal{G} \) is strict in all elements it accesses, but produces a stream strict in all heads and lazy in the tails. Three patterns, those patterns which distinguish among versions of \( \mathcal{F} \), have

\[ \text{fix}(\lambda \pi. (\perp, (\perp, \pi))) \]

as their greatest lower bound. Such a lower bound indicates the improvement possible if only one version can be compiled for a function. If only one function body was to be compiled for \( \mathcal{F} \), then it would not be possible to make use of the versions that are strict in the heads of their values (when applied), since these can only be called as part of a mutually recursive cycle that also calls a version that is lazy in the head of its value. However, it is often possible to combine versions and retain much of their power. For example, versions \( \mathcal{F}|p2 \) and \( \mathcal{F}|p3 \) are identical and could be coalesced into one version.

\( \mathcal{G}|\pi0 \) produces a synthesized pattern that is \( \perp \). The effect of this pattern would be obvious if \( \mathcal{G}|\pi0 \) was applied to an expression using list syntax, such as \( \langle a . b \rangle \).

The following example calls a function that prints the even Fibonacci numbers, seeded with two values from anywhere in the series. Function formal arguments
are now destructed into a flat list of bound variables, however the corresponding
actuals are written as dotted pairs. The function odd? is treated as a primitive with
a single argument whose compilation propagates $\bot$ as its synthesized pattern if
the inherited pattern contains $. The variables a and b are free in the expression.

\[
\text{Skip}(h)\\text{where}\\text{Skip} = \\
\lambda [\text{stream}] . \\
\text{if: (odd?:head:stream} \\
\text{Skip:}\langle\text{tail:stream}\rangle \\
\langle\text{head:stream} . \text{Skip:}\langle\text{tail:stream}\rangle\rangle] \\
\text{h} = \langle a . b . \text{Addall:}\langle h . \langle \text{tail:h}\rangle\rangle\rangle] \\
\text{Addall} = \\
\lambda [c \ d] . \\
\langle\text{add:}\langle\text{head:c head:d}\rangle . \\
\text{Addall:}\langle\text{tail:c . \langle \text{tail:d}\rangle}\rangle\rangle] \\
\]

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

\[
\text{Skip}\vert p1 : \langle \text{h}\vert p2 \rangle\\text{where}\\text{Skip}\vert p1 = \\
\lambda [\text{stream}] . \\
\text{if: (odd?:head:stream} \\
\text{Skip}\vert p1 : \langle \text{tail:stream}\rangle \\
\langle\text{head:stream} . \text{Skip}\vert p1 : \langle\text{tail:stream}\rangle\rangle] \\
\text{h}\vert p2 = \langle a . \langle b . \text{Addall}\vert p1 : \langle \text{h}\vert p2 . \langle \text{tail:h}\vert p3 \rangle \rangle \rangle \rangle] \\
\text{h}\vert p3 = \langle a . \langle b . \text{Addall}\vert p1 : \langle \text{h}\vert p2 . \langle \text{tail:h}\vert p3 \rangle \rangle \rangle \rangle] \\
\text{Addall}\vert p1 = \\
\lambda [c \ d] . \\
\langle\text{add:}\langle\text{head:c head:d}\rangle . \\
\text{Addall}\vert \langle \text{tail:c . \langle \text{tail:d}\rangle}\rangle\rangle] \\
\]

and

\[
p1 = \text{fix}(\lambda \pi.\langle \bot, \pi \rangle) ; \\
p2 = \text{fix}(\lambda \pi.\langle \bot, \pi \rangle) ; \\
p3 = \text{fix}(\langle \bot, \text{p2} \rangle) . \\
\]

$\text{Skip}\vert p1$ is strict in all the heads of its argument, and passes this pattern to the
recursive data structure $\text{h}\vert p2$. $\text{Addall}\vert p1$ inherits the same pattern as $\text{Skip}\vert p1$. 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.

2.5. Representation of ⊥ₚ and pattern fixed points

The equations shown here are not directly executable. If the compiler is implemented using a conventional binding mechanism, then in analyzing the expression

\[(\text{fix:}[\text{Bad}\ \land\ n.\ \text{Bad}:n]):3\]

the compiler finds within equation (C9) that rec-ₚ is bound to "the value of rec-ₚ" and loops indefinitely. The implemented compiler avoids this by maintaining a table of pattern definitions, permitting it to detect such a binding. Every potential divergence must appear as a cyclic pattern of some kind, since only rational patterns are propagated. 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 rec-ₚ is ⊥ₚ.

Pattern fixed points are circularly constructed in (C9) and (C11). The fixed point rec-ₚ is initially represented by a distinguished pattern, interpreted as ⊥ₚ if its value is required before the entire fix expression has been compiled; this distinguished pattern is bound to the synthesized pattern in the table of strictness patterns when compilation of the lambda body is finished. Since the value of rec-ₚ is often not known when a recursive function application is compiled, the compiler currently makes another pass to annotate the argument of this application. (Subsequent passes may further improve the compiled code, although the examples presented here have been compiled using only two passes.)

This technique, described more formally elsewhere [13], does not produce very good approximations to pattern fixpoints for certain kinds of recursive function definitions, for example those using an accumulator to build a result; better techniques for finding fixpoints for such functions are presented elsewhere [12].

2.6. Termination of C

Strictness patterns can be represented by graphs of list structures suggested by their notation (angle brackets become parentheses, ⊥ₚ and S become distinguished tokens). These patterns are rational according to the following definition.

**Definition.** A **rational** strictness pattern is a finite strictness pattern, a cyclic pattern, or a pair whose components are rational patterns.

All finite patterns, such as (⊥, $⊥), can thus be represented by finite graphs. Infinite cyclic patterns, such as II, can be represented by a cyclic graph containing a pointer to the structure representing the repetition. Infinite **acyclic** patterns contain
at least one infinite pattern that is not cyclic, and so cannot be represented by either finite or cyclic graphs.

**Lemma 1.1.** The compiler propagates only rational patterns and compile-time environments containing rational patterns.

**Proof.** By structural induction on the compiler rules. There exists a finite representation for the initial inherited pattern, \( L' \), and the initial environment contains no entries. The desired property is that if a rule inherits a rational pattern \( (rp) \) and a compile-time environment containing only rational patterns \( (rce) \), then the rule returns an \( rp \) and \( rce \). Each rule returns the pattern it initially receives.

- \((C1, 2)\) cease compilation, returning the \( irp \) and \( irce \) (initial \( rp \) and \( rce \)).
- \((C3, 4)\) propagate acyclic finite patterns in which the \( irp \) is embedded; they are cyclic iff the \( irp \) is cyclic. They pass on the \( irce \) to the compilation of the function argument, returning an \( rce \) (by induction hypothesis (IH)).
- \((C5)\) either passes on the \( irp \) and \( irce \) to \((C2)\) or it distributes patterns that are components of the \( irp \) to the compilation of the function arguments. \((C5)\) then passes on the \( irce \) to compilation of the first argument and an \( rce \) (by IH) to the second, returning an \( rce \) (by IH).
- \((C6)\) sends a finite acyclic pattern and the \( irce \) to the compilation of the function argument, returning an \( rce \) (by IH).
- \((C7)\) propagates a finite acyclic pattern and the \( irce \) to the compilation of the predicate. The \( irp \) and an \( rce \) (by IH) are sent to the compilation of the branches. The returned compile-time environment is an \( rce \) as it combines two \( rces \) (by IH) so that only a pattern formed by taking the \textit{meet} of two patterns from the \( rces \) or a pattern function selected from an \( rce \) will appear in the returned compile-time environment.
- \((C8)\) sends the compilation of the function body the \( irp \) and an extension of the \( irce \) containing the rational pattern \( \bot \). It sends an \( rp \) (by IH) and an \( rce \) referring only to the \( irce \) or another \( rce \) (by IH) to the compilation of the function argument. An \( rce \) (by IH) is returned.
- \((C9)\) extends the \( irce \) with two new bindings. The first contains an initial pattern function with \textit{rec-p} for the \( irp \). The circular pattern \textit{rec-p} is defined both as the pattern to be finally synthesized by the compilation of the \textit{fix} expression and as the synthesized pattern to be passed up during compilation of a recursive call. Thus \textit{rec-p} may be written as the \textit{join} (or \textit{meet}) of \textit{rec-p} (inherited by the local lambda variable during compilation of a recursive call) and an \( rp \pi \) (by IH) inherited by the other instances of the local lambda variable within the \textit{fix} expression. This can be written as the rational expression

\[
\text{fix}(\lambda \textit{rec-p}. \pi \bot \textit{rec-p}).
\]

The only other \( irce \) extension contains the \( rp \bot \). \((C9)\) sends the \( irp \) and the extended \( irce \) to the compilation of the function body. The compilation of the function
argument receives \textit{rec-p} and a compile-time environment that selects either the \textit{ircp} or an \textit{rce} (by IH) and so is an \textit{rce}. (C9) returns an \textit{rce} (by IH).

(C10) sends the \textit{irp} and an extended \textit{ircp} containing an initial pattern function binding $\bot$ to the \textit{irp} to the compilation of the data recursion. The returned environment is an \textit{rce}, as it selects either the \textit{ircp} or an \textit{rce} (by IH).

(C11) may behave in one of three ways. It may cease compilation, returning the \textit{irp} and \textit{ircp}. It may behave much as rule 9 does, constructing a recursive pattern and extending a rational pattern function (by IH) by binding \textit{rec-p} to the \textit{irp}. Or it may retrieve an \textit{rp} (by IH) and propagate it, together with the \textit{ircp}, returning an \textit{rce} (by IH).

(C12) may behave in one of four ways. It may return an extended \textit{ircp} containing the meet of the rational printer pattern and the join of the \textit{irp} and an \textit{rp} (by IH). In two cases, it may cease compilation, returning the \textit{irp} and \textit{ircp}. Or, it may behave similarly to rule 10, extending a rational pattern function (by IH) so that it binds $\bot$ to the \textit{irp}.

\textbf{Lemma 1.2.} The compiler executes a finite number of rules.

\textbf{Proof.} Rules 1 through 10 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 11 and 12 do not construct a new version unless the resource permits, so a finite number of versions are introduced. Thus the compiler applies the rules finitely often.

\textbf{Theorem 1.} The compiler terminates.

\textbf{Proof.} Finite cyclic graphs may be compared or combined in finite time. Thus, meet, join, and environment-lookup all terminate. By Lemmas 1.1 and 1.2, the compiler terminates.

2.7. Compiler safety

Strictness analysis is a powerful technique precisely because it is not necessary to know whether a strict expression will evaluate to $\bot_S$. However, any implementation of a lazy list-processing language that improves performance only through strictness analysis risks some loss of semantic strength when printing a list that has a component which is $\bot_S$. The problem is that some element of a list may be $\bot_S$ and it might occur in a position that has been analyzed as “strict.”

Thus, an enveloping portion of the list may “diverge” (i.e. evaluate to $\bot_S$), even though the laziest possible implementation would be able to proceed beyond this point. In a simple case, $(\$ \bot_S . 1)$, this divergence causes the printing operation to lose even the outer left parenthesis, since the laziest possible printer can at least detect that its argument is a list and print a left parenthesis before attempting to
print the list’s contents. A more complicated case, such as $F:0 \Rightarrow (\lambda \cdot \text{inc}. n)$, where $F = \lambda [a \ b]. (\lambda b. (\lambda a. S())$, would cause the loss of the output prefix ‘(1’.

These observations prompt the following definitions of safety;
(1) Weak safety means that the interpretation of source and compiled code is equal when the interpretation of the source code is not $\bot$ and does not contain $\bot$.
(2) Strong safety means that the interpretation of source and compiled code produce the same element in the lattice of values.

Infinite trees 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 $\bot$. For example, a stream of natural numbers can be created by the following expression:

$$[F:0 \text{ where } F = \lambda n. (\lambda \cdot \text{inc}. n)]$$

It can also be created as follows:

$$[F:0 \text{ where } F = \lambda n. (\lambda F: \text{inc}. n \cdot n)]$$

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, loses the initial parentheses produced by traversal over a lazy expression. For these reasons, an admissible answer is defined. From this point on, a statement that “$C$ is safe” means that $C$ is safe when its interpreted source code produces admissible values.

If the user isn’t interested in seeing the preceding elements of a list that contains $\bot$, or in seeing an infinite series of left parentheses, then $C$ produces useful results even when its interpreted source code is not admissible.

2.7.1. Admissible values

Admissible values do not contain $\bot$. In addition, it is desirable to exclude values that contain an infinite series of left parentheses (assuming the printer makes a preorder traversal). The following definition identifies values that, when printed, contain an infinite unbroken sequence of left parentheses.

**Definition.** A tree $s$ is head-infinite if preorder traversal of $s$ requires traversal of an infinite number of head fields, without any tail fields, in some sub-tree of $s$.

Suppose a preorder traversal of a tree outputs an ‘$H$’ every time it traverses a list car, a ‘$T$’ every time it traverses a list cdr, and an ‘$A$’ each time it finds an atom. The traversal of a head-infinite tree would eventually produce an infinite sequence of ‘$H$’s.

**Definition.** A tree is head-finite if it is not head-infinite.

A tree that is head-finite contains no infinite sequence of left parentheses; if $\bot$ doesn’t occur in the tree, then the printer must eventually make progress and produce
output during its traversal of the value produced by interpretation of the compiler’s object code.

**Definition.** An *admissible* value does not contain \( \bot \) and is head-finite.

This definition permits certain kinds of infinite lists to appear in the head of a list, unlike the previous definition [11] which excludes them.

**Theorem 2.** \( C \) is safe.

**Proof.** We assume that the compiler identifies versions with the appropriate patterns and maintains the compile-time environment correctly. The proof is an induction on the safe propagation of the strictness patterns, the basis being that the printer pattern is safe (discussed above), and the initial rule called receives an initial compile-time environment in which no information is stored. The desired property is that if the pattern inherited by a rule safely approximates the printer’s demand and the compiler stores patterns safely, then it safely marks the source code and returns a safe compiler environment.

(C1) Constants can always be marked safely.

(C2) No strictness marks are introduced into the source code, as no further compiler rules are called and no marks are introduced by this one.

In the following cases, \( \alpha \cdot \pi \) contains at least one strictness mark.

(C3) There are two cases. Either \( \alpha \cdot \pi \) does or does not have an outer strictness mark. If \( \alpha \cdot \pi \) does have an outer strictness mark, then by the IH, the process evaluating this expression will evaluate the head of head’s argument, \( e \), to at least its outer structure. Since it must also access that list element, it is safe to embed the inherited pattern in the head of a list pattern that also has an outer mark. If \( \alpha \cdot \pi \) does not have an outer mark, no mark is added to the pattern propagated by the rule and so it is still safe.

(C4) Similar to the argument for (C3).

(C5) There are two cases. If \( \alpha \cdot \pi \) is not a list pattern, then it can only be \( \bot \) or \( \bot \), in which case it is compiled by (C2). If \( \alpha \cdot \pi \) is a list pattern, then by the IH, it is safe to propagate the head of the pattern to the compilation of the head of the source expression, \( e_1 \), and the tail of the pattern to the tail of the source expression, \( e_2 \). It is safe to mark the resulting compiled expressions, \( e_1 \) and \( e_2 \), with the outer marks appearing on the corresponding sub-patterns, because it is these expressions which produce the values required as indicated by \( \alpha \cdot \pi \).

(C6) There are two cases. If \( \alpha \cdot \pi \) has an outer mark, then by the IH, the process evaluating this expression will require its value, and so a pattern requiring both of the primitive’s arguments may safely be propagated to the compilation of \( \langle e_1 e_2 \rangle \). If \( \alpha \cdot \pi \) does not have an outer mark, then the following argument can be made. The printer pattern is specially constructed so that recursive patterns do not have their tails marked. If this had not been done and \( T_p \) had been used as the printer
pattern instead, then all patterns containing inner marks would also be marked on the outside. However, arithmetic and logical primitives do not return streams, and so it is safe to infer an outer mark in this particular case.

(C7) By an argument similar to that given for (C6), the compilation of the predicate expression $e_1$, which is not expected to evaluate to a stream, is given a safe pattern. Since $if$ will evaluate only one of its two branches, $e_2$ or $e_3$, the compiler guarantees that any strictness information exported from compilation of $if$ (contained in the compiler environment in the entries for lambda bound variables) is the *meet* of that received for each variable for which the two environments returned by compilation of the branches have entries. This is the only information exported (other than that specifying the construction of new versions) and since it is exported safely, it is reasonable to pass the inherited pattern on to the compilation of both branches, neither of which will be called before the predicate is evaluated.

(C8) Evaluation of the function body occurs once the actual has been substituted for the formal. Compilation propagates strictness in an equivalent manner by first compiling the function body and then propagating the combination of patterns inherited by the formal to the compilation of the actual. Thus, the compiler safely propagates the inherited pattern to the compilation of body. By the IH, the composite pattern returned by compilation of *body* and associated with the formal, *id*, may safely be used to compile the corresponding actual, *e*.

(C9) Similar to the argument made for (C8), except that the formal variable, *id*, may also inherit patterns from an inner application of the recursively defined function *f* (or a recursive function defined in a scope enclosing the definition of *f*). The solution to this circular equation, defined in Lemma 1.1 for rule (C9), is the lowest possible pattern in $P$ consistent with the compilation of the entire fix expression, and which thus introduces no unnecessary strictness marks. Currently, the implementation of C produces some approximation to this least fixpoint.

(C10) When evaluated, the result of the source expression *fix*: [id e] is that of its body, *e*, so the compiler safely propagates its inherited pattern to the compilation of the body.

(C11) There are three cases. In the first, compilation terminates without introducing marks into the code. The second is safe by an argument similar to that made for (C9). In the third case, the IH allows us to assume that $pa$ was correctly associated with *f* elsewhere. Thus it is safe to propagate it to the compilation of *f*'s argument.

(C12) There are four cases. In the first, as previously discussed, the compiler performs a traversal of a function application equivalent to that of the process executing the code, by first analyzing the function body and then propagating the pattern accumulated by this analysis to the compilation of the formal. The only point at which variables inherit patterns that cannot safely be joined is when they appear in the two branches of an *if*; this *meet* is taken at the appropriate point elsewhere. As *meet* and *join* are distributive, each pattern inherited by compilation of a particular variable can safely be joined to the pattern representing the current accumulation of patterns during the traversal of a function body. The resulting
pattern is safe, and is only weakened by the printer pattern. The third case is safe by an argument similar to that made for (C10). We have assumed versions are identified correctly, so the fourth case is safe as well. □

3. Conclusion

An extension of C has been implemented. Related research by Hughes [15, 17, 33], Wadler [33], and Burn [6], discussed in the following section, does not generate versions, and is based upon a more abstract approach to compilation.

3.1. Comparisons with other work

Hughes has developed a form of strictness analysis based upon contexts, which were initially described in an intuitive way [17], and then formalized as sets of continuations [15] using abstract interpretation. Contexts give the compiler the following information;

1. If all continuations of expression E fail to terminate, then the context of E is the empty set, and code can be inserted to abort the program;
2. If no continuation of E evaluates E, then the context of E is \{⊥S\};
3. If all continuations of E are strict, then E may safely be evaluated immediately;
4. If some continuations evaluate E and others do not, then a closure must be constructed for E, however the code evaluating this closure may use a context derived by removing ⊥S from the context of E.

Contexts are used to identify some expressions that will cause a program to abort, allowing a compiler to substitute an abort command, and some expressions that will not be required by a function at all, which can be replaced with a dummy expression. This is useful extra information that is not provided by the compiler presented here. Hughes does not explore the construction of versions, other than to assert that his approximation of recursive functions would guarantee that there will only be finitely many versions. The initial context he uses is similar to II and requires atoms or pairs with lazy tails. He rejects an initial context that maps any partial value to ⊥S. This makes sense if the user particularly wants to see output before the program loops. However, the compiler must then avoid marking some expressions which it might otherwise have marked, and the user might not want to pay this penalty in general.

Wadler recently formalized contexts as projections [33], a concept from domain theory. Wadler and Hughes create a finite domain of projections that is oriented differently from P; the top element specifies a context to be used when it is not known whether a given function is strict in its argument, and so the argument is left to be evaluated lazily. This lattice is excellent for analysing programs in which the same operation is performed upon every element in a list, commonly executed by a mapping function. However, as Abelson and Sussman point out [1], filters are
common and important functions. They do not necessarily treat each element of their arguments in the same way. And while this finite lattice could be extended to handle more complex patterns, it is created before the source code is compiled. There is a disadvantage to constructing a finite lattice without any reference to the structure of a particular program being compiled, as there may often be useful patterns that are excluded by such a process. For example, Wadler and Hughes's finite domain would not be able to describe strictness in the argument of G's initial application (the first of the extended examples presented in Section 2.4) without making a drastic approximation.

Burn [6] notes that it is important to consider the demand made upon a particular application of a given function when calculating the effect that application will have upon the evaluation of its argument. He labels a function application with a set of evaluation transformers, each of which maps a possible demand made upon the application to a set of corresponding demands made upon the function's arguments, using a finite domain of evaluators which treat the elements of a list in a uniform way. This approach is primarily intended to be used by a special architecture with hardware that maintains this information at run-time. However, he also asserts that "Evaluation transformers can be incorporated into a compiled implementation of functional languages. Most simply this can be done by having a case on the evaluator at the entry point of the code for a function. The code for each case initiates the evaluation of the argument expressions for which the evaluation transformers give a [strict] evaluator at that particular evaluator." [6, p. 463]. The question is, what is to be done with these evaluators at run-time, and the answer appears to be that the target machine must know how to use them. The target machine can be expected to contain representations of a finite number of evaluators, but not an infinite number, as would be required if this compilation technique was used with an infinite lattice of evaluators. In other words, a finite lattice must be established before the object code is analysed simply because the target machine's needs must be considered.

Here again, the domain of various objects representing different kinds of strictness information is determined before compilation takes place. However, it is not possible to know what kind of strictness exists in a particular program without analysing it. This problem is solved if the finite set of patterns required is generated lazily; that is, as the compiler examines the code and determines what is appropriate. C thus economically finds useful patterns that would otherwise be arbitrarily excluded, but pays a penalty for not searching a finite lattice when it comes to finding pattern fixpoints.

Fairbairn and Wray [8] discuss the use of versions in compiling higher order functions without list structures, also discussed in Wray's thesis [35]. They then go on to say that the use of versions "may lead to an unacceptable increase in code size" [8, p. 100], as does Hughes [15]. The work presented here demonstrates that there are interesting programs that would be greatly improved by versions, which do not increase the code size unacceptably in such cases. These cases suggest that
it is worthwhile refining the generation of versions, rather than rejecting their use because the worst case is impractical.

Assume that `map` is defined to produce an infinite list, so that the body of `map` is just the expression

\[
(\text{cons} (\text{function} (\text{head} \ \text{list})) (\text{map} \ \text{function} \ (\text{tail} \ \text{list}))).
\]

There are two possibilities. Either the call to `map` inherits a finite pattern, or it inherits a cyclic pattern. If the pattern is finite, then there is indeed a danger that a large number of versions of `map` may be produced; this would happen if the user gave the compiler a large resource and the finite pattern was a long one, allowing the compiler to unroll `map` several times. If the pattern is infinite, then the number of versions depends upon the cycle length of the pattern and the user's tolerance; either the compiler would be allowed to create a series of versions that referred to each other in a cycle, or it would create a smaller number of versions but fail to close the cycle. However, if the cycle is successfully closed, then versions take up a constant amount of space and avoid an unbounded number of suspensions, a number that may be very large when infinite lists are prominent data structures in the definition of the function.

### 3.2. Areas for future investigation

The ideal strictness compiler would produce many versions, subject to a reasonable resource, but would then coalesce versions according to certain criteria. It is possible that the same piece of code results from compilation of a function whose applications inherit a variety of strictness patterns. References to these versions should be compiled as references to only one distinct version. It may also be possible to develop techniques for selectively weakening versions when finer control over the tradeoff between space and time efficiency is required.

The compiler presented here does not permit fine-tuning, in the sense that it isn't possible to use one resource in producing versions of \( f \) and another to produce versions of \( g \). This is an interesting area for future research.

Another area in which more work needs to be done involves finding pattern fixed points. Currently, the compiler may be forced to propagate an unnecessarily lazy pattern when its search for a pattern fixed point fails. The technique outlined in Section 2.5 works very well and fails gracefully (safely), but without further work its power can't really be compared to that of other methods. However, it efficiently handles the examples presented here, and seems to be potentially a powerful technique.

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. This particular improvement can be easily added to the compilers presented here, and there are probably many more such. Also, it is sufficient to ensure that only cyclic patterns are bounded by \( \Pi \)—this is less restrictive than the constraint introduced by \( C \) on synthesized patterns.
It would be worthwhile investigating what happens when the notion of buffering is introduced. At present, $I^I$ can be seen as a buffer of length one, but it is worth considering an altered printer pattern that is strict in $n$ tails, but repeats in an unmarked tail. For example, the pattern

$$\text{fix}(\lambda \pi.((\pi, (\pi, (\pi, (\pi, (\pi, (\pi)))))))$$

has a buffer of length three. Buffering would allow more than just one element of a tree to be evaluated at a time, but would require that the user accept the loss of a buffer load of tree elements if any of those elements is $\perp_S$, a situation accepted by users of most conventional operating systems.

Interactive programs create special problems in strictness analysis, and while the techniques presented here may work well for such programs, they have not been designed with them in mind. This is another area for future research.

Preliminary experiments suggest that strictness analysis reduces space consumption in some cases, but several people have pointed out that it may be increased in others. More work must be done in this area before the additional constraints necessary when strictness analysis is used to compile programs are fully understood.

3.3. Contribution of research presented here

The work presented here is based upon several straightforward ideas that interact in a fruitful way. The domain of strictness patterns is expressive. Strictness in any list or sublist may be represented in the lattice $P$ of strictness patterns. $C$ is able to take advantage of this expressiveness without propagating patterns that would cause it to loop indefinitely. The result of a function need not be a list that is consumed in a homogeneous way in order for the compiler to produce an appropriate version.

The efficiency offered by versions in loops, especially loops that produce infinite trees, makes them worth exploring. The central loop in a program might be compiled into a cycle of twenty versions, permitting five suspensions to be avoided each time the loop is executed, causing an acceptable increase in code size simply because the speed of this loop is vital. However, when versions are not created, the meet of the patterns inherited by the set of applications of a given function is the best pattern above $\perp_P$ that can safely be used to compile the function. As has been shown, this pattern can be very weak even though each pattern inherited would have produced an efficient version, which produces especially poor object code for functions that inherit cyclic patterns and produce lists.

Once the domain of source expressions includes functions that produce infinite lists, it is possible to generate an infinite number of versions for any of these functions because the result of their application can be consumed in an infinite number of different ways. A compiler that terminates must decide upon a finite number of versions that it will introduce into a given program. The interesting question is—how will it generate these versions? One approach might be to create a set using brute
force, and then determine which, if any, will be useful. A better solution is found by $C$, which lazily creates versions on the fly when needed.

To summarize, versions combined with the expressive power of $P$ permit $C$ to receive and propagate patterns while avoiding unnecessary approximation to a significant extent, and to produce target code which fully profits from this information. Applications that require efficient construction of infinite trees, such as a functional operating system or circuit simulation, are especially likely to benefit.

Acknowledgement

We are grateful to John T. O'Donnell, who suggested the definitions of strong and weak safety, among other useful comments, and to Phil Wadler, who independently observed that the previous definition of admissible values was restrictive and made many useful suggestions. Steven D. Johnson has made several useful comments on earlier work.

References


