Fast Strictness Analysis Via Symbolic Fixpoint Iteration

Charles Consel
Research Report YALEU/DCS/RR-867
September 1991

This work is supported by the Darpa grant N00014-88-K-0573

Fast Strictness Analysis Via Symbolic Fixpoint Iteration

Charles Consel

Yale University
Department of Computer Science
New Haven, CT 06520
consel@cs.yale.edu

September 6, 1991

Abstract

Strictness analysis (at least for flat domains) is well understood. For a few years the main concern was *efficiency*, since the standard analysis was shown to be exponential in the worst case [5]. Thus lots of research evolved to find efficient average-case algorithms. In Yale Haskell we have implemented a new, fairly radical strictness analyzer that computes fixpoints via *symbolic manipulation of boolean functions.* This extremely simple approach also is extremely fast – the strictness analysis phase of our compiler typically takes about 1% of the overall compilation time.

1 Introduction

The goal of strictness analysis is to determine, for every function in a program, the parameters in which it is strict. Strictness information is crucial to the implementation of a non-strict language such as Haskell, since conventional machines are best suited to strict, or eager evaluation. Knowing that a function is strict in a given argument allows one to evaluate that argument eargerly and thus avoid creating delay structures, or "thunks."

Although theoretically well understood, the existing approaches to strictness analysis may be computationally expensive. This paper presents a new approach to strictness analysis based on "early fixpoint computation": a fixpoint of a recursive boolean

function is computed prior to considering actual abstract values. This can be viewed as performing a pending analysis as described in [5], but doing so *statically*. As a result, in practice, we have noticed that this technique requires much fewer iterations than conventional methods. The reader can verify this fact with the examples presented here.

An additional advantage of our symbolic strictness analysis is its great *simplicity* – it is easy to describe (indeed, it corresponds to a typical black-board description of the strictness analysis process), easy to prove correct, and easy to implement.

This paper is organized as follows. Section 2 introduces the approach with examples. Section 3 presents the algorithm. Section 4 discusses how strictness properties are used for code generation. Finally Section 5 assesses the method and proposes some future improvements.

2 The Approach

Usually strictness analysis is achieved through abstract interpretation (see [1, 7, 5], for example) using the two-point abstract domain $\{\bot, \top\}$ with ordering $\bot \sqsubseteq \top$. The idea is that if `f` is a function of three arguments (`x`, `y`, `z`), it is said to be strict in `x` if

$$f^{\#}\ \bot\ y\ z = \bot \quad \textit{for any } y \textit{ and } z$$

where, as is customary, the abstract version of function `f` is denoted by $f^{\#}$.

Let us first introduce our approach to discovering this property for non-recursive functions. Then, we investigate how to extend it to handle recursive functions.

2.1 Non-recursive Functions

One can think of symbolic strictness analysis as expressing the conditions under which a function fails to terminate. For this purpose we can use the value *True* for $\bot$ and *False* for $\top$. As a simple example, consider the following function.

```
f x y z = if x then y else z
```

The "termination condition" of `f` is $x \vee (y \wedge z)$. This boolean term can be read as follows: `f` fails to terminate if `x` fails to terminate or if both `y` and `z` fail to terminate. From these strictness conditions one can derive the extensional strictness behavior of `f`, that is,

$$f^{\#} \bot y z = \bot \quad \textit{for any } y\ z$$
$$f^{\#} x \bot \bot = \bot \quad \textit{for any } x$$

Following this idea, we can define a translation function *fail* that produces a *monotone boolean formula* representing the conditions under which an expression fails to terminate.

1. If k is a constant, then $fail\ \mathtt{k} = False$.
2. If x is a variable, then $fail\ \mathtt{x} = x$.
3. If p is a binary built-in function, strict in both its arguments, then

$$fail\ \mathtt{p}(e_1, e_2) = (fail\ e_1) \vee (fail\ e_2)$$

4. $fail\ (\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) = (fail\ e_1) \vee ((fail\ e_2) \wedge (fail\ e_3))$

where operators $\wedge$ and $\vee$ have the usual meanings.

$$x \wedge y = \begin{cases} True, & \text{if } x = y = True \\ False, & otherwise. \end{cases} \qquad x \vee y = \begin{cases} False, & \text{if } x = y = False \\ True, & otherwise. \end{cases}$$

2.2 Recursive Functions

Our strategy to handle recursive functions consists of performing statically a pending analysis. As such, it relies on the following theorem.

Theorem 1 *([5]) If, while evaluating $f(x)$ we find that it depends on the value of $f(x)$ again, returning $\perp$ as the result of the second (nested) call to $f(x)$ is correct with respect to the semantics of recursive monotone boolean functions.*

Pending analysis is implemented in a manner very similar to caching. In essence, a list of pending arguments is maintained. When the arguments of a function call already exists in the pending list, the value $\perp$ is returned. Notice that this approach aims at computing fixpoints in the presence of actual abstract values.

Our strategy goes one step further in that it abstracts the pending analysis technique from the abstract values: a fixpoint is computed on a recursive boolean function *prior to applying it to abstract values.* This is done as follows. Given a recursive boolean function $f^{\#}$, its body is evaluated without any strictness properties; at each recursive call $f^{\#}(\theta_1, \cdots, \theta_n)$ (where θ_i is a boolean term) a new instance [4] of $f^{\#}$ is created. When a given recursive call matches an already existing instance, it is replaced by $\perp$ (that is *True*). The correctness of this is ensured by Theorem 1 given above.

As an example consider the Factorial function with "accumulator".

```
fact (n, a) = if (n == 1) then a else fact((n - 1), (a * n))
```

Using the translation function $fail$ described earlier, the usual boolean algebraic laws, and the instantiation process outlined above, we have

$$fact^{\#}(n,a) = (n \vee \mathit{False}) \vee (a \wedge fact^{\#}((n \vee \mathit{False}),(n \vee a))) \quad (1)$$
$$fact^{\#}(n,a) = n \vee (a \wedge fact^{\#}(n,(n \vee a))) \quad \text{[Identity]}$$

$$fact^{\#}(n,(n \vee a))) = n \vee ((n \vee a) \wedge fact^{\#}(n,(n \vee (n \vee a)))) \quad (2) \quad \text{[Instant. and Subst.]}$$
$$fact^{\#}(n,(n \vee a)) = n \vee (a \wedge fact^{\#}(n,(n \vee a))) \quad \text{[Assoc. and Absorp.]}$$
$$fact^{\#}(n,(n \vee a)) = n \vee (a \wedge \mathit{True}) \quad (3) \quad \text{[Theorem 1]}$$
$$fact^{\#}(n,(n \vee a)) = n \vee a$$

$$fact^{\#}(n,a) = n \vee (a \wedge (n \vee a)) \quad \text{[Unfold.]}fact^{\#}(\mathrm{n},(\mathrm{n} \vee (\mathrm{n} \vee \mathrm{a})))$$
$$fact^{\#}(n,a) = (n \vee a) \quad \text{[Commut. and Idempot.]}$$

Notice that $fact^{\#}$ is initially instantiated with its paramaters (Line 1). In Line 2, a new instance of function $fact^{\#}$ is created with the values $(n,(n \vee a))$. In Line 3, a recursive call to an existing instance is replaced by value True (by Theorem 1). The final boolean term $(n \vee a)$ indicates that function `fact` is strict in both its parameters.

Let us now give the details of the algorithm.

3 The Algorithm

An important part of the algorithm is the process of transforming a boolean term into a canonical form. This is crucial to the instantiation mechanism: it ensures that a finite number of instances is created.

Let us examine the domain of boolean terms noted **BT**

$$\theta ::= \mathit{True} \mid \mathit{False} \mid x \mid \mathrm{And}(\theta_1,\cdots,\theta_n) \mid \mathrm{Or}(\theta_1,\cdots,\theta_n)$$

where x is an identifier of the program being analyzed.

For simplicity, we use n-ary boolean operators. Also, for a given or-term, we assume that, by associativity, inner or-terms are moved to the top level (similarly for and-terms).

Although the domain of boolean terms is composed of finite sets of values a term may grow infinitely for a given recursive call. To prevent from this, we define a

canonical form for boolean terms as follows: a boolean term is in canonical form if it is in *disjunctive normal form*, lexicographically ordered, and simplified. Let us detail each of these conditions.

We require a boolean term to be in disjunctive normal form. This eases considerably the simplification process because an or-term has a straightforward structure; it consists of boolean values, identifiers or and-terms.

The structure of the disjunctive normal form allows to define a total order on boolean terms; as detailled below, we shall use the lexicographic order, noted $\leq_{lex}$. This ordering makes it possible to determine equality on boolean terms.

$$
\begin{array}{l}
name(\theta_1) \leq_{lex} name(\theta_2) \\
where \\
\qquad\qquad name : \mathbf{BT} \rightarrow \mathbf{String} \\
\qquad\quad name(True) = True \\
\qquad\quad name(False) = False \\
\qquad\qquad name(x) = x \\
name(\mathrm{And}(\theta_1, \cdots, \theta_n) = concatenate(name(\theta_1), \cdots, name(\theta_n))
\end{array}
$$

The last component of the canonical form is the definition of simplification rules to ensure that infinite boolean terms cannot be constructed. These rules are based on the usual algebraic laws of the Boolean algebra. Notice that when a rule applies to both $\vee$ and $\wedge$, the notation $\stackrel{\circ}{\Longrightarrow}$ is used. Also, we assume that the rules are applied in a fixed order.

$$
Idempotence = \mathrm{Or}(\theta_1, \cdots, \theta_n) \stackrel{\circ}{\Longrightarrow} \mathrm{Or}(\theta'_1, \cdots, \theta'_m) \; s.t. \; \forall \, i,j \in \{1, \cdots, m\}, i \neq j \Rightarrow \theta'_i \neq \theta'_j
$$

$$
Identity = \begin{cases}
\mathrm{And}(\theta_1, \cdots, \theta_n) & \Longrightarrow \mathrm{And}(\theta'_1, \cdots, \theta'_m) & s.t. \; \forall \; i \in \{1, \cdots, m\}, \; \theta'_i \neq True \\
\mathrm{And}(\theta_1, \cdots, \theta_n) & \Longrightarrow False & if \; \exists \; i \in \{1, \cdots, n\} \; s.t. \; \theta_i = False \\
\mathrm{Or}(\theta_1, \cdots, \theta_n) & \Longrightarrow \mathrm{Or}(\theta'_1, \cdots, \theta'_m) & s.t. \; \forall \; i \in \{1, \cdots, m\}, \; \theta'_i \neq False \\
\mathrm{Or}(\theta_1, \cdots, \theta_n) & \Longrightarrow True & if \; \exists \; i \in \{1, \cdots, n\} \; s.t. \; \theta_i = True
\end{cases}
$$

The set of boolean terms in canonical form is noted $\mathcal{BT}$. The process of transforming a boolean term into its canonical form is noted *simpl*.

Let us now examine the algorithm displayed in Figure 3. For simplicity we assume that a program consists of a unique function; this restriction is lifted in the next section. Domain **Hist** captures the instantiation mechanism introduced in Section 2.2: it keeps track of the boolean terms with which the boolean function is called. This is represented as a set of call patterns. When a pattern already exists the corresponding call is replaced by value *True* (by Theorem 1).

1. Syntactic Domains

$c \in \mathbf{Const}$ [Constants]
$x \in \mathbf{Var}$ [Variables]
$p \in \mathbf{Po}$ [Strict Built-in Operators]
$f \in \mathbf{Fn}$ [Function Names]
$e \in \mathbf{Exp}$ [Expressions]

$$e ::= c \mid x \mid p(e_1, \cdots, e_n) \mid f(e_1, \cdots, e_n) \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

$$\mathbf{Prog} ::= \{f(x_1, \cdots, x_n) = e\}$$

2. Domains

$\theta \in \mathbf{BT}$ [Boolean Terms]
$\sigma \in \mathbf{Hist} \subseteq \mathcal{BT}$
$\rho \in \mathbf{Env} = \mathbf{Var} \rightarrow (\mathbf{BT} + [\mathbf{BT}^n \times \mathbf{Hist} \rightarrow \mathbf{BT}])$

3. Functions

$\mathcal{S}_{Prog} : \mathbf{Prog} \rightarrow (\mathbf{Hist} \times \mathbf{BT})$
$\mathcal{S} : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{FunEnv} \rightarrow \mathbf{BT}$

$$\mathcal{S}_{Prog}\,[\![\{ f(x_1, \cdots, x_n) = e\}]\!] = ((\rho\,[\![f]\!])(x_1, \cdots, x_n)\ \emptyset) \downarrow 1$$
$$\textit{where } \rho = \bot[(\lambda(\theta_1, \cdots, \theta_n).\ \lambda\sigma.\ (< \theta_1, \cdots, \theta_n > \in \sigma) \rightarrow \textit{True},$$
$$(\mathcal{S}\,[\![e]\!]\ (\bot[\theta_k/x_k])\ (\sigma \cup \{< \theta_1, \cdots, \theta_n >\}))/f]$$

$$\mathcal{S}\,[\![c]\!]\ \rho\ \sigma = < \sigma,\ \textit{False} >$$

$$\mathcal{S}\,[\![x]\!]\ \rho\ \sigma = < \sigma,\ \rho\,[\![x]\!] >$$

$$\mathcal{S}\,[\![p(e_1, \cdots, e_n)]\!]\ \rho\ \sigma = < \sigma_n,\ \mathrm{Or}(\theta_1, \cdots, \theta_n)) >$$
$$\textit{where } < \sigma_1,\ \theta_1 > = \mathcal{S}\,[\![e_1]\!]\ \rho\ \sigma$$
$$\vdots \qquad\qquad \vdots$$
$$< \sigma_n,\ \theta_n > = \mathcal{S}\,[\![e_n]\!]\ \rho\ \sigma_{n-1}$$

$$\mathcal{S}\,[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\ \rho\ \sigma = < \sigma_3,\ \mathrm{Or}(\theta_1,\ \mathrm{And}(\theta_2,\ \theta_3)) >$$
$$\textit{where } < \sigma_1,\ \theta_1 > = \mathcal{S}\,[\![e_1]\!]\ \rho\ \sigma$$
$$< \sigma_2,\ \theta_2 > = \mathcal{S}\,[\![e_2]\!]\ \rho\ \sigma_1$$
$$< \sigma_3,\ \theta_3 > = \mathcal{S}\,[\![e_3]\!]\ \rho\ \sigma_2$$

$$\mathcal{S}\,[\![f(e_1, \cdots, e_n)]\!]\ \rho\ \sigma = (\rho\,[\![f]\!])\ (\textit{simpl}(\theta_1), \cdots, (\textit{simpl}(\theta_n)))\ \sigma_n$$
$$\textit{where } < \sigma_1,\ \theta_1 > = \mathcal{S}\,[\![e_1]\!]\ \rho\ \sigma$$
$$\vdots \qquad\qquad \vdots$$
$$< \sigma_n,\ \theta_n > = \mathcal{S}\,[\![e_n]\!]\ \rho\ \sigma_{n-1}$$

Figure 1: The Algorithm

Extending the Algorithm

In this section we discuss some extensions to our approach to cope with any kind of recursive functions. To do so, we distinguish two classes of recursive functions: mutually and non-mutually recursive functions. This distinction is based on the information provided by the dependency analysis.

For non-mutually recursive functions, the strategy is to analyze functions in "lexical scoping order", that is, starting at the leaves of the dependency graph and working up. This strategy requires a minor modification of our algorithm: once a function is analyzed, the environment is extended with the resulting boolean function. Then, subsequent calls to this function will use this boolean function. As an example, consider again function `fact` whose boolean function is

$$fact^{\#}(n, a) = (n \vee a)$$

assume an inner function contains the call $fact^{\#}(x, y)$; this would produce the boolean term $(x \vee y)$.

For mutually recursive functions, the dependency analysis of the compiler groups together functions from the same strongly connected component. Thus, each function of a strongly connected component is analyzed separately and each recursive call to another function than the one currently analyzed is simply unfolded. Notice that, in fact, only mutually functions called from outside of the strongly connected component has to be analyzed.

4 Code Generation using Strictness Properties

After strictness analysis, strictness properties are used by the code generator to shift a lazy program (FLIC) into a strict one (Scheme). Delays are represented using the cell model [3]. Each delay contains a flag and either a thunk, before evaluation, or value after evaluation. Three lisp functions deal directly with delays: `force`, `delay`, and `forced-delay`. The `forced-delay` function is used to add the delay wrapper to an already evaluated object; the `delay` function does not evaluate its argument until forced. The `force` function cannot be applied to an object not created by `delay` or `forced-delay`. The `forced-delay` function is used instead of `delay` whenever the computation being delayed is simpler than the creation of the thunk.

Strictness information is used to determine the representation of each variable. Strict variables contain no delay cells; lazy variables are always bound to a delay structure. The code generator inserts the required conversions between strict and delayed values.

To allow strict arguments to be passed to functions, a dual entry point scheme is used. Dual entry points are a common optimization for fast calling of known functions [6]. The optimized entry makes use of uncurrying to receive multiple parameters and strictness analysis to receive evaluated parameters. A second entry point, the standard entry, is used for higher order calls and is significantly slower.

5 Assessment

In practice, the strictness analyzer represents a remarkably small overhead in the overall time of compilation. Typically, the strictness analysis phase represents 1% of the overall compile time. This small overhead is due to the following facts. Symbolic fixpoint is computed in very few steps. Indeed, recursive calls usually contain very few variables (that yield simple boolean terms), therefore, few instances will be created. This can be explained by the fact that Haskell is lexically scoped. Also, as discussed in [2] regarding relational analysis, recursive calls usually do not shuffle the variables; in our case this claim is sustained by the strongly typed nature of Haskell: it restricts the set of possible variables that may occur in a given argument of a function.

The main drawback of this approach is that it is restricted to flat domains; extensions to our method to handle non-flat domains are currently being investigated.

Finally, note that this symbolic process to compute the fixpoint is very general. For instance, few modifications are required to perform a binding time analysis. Theorem 1 is defined for the least element of the binding time domain.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] A. D. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1:147–164, 1988.

[4] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.

[5] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *ACM Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[6] S. L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *FPCA '89, 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, 1989.

[7] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *FPCA '87, 2th International Conference on Functional Programming Languages and Computer Architecture*, 1987.