Finding Fixpoints on Function Spaces

Jonathan Young and Paul Hudak

Research Report YALEU/DCS/RR-505
December 1986

This research was supported in part by National Science Foundation grant DCR-845145 and Department of Energy grant FG02-86ER25012.

Finding Fixpoints on Function Spaces

Jonathan Young and Paul Hudak

December 1986

Yale University
Department of Computer Science [1]

Abstract

Recent work in strictness analysis has shown that a natural solution to the first-order case reduces to the evaluation of a recursive monotone boolean function with certain arguments, which has been shown to be complete in deterministic exponential time in the number of arguments to the function being analyzed [6]. Clack & Peyton Jones have suggested an algorithm called *frontier analysis* which they claim has a much better running time in the average case [3]. This paper discusses several different algorithms, including an overview of frontier analysis and a completely different approach which we call *pending analysis*. The pending analysis algorithm is easy to understand and implement and can be shown to have very nice average-case behavior. Furthermore, this algorithm can be "memoized" with very little difficulty and permits "pessimizing" – finding a point above the least fixpoint – in a straightforward manner.

1 Introduction

Strictness analysis determines the termination properties of functions in a program. If a function will not terminate when some of the actual parameters to the function do not terminate, then certain optimizing transformations (e.g. converting call-by-need into call-by-value) will not change

[1]This research was supported in part by National Science Foundation grant DCR-845145 and Department of Energy grant FG02-86ER25012.

the semantics of the program. Several papers recently have shown how to infer an approximation to strictness of a function by means of ***abstract interpretation***. In particular, the program result is abstracted to the domain $\mathbf{2} = \{\bot, \top\}$, where $\bot \sqsubset \top$. The value $\bot$ in the abstracted domain is interpreted to mean that the program can be proved not to terminate, while the value $\top$ allows both termination and non-termination. In this context, the operators $\sqcap$ and $\sqcup$ compute the greatest lower bound (minimum) and least upper bound (maximum) of two arguments.[2] Then, constants are all mapped to $\top$ (they all terminate), while strict functions such as $+$ are mapped to $\lambda x\ y.x \sqcap y$ since they diverge if either argument diverges. The treatment of the conditional is more subtle ($\lambda p\ c\ a.p \sqcap (c \sqcup a)$) but straightforward - see [8,5,2] for the details.

Since the programming languages which are being analyzed usually include recursion, first-order strictness analysis reduces to the evaluation of recursive monotone boolean functions (RMBFs). Unfortunately, this can take as much as an exponential amount of time, where the exponent is the number of arguments to the function being analyzed [5]. Experimental implementations of strictness have shown, however, that in practice the exponential behavior is not achieved. Instead, for the vast majority of typical cases, there are compact non-recursive representations for the recursive strictness functions.

In other words, there is very strong evidence that there exist algorithms which evaluate strictness – i.e., which evaluate an RMBF at a given point – in reasonable time for typical programs. This paper presents several algorithms which attempt to take less than exponential time in the average case. (All are necessarily exponential in the worst case.) We present Clack & Peyton Jones' algorithm for frontier analysis [3,9] and our algorithm for pending analysis, and then show how to add memoizing and pessimizing to our algorithm.

It should be noted that we talk about "the average case behavior" of strictness analysis without attempting a formal definition. In the present context, the distribution of programs we are asked to analyze is not well understood nor well defined. We instead appeal to the reader's intuition

[2]These operators can also be interpreted as lattice *meet* and *join* and, with $\bot$=*false* and $\top$=*true*, as boolean *and* and *or*.

about the properties of "typical" programs.

Although this work was originally motivated from our previous work on strictness analysis and similar inferencing problems, it is clearly applicable to any situation requiring fixpoints on finite function spaces. Many problems in semantic program analysis have this property, and indeed recent work in generalizing attribute grammars to allow circularities [4] has provided a second formal framework (the first being abstract interpretation) within which such requirements arise. Note that even traditional flow analyses such as U-D chaining [1] are also examples of applications of the algorithms presented here.

2 Formal Problem Definition

In this section we will formalize the problem in mathematical terms.

Definition: $\mathbf{2}^n$ is the standard cartesian product of the domain $\mathbf{2}$ with itself n times, under the pointwise ordering.

It can easily be seen that the domain $\mathbf{2}^n$ is a lattice of size 2^n.

Definition: A recursive monotone boolean function (RMBF) f is specified by an equation $f(x_1, \ldots, x_n) = body$, where $body$ is an expression with syntax:

$$exp \ ::= \ \bot \mid \top \mid exp_1 \sqcap exp_2 \mid exp_1 \sqcup exp_2 \mid f(exp_1, \ldots, exp_n) \mid x_i$$

The desired semantics of an RMBF f is that f is the least fixpoint of its defining equation on the boolean domain $\bot \sqsubset \top$, with the standard interpretation of $\sqcap$ and $\sqcup$. That is, if we construct from $body$ the functional $G = \lambda f.\lambda x_1 \ldots x_n.body$ which maps from $(\mathbf{2}^n \rightarrow \mathbf{2}) \rightarrow (\mathbf{2}^n \rightarrow \mathbf{2})$, then f is the least fixpoint of G. One algorithm to find this fixpoint is to iterate G applied to Ω, the function which maps all arguments to $\bot$:

$$\begin{aligned} f^0 &= \Omega \\ f^1 &= G(\Omega) \\ &\vdots \\ f^i &= G(f^{i-1}) \\ &\vdots \end{aligned}$$

Clearly this sequence terminates at a unique fixpoint, since there are a finite number of such funtions and G is monotonic. It is not hard to prove that the fixpoint so constructed is in fact the least fixpoint of G (see [11], for example). We call this algorithm "pumping with bottom." Unfortunately, comparing f^i with f^{i+1} is NP-complete [6], so this is not a particularly efficient algorithm, being exponential even in the best case.

Definition: *RMBF evaluation*: Given an RMBF, $f(x_1, \ldots, x_n) = \mathit{body}$, and a set of arguments $\bar{x} =< x_1, \ldots, x_n >$, evaluate $f(x_1, \ldots, x_n)$.

Theorem: RMBF evaluation is complete in (deterministic) exponential time in the number of arguments [5].

The problem we address in this paper is to find an algorithm for RMBF evaluation which has efficient average-case behavior.

3 Frontier Analysis

Clack and Peyton Jones [3] have observed that for most programs, the fixpoint is reached in a small number of iterations and that it is usually a very simple function.

One can represent this function by representing all possible argument tuples as the lattice $\mathbf{2}^n$, and then labelling each point in the lattice with a result, either $\perp$ or $\top$. Since f is monotonic, all the $\top$s are above all of the $\perp$s, and Clack and Peyton Jones suggest that a compact representation for f is the boundary between the $\perp$ and $\top$ regions. They call this boundary the *frontier*. In particular, a ($\perp$-)frontier is a set of points such that (1) $f\ \bar{x} = \perp$ iff there is a $\bar{c}$ in the frontier such that $\bar{x} \sqsubseteq \bar{c}$, and (2) no two points in the frontier are comparable. (The dual notion of a $\top$-frontier is defined similarly.) Because there is a one-to-one correspondence between functions and frontiers, their algorithm just repeatedly determines the frontiers of $f^0, f^1, \ldots$ until there is no change from the previous iteration. If the frontier has not changed, then the function cannot have changed, and the fixpoint f has been found.

Since the objective is to avoid exponentiality in typical cases, they propose looking for the $\top$-frontier "in parallel" with the $\perp$-frontier. That is, they alternate steps in the construction of the $\top$-frontier and the $\perp$-frontier, and stop when one of the constructions terminates. This introduces a subtlety into their algorithm, since after each fixpoint iteration, the algorithm may extend the $\perp$-frontier downwards from the previous $\perp$-frontier but it must start looking for a $\top$-frontier from the bottom of the lattice again. (This is clear because G is monotonic. Where $f^i(\bar{x})$ is $\perp$, $f^{i+1}(\bar{x})$ could be $\top$, so the frontiers move monotonically downwards as we iterate towards the fixpoint.) The reader is referred to [9] for more details.

It can be seen without too much difficulty that computing the $\perp$-frontier of a function in fact corresponds to computing the simplified disjunctive normal form (DNF) for that function. Simplified DNF in this case means DNF in which all clauses include all variables, but some clauses are removed because they are covered by others. (A clause *covers* another clause if some of the positive literals in the first clause are negative literals in the second clause while the rest of the clauses are the same. Covered clauses can be deleted because we are restricted to monotonic functions.) Dually, a $\top$-frontier corresponds to a similarly simplified conjunctive normal form (CNF).

4 Pending Analysis

We now propose a very different algorithm which builds on the observation that it is safe to return $\perp$ from a call which is the second recursive call with the same arguments.

The "pumping with $\perp$" algorithm presented in section 1 in essence finds the least fixpoint from the "inside-out," and is guaranteed to terminate. If we were to use an "outside-in" strategy, however, we would find that it would not terminate for certain function/argument combinations. For example, consider $f\ x = f\ x$, an RMBF which we wish to apply to an argument, $\top$. The semantics above specifies that the answer should be $\perp$, but a naive outside-in strategy will simply loop. The traditional reason given for this "problem" is that $\perp$ is the value corresponding to non-termination, and the evaluation strategy that we have chosen simply refuses to terminate when the answer should be bottom. But we know that we can actually compute this bottom value explicitly, since we have a finite domain – in essence we need a way to "flatten" the two-element domain **2**. The idea behind pending analysis is to avoid non-termination by detecting circularities during RMBF evaluation.

In the rest of this discussion, when we refer to *evaluation*, we mean normal-order evaluation.

Definition: We say that $f\ \bar{x}$ *depends on* or *calls* $f\ \bar{y}$ if the (normal-order) evaluation of $f\ \bar{x}$ results in an expression in which $f\ \bar{y}$ is the leftmost, outermost redex.

Definition: While evaluating $f\ \bar{x}$, if $f\ \bar{y}$ occurs as a redex, we will call "returning v from $f\ \bar{y}$" the action of replacing $f\ \bar{y}$ with v.

Theorem: If, while evaluating $f\ \bar{x}$ we find that it depends on the value of $f\ \bar{x}$ again, returning $\perp$ as the result of the second (nested) call to $f\ \bar{x}$ is correct with respect to the semantics of RMBF.

Proof: (Informal) Suppose we do this, and the original call to $f\ \bar{x}$ returns $\perp$. Then we didn't really do any substitution for $f\ \bar{x}$ - it is $\perp$. Alternatively, suppose, after doing the substitution, the original call returned $\top$. In this case, even if we had returned $\top$ from the second call, the original call would still have been obliged to return $\top$, too, because all of the operators are monotonic.

There is some similarity between this algorithm and that implied by the "minimum function graph" semantics in [7]. It should also be noted that the above theorem does *not* hold on arbitrary function spaces. In particular, it only holds as stated for "flat" domains – domains with maximum chain length 1.[3]

We will now show formally that is is correct to return $\perp$ for $f\ \bar{x}$ while evaluating $f\ \bar{x}$. Let G be the functional of which f is the least fixpoint as defined above, and let $h[v/\bar{x}]$ denote the function which is the same as h except at $\bar{x}$, where it has the value v.

Theorem: Let m and n be arbitrary positive integers and $\bar{x} \in \mathbf{2}^n$. Then

$$G^m(G^n(\Omega))(\bar{x}) = G^m(G^n(\Omega)[\perp/\bar{x}])(\bar{x})$$

Proof: Clearly $\sqsupseteq$ holds. If

$$G^m(G^n(\Omega))(\bar{x}) = \perp,$$

[3]It is possible, however, to generalize the result to domains having a finite maximum chain length of k. In this case, we need to actually find the fixpoint of $\lambda v.$ "evaluate $f\ \bar{x}$, returning v for $f\ \bar{x}$ during this evaluation". This can be done iteratively in at most k steps.

then there is nothing to show. So we assume that

$$G^m(G^n(\Omega))(\bar{x}) = \top$$

Similarly, we can assume that

$$G^n(\Omega)(\bar{x}) = \top$$

Let j be the largest integer such that

$$G^j(\Omega)(\bar{x}) = \bot$$

Now, clearly $j \leq n$. So

$$G^j(\Omega) \sqsubseteq G^n(\Omega)$$

and since $G^j(\Omega)(\bar{x}) = \bot$, we also have

$$G^j(\Omega) \sqsubseteq G^n(\Omega)[\bot/\bar{x}]$$

But now

$$\top = G^m(G^j(\Omega))(\bar{x}) \sqsubseteq G^m(G^n(\Omega)[\bot/\bar{x}])(\bar{x})$$

so

$$G^m(G^n(\Omega)[\bot/\bar{x}])(\bar{x}) = \top$$

This shows that it is always correct to return $\bot$ for $f\ \bar{x}$ when evaluating $f\ \bar{x}$.

5 Memoizing and Pessimizing Pending Analysis

There are two obvious ways in which to implement pending analysis: by means of a list (table, etc.) of the arguments pending or by means of closures. However, if the fixpoint function is also to be memoized in a table of some sort, then it is not difficult to see that we can dispense with the table of pending arguments. Instead, we store $\bot$ in the table at those points where the function is pending *before* recursing, while storing the result in the table *afterwards*.

It should be clear to the reader that if we occasionally return $\top$ instead of evaluating some call $f\ \bar{y}$, then we will reach a result which is *above* the value of the least fixpoint, f at $\bar{y}$. This is especially important in strictness analysis, when any point above the least fixpoint is a "safe" point from which to begin program optimizations. Thus, when our algorithm is taking too long, we can at any time "pessimize" and produce an answer which is safe, although not optimal, and yet not totally pessimistic as would be the top element in the lattice.

6 Implementation and Analysis

Both of the above algorithms have been implemented in T, a dialect of Scheme[10]. A straightforward implementation of frontier analysis took about 150 lines of T code, while pending analysis took 9 lines for a functional implementation and 21 lines with caching. Preliminary testing has shown that pending analysis runs (cpu time) between 5 and 30 times as fast as frontier analysis, but more benchmarks are needed.

Since we concentrated on correct rather than highly optimized implementations, many optimizations are possible and desirable for both algorithms. In particular, a representation of the elements in the domain $\mathbf{2}^n$ which could compare two elements efficiently would speed up frontier analysis, as would an efficient representation of frontiers. Pending analysis, on the other hand, would benefit from an efficient table lookup package, or, when memoized, by storing the cache in a vector of length 2^n.

It is difficult to characterize precisely the runtime behavior of these two algorithms because we are interested in the average case. Still, since frontier analysis is actually constructing the simplified CNF and DNF for the function, it can be seen to behave best when these expressions are small. On the other hand, frontier analysis expects the fixpoint to be found in very few iterations – when this is not the case, frontier analysis will not perform well.

The chief advantage of pending analysis is that it evaluates no further down the call tree than absolutely necessary. On the other hand, it is not particularly efficient when the function to be evaluated makes lots of recursive calls but eventually "bottoms out," and does not take full advantage of the monotonicity of the function it is evaluating. This problem does not appear to be too severe in practice.

7 Conclusions

We have presented two algorithms for finding fixpoints for recursive monotone boolean functions. Both of them have a behavior which is exponential in the worst case, but we have argued that by catching and breaking recursive dependencies on pending computations, we can evaluate typical RMBFs in much less than an exponential amount of time on the average. In addition, we have shown how to use pending analysis to "pessimize" the result when the computation has proceeded too far, as well as how to gain even more effeciency by memoizing (at the expense of the memo storage).

8 Acknowledgements

I am indebted to Simon Peyton Jones for his assistance in understanding and implementing frontier analysis and for pointing out the correspondence between "pending" and minimal function graphs. I also thank Paul Hudak and the Wrestlers for their invaluable assistance in getting the details in this paper correct, and Neil Immerman for his help with the pending correctness proof.

References

[1] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[2] G.L. Burn, C.L. Hankin, and S. Abramsky. *The theory of strictness analysis for higher order functions*, pages 42–62. Springer-Verlag, 1985.

[3] C. Clack and S.L. Peyton Jones. Strictness analysis – a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49, Springer-Verlag LNCS 201, September 1985.

[4] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proc. SIGPLAN '86 Sym. on Compiler Construction*, pages 85–98, ACM, 1986.

[5] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 97–109, January 1986.

[6] P. Hudak and J. Young. *A set-theoretic characterization of function strictness in the Lambda Calculus.* Research Report YALEU/DCS/RR-391, Yale University, June 1985.

[7] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th Sym. on Prin. of Prog. Lang.*, pages 296–306, ACM, January 1986.

[8] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[9] S.L. Peyton Jones and C. Clack. *Finding fixpoints in abstract interpretation*, page to appear. Ellis Horwood, 1987.

[10] J.A. Rees, N.I. Adams, and J.R. Meehan. *The T Manual.* Technical Report 4th edition, Yale University, January 1984.

[11] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.