A Set-Theoretic Characterization of Function Strictness in the Lambda Calculus

Paul Hudak and Jonathan Young

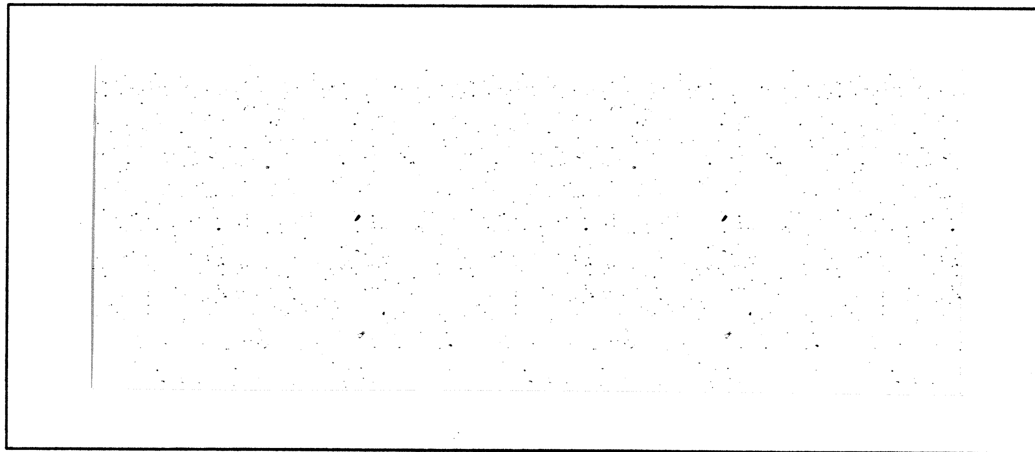
Research Report YALEU/DCS/RR-391
January 1985 (revised April 1985)

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520
Arpanet: hudak@yale, young@yale

This research was supported in part by NSF Grant MCS-8302018, and a Faculty Development Award from IBM.

Table of Contents

1. Introduction 1
2. Preliminaries 2
    2.1. Syntax 2
    2.2. Standard Semantics 3
        2.2.1. Standard Semantic Categories 3
        2.2.2. Auxiliary Semantic Functions 3
        2.2.3. Standard Semantic Functions 3
3. Introduction to Strictness Analysis 4
    3.1. A Naive Approach 4
    3.2. An Effective First-Order Solution 5
        3.2.1. Non-standard Semantic Categories (First-Order Strictness) 6
        3.2.2. Non-standard Semantic Functions (First-Order Strictness) 6
    3.3. Computing the Least Fixpoint 7
    3.4. A Boolean Characterization 8
    3.5. Correctness 9
4. An Extension to Handle High-Order Functions 9
    4.1. Preliminaries 10
    4.2. Strictness Ladders 10
        4.2.1. Non-standard Semantic Categories (High-Order Strictness) 11
        4.2.2. Non-standard Semantic Functions (High-Order Strictness) 11
    4.3. Computing the Least Fixpoint of Strictness Ladders 12
    4.4. Other Interpretations 13
    4.5. Comparison to First-Order Analysis 14
    4.6. A Final Example 15
5. Acknowledgements 16
I. NP-completeness of Inequivalence of Monotone Boolean Formulae (MF-INEQ) 17
II. Correctness Proofs for First-Order Strictness 18



YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

A Set-Theoretic Characterization of Function Strictness in the Lambda Calculus

Paul Hudak
Jonathan Young

Research Report YALEU/DCS/RR-391
January 1985 (revised April 1985)

Yale University
Department of Computer Science

Abstract

A function **f** is said to be *strict* in one of its formal parameters if it always evaluates the corresponding actual parameter whenever the function call returns a value (i.e., does not diverge). Detecting which arguments a function will surely evaluate is a problem that arises often in program transformation and compiler optimization. We present a strategy that allows one to effectively infer strictness properties of functions expressed in the lambda calculus. It is similar to work done by Mycroft [10], but with the following improvements: (1) a denotational semantics notation is used for abstract interpretation, (2) an intuitive set-theoretic characterization of the strict variables is used instead of boolean values, and (most importantly) (3) we extend the analysis to include high-order functions, using a unique inferencing strategy based on infinite sequences of functions. We also show that the most obvious strategy for computing the least fixpoint of the functional characterizing strictness is NP-complete, but that in practice, because the number of arguments to most functions is small, the complexity seems to be tractable.

This research was supported in part by NSF Grant MCS-8302018,
and a Faculty Development Award from IBM.

1. Introduction

A function **f** is said to be *strict* in one of its formal parameters if it always evaluates the corresponding actual parameter whenever the function call returns a value (i.e., does not diverge). More formally, a function $f(x_1,x_2,\ldots,x_n)$ is strict in x_i if $f(x_1,\ldots,x_{i-1},\perp,x_{i+1},\ldots,x_n) = \perp$ for all values of x_j, $j \neq i$. Detecting which arguments a function will surely evaluate is a problem that arises often in program transformation and compiler optimization. It is especially important in language implementations supporting normal-order evaluation (such as ALFL [4], SASL [11], and FEL [9]), where knowing that a function will evaluate a certain argument allows one to compute its value ahead of time, thus avoiding the overhead of a "closure," "self-modifying thunk" [5], "future," or some similar object. One can think of this as converting from a "call-by-name" or "call-by-need" evaluation strategy to one of "call-by-value." Another advantage of such an effort is that on parallel architectures it allows one to "eagerly" compute several arguments in parallel, with a potentially large reduction in overall execution time [6, 7].

In this paper we present a strategy that allows one to effectively infer strictness properties of functions in the lambda calculus. It is similar to earlier work by Mycroft [10], but with the following differences: (1) a denotational semantics notation is used for abstract interpretation, (2) an intuitive set-theoretic characterization of the strict variables is used instead of boolean values, and (most importantly) (3) we extend the analysis to include high-order functions, an extension that is crucial for any implementation of a programming language that treats functions as "first-class citizens." The inferencing strategy used for high-order functions is unique, and we believe it can be generalized to other domains. We also show that the most obvious strategy for computing the least fixpoint of the functional characterizing strictness is NP-complete, but that in practice, because the number of arguments to most functions is small, the complexity seems to be tractable.

2. Preliminaries

2.1. Syntax

Instead of adhering to the conventional syntax for the lambda calculus, we use a "syntactic sugaring" that has become quite popular in the functional programming community. This notation allows one to give *names* to functions by expressing them as a set of mutually recursive equations (and also, at least notationally, avoids the need for the **Y** operator). We write $f(x_1,x_2,...,x_n) = body$ to define the **n**-ary function **f** (equivalently, in unnamed form, $\lambda(x_1,x_2,...,x_n).body$). We assume that the language has been extended to include some standard set of primitive constants (including functions) such as those to support arithmetic. Generally function application is written as $f(e_1,e_2,...,e_n)$, where **f** may be either a user-defined function or primitive operator (informally we sometimes use infix notation for primitive functions, as in e_1+e_2 – the context should make the meaning clear). We also take as primitive a conditional operator, which permits expressions of the form $pred \rightarrow con,alt$, and is equivalent to the more traditional **if pred then con else alt**. Finally, we consider our program to be a set of mutually recursive function definitions (observing standard lexical scoping conventions as in the lambda calculus). We formalize all this in the following abstract syntax:

$c \in Con$ is the set of constants (such as integers and booleans).
$x \in Bv$ is the set of bound variables.
$f \in Fn$ is the set of function variables.
$p \in Pf$ is the set of primitive functions.
$e \in Exp$ is the set of expressions defined by:

$$e ::= c \mid x \mid e_1 \rightarrow e_2,e_3 \mid f(e_1,...,e_n) \mid p(e_1,...,e_n)$$

$pr \in Prog$ is the set of equation groups (programs) defined by:

$$\begin{aligned} pr ::= \{ \; & f_1(x_1,...,x_m) = e_1, \\ & f_2(x_1,...,x_m) = e_2, \\ & ... \\ & f_n(x_1,...,x_m) = e_n \; \} \end{aligned}$$

Note that without loss of generality we assume that each function has **m** arguments. Presumably the result of the program is the value of one of the functions being defined, such as the first or the last – this issue does not concern us here. Similarly, we assume that "nested" sets of equations can be handled through the obvious extensions, but for clarity we leave out the details.

Note also that an important restriction at this point is that function names may appear only in function application position; i.e., they may not be passed as arguments to functions nor returned as values from expressions. We relax this restriction completely in Section 4.

2.2. Standard Semantics

2.2.1. Standard Semantic Categories

Int	the standard flat domain of integers.
Bool	the standard flat domain of boolean values.
$\mathbf{Bas} = \mathbf{Int} + \mathbf{Bool}$	the domain of basic values.
$\mathbf{Fun} = \mathbf{Bas}^n \rightarrow \mathbf{Bas}$	the domain of first-order functions.
$\mathbf{D} = \mathbf{Bas} + \mathbf{Fun} + \{\mathbf{error}\}$	the domain of denoteable values.
$\mathbf{Env} = (\mathbf{Bv} + \mathbf{Fn}) \rightarrow \mathbf{D}$	the domain of environments.

2.2.2. Auxiliary Semantic Functions

Domain predicates: **Int?, Bool?, Bas?, Env?, and D?**.

Conditional:

$$e_1 \rightarrow e_2, e_3 = \begin{cases} \bot, & \text{if } e_1 = \bot \\ \mathbf{error}, & \text{if not Bool?}(e_1) \\ e_2, & \text{if } e_1 = \mathbf{true} \\ e_3, & \text{otherwise.} \end{cases}$$

Primitive functions:

$$\mathbf{x} + \mathbf{y} = \begin{cases} \bot, & \text{if } \mathbf{x} = \bot \text{ or } \mathbf{y} = \bot \\ \text{the sum of } \mathbf{x} \text{ and } \mathbf{y}, & \text{otherwise.} \end{cases}$$

$$\mathbf{x} \text{ AND } \mathbf{y} = \begin{cases} \bot, & \text{if } \mathbf{x} = \bot \text{ or } \mathbf{y} = \bot \\ \text{the logical AND of } \mathbf{x} \text{ and } \mathbf{y}, & \text{otherwise.} \end{cases}$$

and so on for other primitive functions.

2.2.3. Standard Semantic Functions

We adopt the convention of using double brackets $[\![\ldots]\!]$ around syntactic arguments.

K: **Con** → **Bas**, mapping syntactic constants to semantic values.
P: **Pf** → **Fun**, mapping primitive functions to semantic values.
E: **Exp** → **Env** → **D**, giving a meaning to expressions.
Ep: **Prog** → **Env**, giving a meaning to whole programs.

$$
\begin{aligned}
\mathbf{K}[\![\mathbf{n}]\!] &= \mathbf{n}, \text{ if } \mathbf{n} \text{ is an integer} \\
\mathbf{K}[\![\mathbf{true}]\!] &= \mathbf{true} \\
\mathbf{K}[\![\mathbf{false}]\!] &= \mathbf{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{P}[\![+]\!] &= \lambda(x,y).\ (\mathrm{Int?}\ x \text{ and } \mathrm{Int?}\ y) \rightarrow x + y,\ \mathrm{error} \\
\mathbf{P}[\![\mathbf{and}]\!] &= \lambda(x,y).\ (\mathrm{Bool?}\ x \text{ and } \mathrm{Bool?}\ y) \rightarrow x \text{ AND } y,\ \mathrm{error} \\
\mathbf{P}[\![\mathbf{cons}]\!] &= \lambda(x,y).\ \langle x,y \rangle
\end{aligned}
$$

and so on for other primitive functions.

$$
\begin{aligned}
&\mathbf{E}[\![c]\!]\,env = \mathbf{K}[\![c]\!] \\
&\mathbf{E}[\![x]\!]\,env = env[\![x]\!] \\
&\mathbf{E}[\![e_1 \rightarrow e_2,e_3]\!]\,env = \mathbf{E}[\![e_1]\!]\,env \rightarrow \mathbf{E}[\![e_2]\!]\,env,\ \mathbf{E}[\![e_3]\!]\,env \\
&\mathbf{E}[\![f(e_1,\ldots,e_2)]\!]\,env = env[\![f]\!](\mathbf{E}[\![e_1]\!]\,env,\ \ldots,\ \mathbf{E}[\![e_n]\!]\,env) \\
&\mathbf{E}[\![p(e_1,\ldots,e_2)]\!]\,env = \mathbf{P}[\![p]\!](\mathbf{E}[\![e_1]\!]\,env,\ \ldots,\ \mathbf{E}[\![e_2]\!]\,env)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{Ep}[\![\{\ & f_1(x_1,\ldots,x_m) = e_1, \\
& f_2(x_1,\ldots,x_m) = e_2, \\
& \ldots \\
& f_n(x_1,\ldots,x_m) = e_n\ \}]\!] = env' \\
& \textbf{whererec } env' = [\ \lambda(v_1,\ldots,v_m).\ \mathbf{E}[\![e_1]\!]\ env'[v_1/x_1,\ldots,v_m/x_m] \ /\ f_1, \\
& \qquad \ldots \\
& \qquad \lambda(v_1,\ldots,v_m).\ \mathbf{E}[\![e_n]\!]\ env'[v_1/x_1,\ldots,v_m/x_m] \ /\ f_n\ ]
\end{aligned}
$$

Note that the "meaning" of a program is an *environment* containing values for all of the top-level functions $\mathbf{f}_1$ through $\mathbf{f}_n$.

3. Introduction to Strictness Analysis

3.1. A Naive Approach

A naive approach to inferring function strictness can be described in the following way: The function f defined by $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) = \mathbf{body}$ is strict in $\mathbf{x}_i$ whenever the expression **body** is guaranteed to evaluate $\mathbf{x}_i$, according to the following (recursive) set of rules:

1. The evaluation of a parameter **x** always evaluates **x**.

2. The evaluation of $\mathbf{p}(\mathbf{e}_1, \ldots, \mathbf{e}_n)$ depends on **p**. For example, if it is a strict binary operator such as +, then both $\mathbf{e}_1$ and $\mathbf{e}_2$ are always evaluated; if it is a "sequential" operator such as **and**, then only $\mathbf{e}_1$ is always evaluated.

3. The evaluation of $\mathbf{e}_1 \rightarrow \mathbf{e}_2,\mathbf{e}_3$ always results in the evaluation of $\mathbf{e}_1$, and also all variables that are evaluated in *both* $\mathbf{e}_2$ and $\mathbf{e}_3$.

4. The evaluation of $\mathbf{f}(\mathbf{e}_1,\ldots,\mathbf{e}_n)$ always results in the evaluation of $\mathbf{e}_i$ whenever **f** is strict in its ith formal parameter.

Formalizing the above analysis for a set of function definitions results in a set of mutually recursive (because of the last rule) equations that can be solved in any number of ways. This is essentially the analysis carried out by Johnsson [8], except that he uses a boolean system that indicates whether or not a *particular* variable will be evaluated or not (his analysis thus requires solving a set of mutually recursive boolean equations instead of set equations).

Unfortunately, the above analysis has a fundamental difficulty, which becomes apparent when one considers the following simple program:

$$\{\ \mathbf{f(x,y,z)} = \mathbf{x{=}0 \rightarrow y,z} \\ \quad \mathbf{g(a,b)} = \mathbf{f(a,b,(b+1))}\ \}$$

Note that **g** is strict in both **a** and **b**, yet the simple analysis described earlier only detects that **g** is strict in **a**. This problem arises because even though we take into account the interactions between the consequent and alternate expressions in the conditional, *we fail to propagate that information across function boundaries*. Correcting this situation requires a nontrivial shift in the methodology, for now we must compute a function that characterizes the strictness property of each defined function *in terms of information describing what is evaluated in the arguments to the function*. The work by Mycroft [10] does exactly that, but using a boolean system similar to Johnsson's. In the next section we present a solution based on the more intuitive idea of functions on sets.

3.2. An Effective First-Order Solution

By reconsidering the problem from a set-theoretic standpoint, we can formalize our solution as follows: Intuitively, for any expression **e**, we let $N[\![\mathbf{e}]\!]$ denote the set of free variables which are "needed" to compute the value of **e**. Applying this notion intuitively to the above example means that $N[\![\mathbf{f(x,y,z)}]\!] = N[\![\mathbf{x \rightarrow y,z}]\!] = N[\![\mathbf{x}]\!] \cup (N[\![\mathbf{y}]\!] \cap N[\![\mathbf{z}]\!])$. Note carefully how the conditional is treated -- the intersection of the consequent and alternate comes from the fact that something evaluated in *both* subexpressions will be evaluated regardless of the value of the predicate (unless the predicate diverges, in which case the entire expression diverges). Continuing with the example, $N[\![\mathbf{g(a,b)}]\!] = N[\![\mathbf{f(a,b,(b+1))}]\!] = N[\![\mathbf{a}]\!] \cup (N[\![\mathbf{b}]\!] \cap N[\![\mathbf{b+1}]\!])$. Clearly, $N[\![\mathbf{b+1}]\!] = N[\![\mathbf{b}]\!]$, so $N[\![\mathbf{g(a,b)}]\!] = N[\![\mathbf{a}]\!] \cup N[\![\mathbf{b}]\!]$, as intuitively inferred.

We can formalize the above analysis by using *abstract interpretation* [1] of the original functions to derive a new set of mutually recursive equations that capture the properties of interest. Rather than give an "informal" abstract interpretation as suggested above, we simply provide an alternative, or "non-standard," semantics that captures the properties that we desire. Such a treatment requires that the function N have type $\mathbf{Exp} \rightarrow \mathbf{Senv} \rightarrow \mathbf{Sv}$, where **Senv** is an environment containing strictness properties of free variables, and **Sv** is the domain of sets of strict variables. More formally:

3.2.1. Non-standard Semantic Categories (First-Order Strictness)

Sv, domain of sets of strict variables.

Sfun = $\mathbf{Sv}^n \rightarrow \mathbf{Sv}$, the function space mapping sets of strict variables to other sets of strict variables.

Senv = $(\mathbf{Bv}+\mathbf{Fn}) \rightarrow (\mathbf{Sv}+\mathbf{Sfun})$, the "strictness" environment.

3.2.2. Non-standard Semantic Functions (First-Order Strictness)

Ps: **Pf** $\rightarrow$ **Sfun**, mapping primitive functions to a function that captures the strictness properties.

N: **Exp** $\rightarrow$ **Senv** $\rightarrow$ **Sv**, as intuitively described earlier.

S: **Prog** $\rightarrow$ **Senv**, giving a meaning to whole programs.

$\mathbf{Ps}[\![+]\!] = \lambda(\hat{x},\hat{y}).\ \hat{x} \cup \hat{y}$ (since + evaluates both of its arguments)

$\mathbf{Ps}[\![\mathbf{and}]\!] = \lambda(\hat{x},\hat{y}).\ \hat{x}$ (since "sequential and" always evaluates its first argument)

$\mathbf{Ps}[\![\mathbf{cons}]\!] = \lambda(\hat{x},\hat{y}).\ \emptyset$ (since "lazy cons" evaluates neither argument)

and so on for other primitive functions.

$$\mathbf{N}[\![c]\!]\,\mathbf{senv} = \emptyset$$
$$\mathbf{N}[\![x]\!]\,\mathbf{senv} = \mathbf{senv}[\![x]\!]$$
$$\mathbf{N}[\![e_1 \rightarrow e_2,e_3]\!]\,\mathbf{senv} = \mathbf{N}[\![e_1]\!]\,\mathbf{senv} \cup (\mathbf{N}[\![e_2]\!]\,\mathbf{senv} \cap \mathbf{N}[\![e_3]\!]\,\mathbf{senv})$$
$$\mathbf{N}[\![f(e_1,\ldots,e_n)]\!]\,\mathbf{senv} = \mathbf{senv}[\![f]\!](\mathbf{N}[\![e_1]\!]\,\mathbf{senv}, \ldots, \mathbf{N}[\![e_n]\!]\,\mathbf{senv})$$
$$\mathbf{N}[\![p(e_1,\ldots,e_n)]\!]\,\mathbf{senv} = \mathbf{Ps}[\![p]\!](\mathbf{N}[\![e_1]\!]\,\mathbf{senv}, \ldots, \mathbf{N}[\![e_n]\!]\,\mathbf{senv})$$

$$\begin{aligned}\mathbf{S}[\![\{\ & f_1(x_1,\ldots,x_m) = e_1,\\ & f_2(x_1,\ldots,x_m) = e_2,\\ & \ldots\\ & f_n(x_1,\ldots,x_m) = e_n\ \}]\!] = \mathbf{senv}'\\ & \textbf{whererec}\ \mathbf{senv}' = [\ \lambda(\hat{x}_1,\ldots,\hat{x}_m).\ \mathbf{N}[\![e_1]\!]\ \mathbf{senv}'[\hat{x}_1/x_1,\ldots,\hat{x}_m/x_m]\ /\ f_1,\\ & \qquad\qquad \ldots\\ & \qquad\qquad \lambda(\hat{x}_1,\ldots,\hat{x}_m).\ \mathbf{N}[\![e_n]\!]\ \mathbf{senv}'[\hat{x}_1/x_1,\ldots,\hat{x}_m/x_m]\ /\ f_n\ ]\end{aligned}$$

Thus the meaning of a program is still an environment, but now one that binds the top-level functions to elements of **Sfun**. In general we say that $\mathbf{senv}'[\![f]\!]$ is the *strictness function* of **f**, which we write in shorthand as $\hat{f}$. This corresponds to our use of $\hat{x}$ for bound variables in the "strictness domain," as a way of emphasizing that we are using a non-standard semantics.

Note that the environment **senv'** establishes the property that when $\hat{f}$ is applied to the sets corresponding to the strictness behavior of **f**'s arguments, it returns the set of variables "needed" by the application of **f** to those arguments. Note further that expressions like $+(x,y)$ get mapped to $\hat{x} \cup \hat{y}$, not $\{\hat{x},\hat{y}\}$. This emphasizes the fact that $\hat{x}$ and $\hat{y}$ are *sets*, and that $\hat{f}$ is a function on sets and is an abstract interpretation of **f** that describes its strictness properties.

3.3. Computing the Least Fixpoint

Consider the following recursive function:[1]

$$\begin{aligned} f(x,y,z,p,q) = p{>}0 \rightarrow\ & (p{=}1 \rightarrow x{+}z, \\ & \qquad f(z,z,0,p\text{-}1,x)\ ), \\ & f(0,0,z,1,y) \end{aligned}$$

It follows from the above definitions that:

$$\begin{aligned} \hat{f}(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) &= \hat{p} \cup ((\hat{x} \cup \hat{z}) \cap \\ &\qquad (\hat{p} \cup (\hat{f}(\hat{z},\hat{z},\emptyset,\emptyset,\hat{x}) \cap \hat{f}(\emptyset,\emptyset,\hat{z},\emptyset,\hat{y})\ ))) \\ &= \hat{p} \cup ((\hat{x} \cup \hat{z}) \cap \\ &\qquad \hat{f}(\hat{z},\hat{z},\emptyset,\emptyset,\hat{x}) \cap \hat{f}(\emptyset,\emptyset,\hat{z},\emptyset,\hat{y})\ ) \end{aligned}$$

From this a functional G can easily be defined such that:

$$\hat{f}(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) = G(\hat{f})(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}), \text{ or:}$$

$$\hat{f} = G(\hat{f})$$

so that $\hat{f}$ is a fixpoint of the functional G. One standard way of computing such a fixpoint is by constructing Kleene's ascending chain of "approximations," starting with the bottom element in the lattice of functions, and taking the least upper bound as the fixpoint. In our case the lattice of functions is formed based on the superset relation; that is:

$$\hat{x}_1 \sqsubseteq \hat{x}_2 \quad \text{iff} \quad \hat{x}_1 \supseteq \hat{x}_2$$
$$\text{and } f_1 \sqsubseteq f_2 \quad \text{iff} \quad f_1(\hat{x}) \sqsubseteq f_2(\hat{x}) \text{ for all } \hat{x}$$

which generalizes in the obvious way to n-ary functions. The superset relation is used because we wish to find as many strict variables as possible. The least defined element in Sv we denote by $\perp_{Sv}$, and it is simply the set of all strict variables of interest. Thus the least defined function $\cup_n$ in each n-ary function space simply returns $\perp_{Sv}$, but for convenience can be thought of as returning the union of its n arguments. Note that by construction, all of our functions are monotonic and continuous (because they are all constructed solely from set union and intersection), thus guaranteeing a unique least fixpoint.

Continuing with our example, we get the following ascending chain of refined estimates for the desired function (elements in the chain are identified by superscripting):

$$\begin{aligned} \hat{f}^0(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) &= \cup_5(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) \\ &= \hat{x} \cup \hat{y} \cup \hat{z} \cup \hat{p} \cup \hat{q} \end{aligned}$$

$$\begin{aligned} \hat{f}^1(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) &= \hat{p} \cup ((\hat{x} \cup \hat{z}) \cap (\hat{z} \cup \hat{x}) \cap (\hat{z} \cup \hat{y})) \\ &= \hat{p} \cup \hat{z} \cup (\hat{x} \cap \hat{y}) \end{aligned}$$

$$\begin{aligned} \hat{f}^2(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) &= \hat{p} \cup ((\hat{x} \cup \hat{z}) \cap (\hat{z} \cap \hat{z})) \\ &= \hat{p} \cup \hat{z} \end{aligned}$$

---

[1] This interesting example is due to Simon Peyton Jones.

$$\hat{f}^3(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) = \hat{p} \cup ((\hat{x} \cup \hat{z}) \cap \emptyset \cap \hat{z})$$
$$= \hat{p}$$

$$\hat{f}^4(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) = \hat{p} \cup ((\hat{x} \cup \hat{z}) \cap \emptyset \cap \emptyset)$$
$$= \hat{p}$$

Clearly $\hat{f}^i = \hat{f}^3$ for all $i > 3$, so $\hat{f}^3$ must be the least fixpoint.

In this example it was obvious when the least upper bound in the sequence was reached, but how is this done in general? In other words, how do we determine when $\hat{f}^i = \hat{f}^{i+1}$? This problem is the same as determining the equivalence of two monotone boolean formulae, since membership of an element in $\hat{f}(...)$ depends only on the "signature" of the parameters $\hat{x}$, ..., $\hat{q}$ (not, say, on the size of the sets, etc.). The boolean formulae are obtained by substituting logical-and for set-union, and logical-or for set-intersection. Unfortunately, determining such equivalence is NP-complete, as the analysis in Appendix I demonstrates. Note that this does *not* mean that determining these fixpoints in general is NP-complete, just that the iterative strategy described above is.

The obvious exponential algorithm to test for equality is to try all 2^n combinations of true (corresponding to non-empty) and false (corresponding to $\emptyset$) arguments in the derived boolean formulae. Because the functions in the chain are increasing in definedness, the 2^n combinations need to be tried only once. In most applications user-defined functions do not have a large number of arguments, and generally the number of arguments to individual functions does not increase with program size, so this method may be practical despite its exponential nature.

It would be convenient if we could look for a fixpoint when *applying* $\hat{f}^i$ to some given arguments, rather than finding a fixpoint in the functional itself. However, the above example shows that this is not the case: even though the sequence of functions $<\hat{f}^i>$ is *strictly increasing* until the fixpoint is reached, the sequence of sets $<\hat{f}^i(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q})>$ may not be. The example above was specifically designed to demonstrate this, using actual parameters $\hat{x} = \{x\}$, $\hat{y} = \{y\}$, $\hat{z} = \{z\}$, $\hat{p} = \{p\}$, and $\hat{q} = \{q\}$; i.e. for an "ordinary" application of **f**. For here we find that the sequence of approximations to the sets is $\{x,y,z,p,q\}$, $\{p,z\}$, $\{p,z\}$, $\{p\}$, $\{p\}$, ... -- note the "false summit" reached at $\{p,z\}$.

3.4. A Boolean Characterization

It should be noted that it is easy to convert our analysis to a boolean characterization similar to Mycroft's [10]. To do this, we would write **N' [[exp,x]] senv** as a predicate that returns true when **x** is a free variable that is "needed" in **exp**. In other words, we define **N'** by:

N' [[exp,x]] senv iff **x** ∈ N **[[exp]] senv**

Alternatively, the construction can be created directly by replacing set-union and set-intersection with logical-and and logical-or, respectively, changing the rule N **[[x]] senv = senv [[x]]** to **N' [[var,x]] senv = var** ∈ **(senv [[x]])**, and making similar minor changes to the other rules. The details are left to the reader.

Rather than use this boolean approach, we feel that it is intuitively more appealing to determine for an arbitrary expression **e** the set of *all* things that will be evaluated if **e** is. Indeed, by giving a unique name to each node in a parse-tree, we can use this same strategy to determine all *subexpressions* that will be evaluated. This may be very useful for compiler optimizations.

3.5. Correctness

In what sense is our algorithm correct? At a minimum, it should possess the following *safety property*: the analysis must never falsely declare that a function is strict in its **i**th argument. This is important, since presumably one of the primary reasons for doing the analysis is to allow compiler optimizations that might change the program semantics if the analysis were wrong. However, we *cannot* expect the converse to hold, since that would constitute a direct solution to the halting problem. That is, we cannot always expect the analysis to determine that a function is indeed strict in its **i**th argument. Our analysis is thus only an *approximation*, but that is the best that we can hope for.

Appendix II contains detailed correctness proofs for the following theorems:

Theorem 1: (Safety). Let $\mathbf{env'} = \mathbf{Ep}[\![\mathbf{p}]\!]$ and $\mathbf{senv'} = \mathbf{S}[\![\mathbf{p}]\!]$ for some program **p**:

$$\begin{aligned} \mathbf{p} = \{ \; & f_1(x_1,\ldots,x_m) = e_1, \\ & f_2(x_1,\ldots,x_m) = e_2, \\ & \ldots \\ & f_n(x_1,\ldots,x_m) = e_n \; \} \end{aligned}$$

Then $x \in \mathbf{senv'}[\![f_i]\!](\hat{x}_1,\ldots,\hat{x}_m) \Rightarrow \mathbf{env'}[\![f_i]\!](d_1,\ldots,d_m)=\bot,\ i=1,\ldots,n$
where $d_j=\bot$ **whenever** $x \in \hat{x}_j,\ j=1,\ldots,m$.

Although the proof of Theorem 1 is tedious, the reader is urged to read through it, since it demonstrates a classical combination of structural and fixpoint induction.

Theorem 2: (Termination). If $\bot_{Sv}$ is finite, then the standard iterative technique of determining $\mathbf{S}[\![\mathbf{p}]\!]$ always terminates in a finite number of steps, for all **p**.

4. An Extension to Handle High-Order Functions

Despite the simplicity and intuitive appeal of the analysis given so far, it is only applicable to first-order systems – note that we have not considered functions passed as parameters in function calls or returned as values from expressions (including function calls). For example, consider this simple program:

$$\begin{aligned} \{ \; & f(x) = x \\ & g(x) = x+1 \\ & h(a,b) = (a{=}0 \rightarrow f,g)\ b \; \} \end{aligned}$$

Our simple set theoretic construction is unable to detect the fact that **h** is strict in **b**, even though it is obvious to the reader that it is. Indeed, we have avoided this situation entirely by disallowing functions from appearing in other than function application position, which would rule out the way **f** and **g** are

used here (i.e., as the result of the conditional). We now remove that restriction entirely and present a new analysis that effectively deals with high-order functions of this sort.

4.1. Preliminaries

We now consider all functions to be "curried" (including primitive ones) and thus our function definitions look like: $\mathbf{f}\ \mathbf{x}_1\ \mathbf{x}_2\ \ldots\ \mathbf{x}_n = \mathbf{body}$. However, we use the more verbose lambda calculus notation $\mathbf{f} = \lambda \mathbf{x}_1.\ \lambda \mathbf{x}_2.\ \ldots\ \lambda \mathbf{x}_n.\ \mathbf{body}$ (or, equivalenty, $\mathbf{f} = \lambda \mathbf{x}_1\ \mathbf{x}_2\ \ldots\ \mathbf{x}_n.\ \mathbf{body}$), because it simplifies the inferencing rules. The one other change to the syntax is that we now allow *nested* groups of equations, and we take the value of the right-hand-side of the first equation as the value of the equation group. The new abstract syntax is thus:

$\mathbf{c} \in \mathbf{Con}$ is the set of constants, *including primitive functions*.
$\mathbf{x},\mathbf{f} \in \mathbf{V}$ is the set of bound and function variables.
$\mathbf{p} \in \mathbf{Pf}$ is the set of primitive functions.
$\mathbf{e} \in \mathbf{Exp}$ is the set of expressions defined by:

$$\mathbf{e} ::= \mathbf{c} \mid \mathbf{x} \mid \mathbf{f} \mid \mathbf{e}_1 \rightarrow \mathbf{e}_2,\mathbf{e}_3 \mid \mathbf{e}_1\ \mathbf{e}_2 \mid \lambda \mathbf{x}.\mathbf{e} \mid \mathbf{pr}$$

$\mathbf{pr} \in \mathbf{Prog}$ is the set of equation groups (programs) defined by:

$$\begin{aligned} \mathbf{pr} ::= \{\ & \mathbf{f}_1 = \mathbf{e}_1, \\ & \mathbf{f}_2 = \mathbf{e}_2, \\ & \ldots \\ & \mathbf{f}_n = \mathbf{e}_n\ \} \end{aligned}$$

4.2. Strictness Ladders

The important observation to be made is that an expression not only has a "direct strictness" (the set of variables which are evaluated when it is), but it also has a "delayed strictness" (the set of variables which are evaluated when the expression is *applied*). In fact, an expression has a doubly, triply, indeed "n-ly" delayed strictness, corresponding to the variables which will be evaluated when the expression is applied twice, three times, and n times, respectively. Thus, the *single* functions given by our previous analysis no longer suffice -- we now denote the strictness of an arbitrary expression as a *sequence of functions* called its *strictness ladder*.

More formally, with every expression **exp** in "strictness" environment **s** we associate a strictness ladder $\mathbf{L}[\![\mathbf{exp}]\!]\mathbf{s}$ that provides strictness properties of **exp** both as an isolated value and as a function to be applied. $\mathbf{L}[\![\mathbf{exp}]\!]\mathbf{s}$ is a sequence of functions of increasing arity -- we write $\mathbf{L}_i[\![\mathbf{exp}]\!]\mathbf{s}$ to denote the ith function in the sequence (starting with 0). Intuitively, $\mathbf{L}_0[\![\mathbf{exp}]\!]\mathbf{s}$ is a nullary function (i.e., a set of strict variables) that is the same as $\mathbf{N}[\![\mathbf{exp}]\!]\mathbf{s}$ in our previous analysis. But in addition we now have information that captures **exp**'s behavior as a function. In particular, $\mathbf{N}[\![\mathbf{exp}\ \mathbf{e}]\!]\mathbf{s} = \mathbf{L}_0[\![\mathbf{exp}]\!]\mathbf{s} \cup ((\mathbf{L}_1[\![\mathbf{exp}]\!]\mathbf{s})\ (\mathbf{L}[\![\mathbf{e}]\!]\mathbf{s}))$, since when evaluating a function call using normal-order reduction, one evaluates the function, then applies it to its argument. Similar results are obtained for repeated (i.e., curried) applications. Note that the *entire strictness ladder* $\mathbf{L}[\![\mathbf{e}]\!]\mathbf{s}$ is passed to $\mathbf{L}_1[\![\mathbf{exp}]\!]\mathbf{s}$, since we do

not know how **e** will be used within the body of **exp**; i.e., it could be used as a base value, applied to one argument, applied to two, etc. Similarly, the ith element in a strictness ladder is a curried function of i other strictness ladders. We formalize all this below (which the reader should compare to that given for the first-order case).

4.2.1. Non-standard Semantic Categories (High-Order Strictness)

Sv, domain of sets of strict variables.
Sfun = **Sv** + (**Sv** → **Sfun**) + {**serr**}, the function space mapping sets of strict variables to other sets of strict variables.
Slad = **Sv** × (**Slad** → **Sv**) × (**Slad** → **Slad** → **Sv**) × ... , the domain of strictness ladders.
Senv = **V** → **Slad**, the "strictness" environment.

For clarity we write $e_1 \cap_i e_2$ to mean $\lambda x_1\ x_2 \ldots x_i.\ (e_1\ x_1\ x_2 \ldots\ x_i) \cap (e_2\ x_1\ x_2 \ldots x_i)$. That is, $\cap_i$ distributes intersection over function application. By convention, $e_1 \cap_0 e_2$ is the same as $e_1 \cap e_2$. Finally, define $\bot_0$ = **serr**, and $\bot_i = \lambda x.\bot_{i-1}$, for $i>0$. $\bot_i$ is used in this way as an "error element," so we define $\bot_0 \cap e = e \cap \bot_0 = \bot$. Note by induction that this implies $\bot_i \cap_i e = e \cap_i \bot_i = \bot_i$.

4.2.2. Non-standard Semantic Functions (High-Order Strictness)

Kl: **Con** → **Sfun**, mapping constants (including functions) to **Sfun**.
L: **Exp** → **Senv** → **Slad**, mapping expressions to strictness ladders.
Sl: **Prog** → **Senv**, giving a meaning to a whole program.

$Kl[\![c]\!] = \langle \emptyset, \bot_1, \bot_2, \ldots \rangle$, if c is an integer or boolean value.
$Kl[\![+]\!] = \langle \emptyset, \lambda\hat{x}.\emptyset, \lambda\hat{x}\ \hat{y}.\hat{x}_0 \cup \hat{y}_0, \bot_3, \bot_4, \ldots \rangle$.
$Kl[\![=]\!] = Kl[\![+]\!]$.
$Kl[\![AND]\!] = \langle \emptyset, \lambda\hat{x}.\emptyset, \lambda\hat{x}\ \hat{y}.\hat{x}_0, \bot_3, \bot_4, \ldots \rangle$.
and so on for other primitive functions.

$$L[\![c]\!]s = Kl[\![c]\!]$$
$$L[\![x]\!]s = s[\![x]\!]$$
$$L[\![f]\!]s = s[\![f]\!]$$

$$L[\![e_1 \to e_2, e_3]\!]s = \langle L_0[\![e_1]\!]s \cup (L_0[\![e_2]\!]s \cap L_0[\![e_3]\!]s),\ L_1[\![e_2]\!]s \cap_1 L_1[\![e_3]\!]s,\ L_2[\![e_2]\!]s \cap_2 L_2[\![e_3]\!]s, \ldots \rangle$$

$$L[\![e_1\ e_2]\!]s = \langle L_0[\![e_1]\!]s \cup ((L_1[\![e_1]\!]s)\ (L[\![e_2]\!]s)),\ (L_2[\![e_1]\!]s)\ (L[\![e_2]\!]s),\ (L_3[\![e_1]\!]s)\ (L[\![e_2]\!]s), \ldots \rangle$$

$$L[\![\lambda x.e]\!]s = \langle \emptyset,\ \lambda\hat{x}.L_0[\![e]\!]s[\hat{x}/x],\ \lambda\hat{x}.L_1[\![e]\!]s[\hat{x}/x], \ldots \rangle$$

$$L[\![\{\ f_1 = e_1,\ f_2 = e_2,\ \ldots\ f_n = e_n\ \}]\!]s = L[\![e_1]\!]s'$$
$$\text{whererec } s' = s[\ L[\![e_1]\!]s' / f_1\ \ldots\ L[\![e_n]\!]s' / f_n\ ]$$

$$\begin{aligned}
\mathbf{Sl}[\![\{\ & f_1 = e_1, \\
& f_2 = e_2, \\
& \ldots \\
& f_n = e_n\ \}]\!] = s' \\
& \quad \mathbf{whererec}\ s' = [\ L[\![e_1]\!]s' / f_1 \\
& \qquad\qquad\qquad \ldots \\
& \qquad\qquad\qquad L[\![e_n]\!]s' / f_n\ ]
\end{aligned}$$

Thus the abstract "meaning" of a program is again a "strictness environment," but that now binds the top-level functions to *strictness ladders*. Similar to our earlier analysis, we refer to the strictness ladder of one of these functions as $\hat{f}$, and its ith element is $\hat{f}_i$.

As an example, consider the program given earlier, rewritten below in curried form:

$$\begin{aligned}
\{\ & f\ x = x \\
& g\ x = +\ x\ 1 \\
& h\ a\ b = ((=\ a\ 0) \rightarrow f, g)\ b\ \}
\end{aligned}$$

From this we derive the following (in which, for clarity, we omit the environment argument to L in all cases):

$$\hat{f} = L[\![\lambda x.x]\!] = \langle \emptyset, \lambda\hat{x}.\hat{x}_0, \lambda\hat{x}.\hat{x}_1, \ldots \rangle$$

$$\begin{aligned}
\hat{g} = L[\![\lambda x.+\ x\ 1]\!] &= \langle \emptyset, \lambda\hat{x}.L_0[\![+\ x\ 1]\!], \lambda\hat{x}.L_1[\![+\ x\ 1]\!], \ldots \rangle \\
&= \langle \emptyset, \lambda\hat{x}.\hat{x}_0, \lambda\hat{x}.\bot_1, \ldots \rangle
\end{aligned}$$

$$L[\![(=\ a\ 0) \rightarrow f,g]\!] = \langle \hat{a}_0 \cup (\hat{f}_0 \cap \hat{g}_0), \hat{f}_1 \cap_1 \hat{g}_1, \hat{f}_2 \cap_2 \hat{g}_2, \ldots \rangle$$

$$L[\![((=\ a\ 0) \rightarrow f,g)b]\!] = \langle \hat{a}_0 \cup (\hat{f}_0 \cap \hat{g}_0) \cup ((\hat{f}_1 \cap_1 \hat{g}_1)\hat{b}), (\hat{f}_2 \cap_2 \hat{g}_2)\hat{b}, (\hat{f}_3 \cap_3 \hat{g}_3)\hat{b}, \ldots \rangle$$

which, after substituting for $\hat{f}$ and $\hat{g}$, yields:

$$\begin{aligned}
L[\![((=\ a\ 0) \rightarrow f,g)b]\!] &= \langle \hat{a}_0 \cup \hat{b}_0, \hat{b}_1 \cap_1 \bot_1, \hat{b}_2 \cap_2 \bot_2, \ldots \rangle \\
&= \langle \hat{a}_0 \cup \hat{b}_0, \bot_1, \bot_2, \ldots \rangle
\end{aligned}$$

so finally:

$$\begin{aligned}
\hat{h} &= L[\![\lambda a\ b.((=\ a\ 0) \rightarrow f,g)b]\!] \\
&= \langle \emptyset, \lambda\hat{a}.\emptyset, \lambda\hat{a}\ \hat{b}.\hat{a}_0 \cup \hat{b}_0, \lambda\hat{a}\ \hat{b}.\bot_1, \lambda\hat{a}\ \hat{b}.\bot_2, \ldots \rangle \\
&= \langle \emptyset, \lambda\hat{a}.\emptyset, \lambda\hat{a}\ \hat{b}.\hat{a}_0 \cup \hat{b}_0, \bot_3, \bot_4, \ldots \rangle
\end{aligned}$$

Note that $\hat{h}_2$ indicates that the function h is strict in both of its arguments, as was intuitively inferred earlier.

4.3. Computing the Least Fixpoint of Strictness Ladders

Given the resulting set of equations defining the strictness ladders, the next question is how to find an appropriate solution; i.e., a least fixpoint. This would be a trivial task if the set of equations were finite, since then the standard iterative technique used in Section 3.2 would suffice. However, each equation in our new analysis represents an entire strictness ladder, which is an *infinite* sequence of functions.

Fortunately the elements of a ladder usually degenerate at some point to error functions of the form $\perp_i$. Suppose in such a situation the maximum length of the "relevant" part of each ladder is less than **n**. Then each equation $\hat{\mathbf{f}} = \mathbf{exp}$ can be replaced by **n** equations of the form $\{ \hat{\mathbf{f}}_0 = \mathbf{exp}_0, \hat{\mathbf{f}}_1 = \mathbf{exp}_1, \ldots, \hat{\mathbf{f}}_{n-1} = \mathbf{exp}_{n-1} \}$. In addition, occurrences of non-subscripted variables such as $\hat{\mathbf{f}}$ can be replaced by the n-tuple $<\hat{\mathbf{f}}_0, \hat{\mathbf{f}}_1, \ldots, \hat{\mathbf{f}}_{n-1}>$. If we then define the set of least defined functions as $\cup_i = \lambda \mathbf{x}_1\ \mathbf{x}_2 \ldots \mathbf{x}_i.\ \mathbf{x}_1 \cup \mathbf{x}_2 \cup \ldots \cup \mathbf{x}_i$, we can form an ascending chain of approximations as before, stopping when the least upper bound is reached.

Unfortunately, not all strictness ladders degenerate in the way described above. For example, consider the strictness ladder for the function $\mathbf{f} = \lambda \mathbf{x}.\mathbf{f}\ \mathbf{x}\ \mathbf{x}$:

$$\hat{\mathbf{f}} = <\emptyset,\ \lambda\hat{\mathbf{x}}.\hat{\mathbf{f}}_0 \cup \hat{\mathbf{f}}_1\ \hat{\mathbf{x}} \cup \hat{\mathbf{f}}_2\ \hat{\mathbf{x}}\ \hat{\mathbf{x}},\ \lambda\hat{\mathbf{x}}.\hat{\mathbf{f}}_3\ \hat{\mathbf{x}}\ \hat{\mathbf{x}},\ \ldots >^2$$

Note that each element in the ladder after the first is defined in terms of the *next* element! Clearly we cannot evaluate this by the methods described so far. This problem arises because **f** does not have a well-defined "depth" or "order" for its type, which appears to be $\mathbf{D} \rightarrow \mathbf{D} \rightarrow \mathbf{D} \rightarrow \ldots$.

We feel that functions such as the above have little utility (except as counterexamples!), and our approach to solving the problem is to disallow them from our domain of programs by a suitable typing discipline. In particular, most versions of the typed lambda calculus provides the necessary constraints, and schemata of this kind have been studied extensively elewhere ([2, 3]). In general it is convenient to disallow functions that have "infinite" types, such as **f** defined earlier and $\mathbf{f'} = \lambda \mathbf{x}.\ \mathbf{f'}$ which also has type $\mathbf{D} \rightarrow \mathbf{D} \rightarrow \mathbf{D} \rightarrow \ldots$[3] However, this restriction does not seem to always be necessary, because even though the ladder for **f'** does not degenerate to error elements as described earlier, there is no dependence of early elements on later elements as was the case for **f**. Thus the ladder may be computed "lazily" by demand.

After restricting our function space as described above, it is easy to show that all functions have a strictness ladder whose "relevant part" is finite. For, since the type of **f** is finite (say l), we can consider this to be the length of the ladder (an overestimate), since no element $\hat{\mathbf{f}}_i$, $0 \leq \mathbf{i} \leq l$, can depend on $\hat{\mathbf{f}}_k$, $\mathbf{k} > l$. With such a restriction we can now compute the least fixpoint as described earlier.

4.4. Other Interpretations

It should be noted that $\mathbf{L}[\![\mathbf{f}\ \mathbf{x}]\!] = <\hat{\mathbf{f}}_0 \cup (\hat{\mathbf{f}}_1\ \hat{\mathbf{x}}),\ \hat{\mathbf{f}}_2\ \hat{\mathbf{x}},\ \hat{\mathbf{f}}_3\ \hat{\mathbf{x}},\ \ldots >$ and thus:

$$\mathbf{L}[\![\lambda \mathbf{x}.\mathbf{f}\ \mathbf{x}]\!] = <\emptyset,\ \lambda\hat{\mathbf{x}}.\hat{\mathbf{f}}_0 \cup (\hat{\mathbf{f}}_1\ \hat{\mathbf{x}}),\ \lambda\hat{\mathbf{x}}.\hat{\mathbf{f}}_2\ \hat{\mathbf{x}},\ \ldots >$$
$$= <\emptyset,\ \lambda\hat{\mathbf{x}}.\hat{\mathbf{f}}_0 \cup (\hat{\mathbf{f}}_1\ \hat{\mathbf{x}}),\ \hat{\mathbf{f}}_2,\ \hat{\mathbf{f}}_3,\ \ldots >$$

However, by Eta-conversion $\lambda \mathbf{x}.\mathbf{f}\ \mathbf{x} \equiv \mathbf{f}$, yet:

---

[2]Since $\mathbf{L}[\![\mathbf{f}\ \mathbf{x}]\!] = <\hat{f}_0 \cup (\hat{f}_1\ \hat{x}),\ \hat{f}_2\ \hat{x},\ \ldots>$ and $\mathbf{L}[\![(\mathbf{f}\ \mathbf{x})\ \mathbf{x}]\!] = <\hat{f}_0 \cup (\hat{f}_1\ \hat{x}) \cup (\hat{f}_2\ \hat{x}\ \hat{x}),\ \hat{f}_3\ \hat{x},\ \ldots>$.

[3]It is interesting to note that **f** and **f'** are equivalent in the Beta-theory of the lambda calculus -- their Bohm trees are identical.

$$\mathbf{L}[\![\mathbf{f}]\!] = \hat{\mathbf{f}} = <\hat{\mathbf{f}}_0, \hat{\mathbf{f}}_1, \hat{\mathbf{f}}_2, \hat{\mathbf{f}}_3, \ldots >$$

So strictness is not preserved under Eta-conversion by our analysis. This reflects our interpretation of a lambda expression as a "thunk," whose body is not evaluated until it is called, and is manifested more seriously in examples such as **g x y = + x y**, which yields:

$$\mathbf{L}[\![\mathbf{g}]\!] = \hat{\mathbf{g}} = <\emptyset, \lambda\hat{x}.\emptyset, \lambda\hat{x}\ \hat{y}.\hat{x}_0 \cup \hat{y}_0, \bot_3, \bot_4, \ldots >.$$

For here note that $\mathbf{L}[\![\mathbf{g}\ \mathbf{e}_1]\!] = <\emptyset, \lambda\hat{y}.\mathbf{L}_0[\![\mathbf{e}_1]\!] \cup \hat{y}_0, \bot_2, \ldots >$, and thus $\mathbf{g}\ \mathbf{e}_1$ does *not* evaluate $\mathbf{e}_1$ (since $\mathbf{L}_0[\![\mathbf{g}\ \mathbf{e}_1]\!] = \emptyset$), even though $\mathbf{g}\ \mathbf{e}_1\ \mathbf{e}_2$ does.

Although this interpretation of strictness is reasonable, it is easy to imagine situations where one would want $\mathbf{g}\ \mathbf{e}_1$ to evaluate $\mathbf{e}_1$ whenever $\mathbf{g}\ \mathbf{e}_1\ \mathbf{e}_2$ does. Indeed, the only time one would normally evaluate $\mathbf{g}\ \mathbf{e}_1$ is when it is about to be applied, so evaluating $\mathbf{e}_1$ "early" would seem to be a safe thing to do. The only exception to this is in the use of predicates such as **function?** which might be expected to return **true** whenever its argument is a function. But since Eta-conversion is preserved in only limited and often differing ways in a given implementation, it might also be reasonable to define **function?** $(\mathbf{g}\ \bot) = \bot$ instead of **true**. We can alter our analysis to conform to this new interpretation by simply changing the definition of **L** when applied to lambda expressions, from:

$$\mathbf{L}[\![\lambda \mathbf{x}.\mathbf{e}]\!] = <\emptyset, \lambda\hat{x}.\ \mathbf{L}_0[\![\mathbf{e}]\!]\ , \lambda\hat{x}.\mathbf{L}_1[\![\mathbf{e}]\!], \ldots >$$

to:

$$\mathbf{L}[\![\lambda \mathbf{x}.\mathbf{e}]\!] = <(\lambda\hat{x}.\mathbf{L}_0[\![\mathbf{e}]\!])\hat{\emptyset}, \lambda\hat{x}.\mathbf{L}_0[\![\mathbf{e}]\!], \lambda\hat{x}.\mathbf{L}[\![\mathbf{e}]\!]_1, \ldots >$$

where $\hat{\emptyset}$ is the "null ladder" defined by $<\emptyset, \lambda x_1.\emptyset, \lambda x_1\ x_2.\emptyset, \ldots >$.

With this new interpretation, we arrive at the following for the function **g** defined above:

$$\mathbf{L}[\![\mathbf{g}]\!] = <\emptyset, \lambda\hat{x}.\hat{x}_0, \lambda\hat{x}\ \hat{y}.\hat{x}_0 \cup \hat{y}_0, \bot_3, \bot_4, \ldots >$$

so that $\mathbf{L}_0[\![\mathbf{g}\ \mathbf{e}_1]\!] = \mathbf{L}_0[\![\mathbf{e}_1]\!]$, which means that $\mathbf{e}_1$ *is* evaluated when $\mathbf{g}\ \mathbf{e}_1$ is. This result highlights the generality of the strictness ladder approach.

Note that a similar change (to preserve consistency in the way partial applications are treated) could be made to the primitive definition of + as given by **Kl**:

$$\mathbf{Kl}[\![+]\!] = <\emptyset, \lambda\hat{x}.\hat{x}_0, \lambda\hat{x}\ \hat{y}.\hat{x}_0 \cup \hat{y}_0, \bot_3, \bot_4, \ldots >$$

4.5. Comparison to First-Order Analysis

It should be obvious that our new analysis provides additional information that the old analysis does not. It is also the case that the new analysis does not lose any of the power of the old. In particular, suppose we have an uncurried function **f** of **n** arguments defined by $\mathbf{f}(\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n) = \mathbf{exp}$ and subject to the restrictions given in Section 3.2 (i.e., functions only appear in application position). We would like the strictness properties computed for it to be the same as for the curried function **f'** defined by $\mathbf{f'}\ \mathbf{x}_1\ \mathbf{x}_2\ \ldots\ \mathbf{x}_n = \mathbf{exp'}$, where **exp'** corresponds to **exp** except that all function applications are curried. In other words, we would like the set of "things" needed to evaluate an application of the uncurried function to be the same as those needed to evaluate the application of the corresponding curried function.

We can state this more precisely by first creating an induction hypothesis in which **se** and **se'** are two "strictness" environments whose bindings preserve the property in question for the first-order and high-order cases, respectively. Then what we wish to prove is:

Theorem 3: $\mathbf{N}[\![\mathbf{exp}]\!]\,\mathbf{se} = \mathbf{L}_0[\![\mathbf{exp'}]\!]\,\mathbf{se'}$

That is, the first element of the strictness ladder provides all of the information that the first-order analysis provided. A review of the definitions for N and L should convince the reader of this, although the details of the proof are tedious, and are omitted here.

4.6. A Final Example

The observant reader will have noted that we did not provide a strictness ladder for **cons** in the definition of Kl. It turns out that one can define **cons**, **car**, and **cdr** as high-order functions in the pure lambda calculus, and get the "lazy" behavior that we desire. Furthermore, it is an ideal test of our strictness analysis, since high-order functions get used in several ways. For example, we show below that our analysis is able to determine that the function **f**:

$$\mathbf{f\ p\ x\ y} = (\mathbf{p} \rightarrow \mathbf{car}, \mathbf{cdr})\ (\mathbf{cons\ x}\ (+\ \mathbf{x\ y}))$$

is strict in **p** *and* **x**!

First define **cons**, **car**, and **cdr** by:

$$\mathbf{cons\ x\ y\ g} = \mathbf{g\ x\ y}$$
$$\mathbf{car\ a} = \mathbf{a}\ (\lambda \mathbf{x\ y.\ x})$$
$$\mathbf{cdr\ a} = \mathbf{a}\ (\lambda \mathbf{x\ y.\ y})$$

Continuing to permit ourselves to be slightly "loose" with the notation, let:

$$\mathbf{lx} = \mathbf{L}[\![\lambda \mathbf{x\ y.\ x}]\!] = <\emptyset, \lambda\hat{x}.\emptyset, \lambda\hat{x}\ \hat{y}.\ \hat{x}_0, \lambda\hat{x}\ \hat{y}.\ \hat{x}_0, \ldots>$$
$$\mathbf{ly} = \mathbf{L}[\![\lambda \mathbf{x\ y.\ y}]\!] = <\emptyset, \lambda\hat{x}.\emptyset, \lambda\hat{x}\ \hat{y}.\ \hat{y}_0, \lambda\hat{x}\ \hat{y}.\ \hat{y}_0, \ldots>$$

Then the strictness ladders for **cons**, **car**, and **cdr** are:

$$\mathbf{L}[\![\mathbf{cons}]\!] = \widehat{\mathbf{cons}} = <\emptyset, \lambda\hat{x}.\emptyset, \lambda\hat{x}\ \hat{y}.\emptyset, \lambda\hat{x}\ \hat{y}\ \hat{g}.\ \hat{g}_0 \cup (\hat{g}_1\ \hat{x}) \cup (\hat{g}_2\ \hat{x}\ \hat{y}), \lambda\hat{x}\ \hat{y}\ \hat{g}.\ \hat{g}_3\ \hat{x}\ \hat{y}, \ldots>$$
$$\mathbf{L}[\![\mathbf{car}]\!] = \widehat{\mathbf{car}} = <\emptyset, \lambda\hat{a}.\ \hat{a}_0 \cup (\hat{a}_1\ \mathbf{lx}), \lambda\hat{a}.\ \hat{a}_2\ \mathbf{lx}, \ldots>$$
$$\mathbf{L}[\![\mathbf{cdr}]\!] = \widehat{\mathbf{cdr}} = <\emptyset, \lambda\hat{a}.\ \hat{a}_0 \cup (\hat{a}_1\ \mathbf{ly}), \lambda\hat{a}.\ \hat{a}_2\ \mathbf{ly}, \ldots>$$

Returning now to the original program, it should be clear that:

$$\mathbf{L}[\![\mathbf{cons\ x}\ (+\ \mathbf{x\ y})]\!] = <\emptyset, \lambda\hat{g}.\ \hat{g}_0 \cup (\hat{g}_1\ \hat{x}) \cup (\hat{g}_2\ \hat{x}\ \mathbf{x\hat{u}y}), \lambda\hat{g}.\ \hat{g}_3\ \hat{x}\ \mathbf{x\hat{u}y}, \ldots>$$

where $\mathbf{x\hat{u}y} = <\hat{x}_0 \cup \hat{y}_0, \ldots>$

and furthermore:

$$\mathbf{L}[\![\mathbf{p} \rightarrow \mathbf{car},\mathbf{cdr}]\!] = <\hat{p}_0 \cup (\hat{car}_0 \cap \hat{cdr}_0),\ \hat{car}_1 \cap_1 \hat{cdr}_1,\ \hat{car}_2 \cap_2 \hat{cdr}_2,\ ...>$$

$$= <\hat{p}_0,\ \hat{car}_1 \cap_1 \hat{cdr}_1,\ \hat{car}_2 \cap_2 \hat{cdr}_2,\ ...>$$

$$= <\hat{p}_0,\ \lambda \hat{a}.\hat{a}_0 \cup ((\hat{a}_1\ lx) \cap (\hat{a}_1\ ly)),\ ...>$$

Combining the two using the rule for function application finally yields:

$$<\hat{p}_0 \cup (\ (lx_0 \cup (lx_1\ \hat{x}) \cup (lx_2\ \hat{x}\ x\hat{u}y)\ ) \cap (ly_0 \cup (ly_1\ \hat{x}) \cup (ly_2\ \hat{x}\ x\hat{u}y)\ )\ ),\ ...$$

$$= <\hat{p}_0 \cup (\ \hat{x}_0 \cap x\hat{u}y_0\ ),\ ...>$$

$$= <\hat{p}_0 \cup \hat{x}_0,\ ...>$$

(Whew!)

5. Acknowledgements

We wish to thank Professor Michael Fischer for the details of the proof of NP-Completeness contained in the Appendix. Also many thanks to Simon Peyton Jones, whose enlightening visit inspired us to pursue this topic further. Finally, the National Science Foundation provided partial support for this research under grant MCS-8302018.

I. NP-completeness of Inequivalence of Monotone Boolean Formulae (MF-INEQ)

Proof by reducing SAT to MF-INEQ. (Credited to Mike Fischer)

Let F be a CNF formula over variables $x_1, \ldots, x_n$. Construct a monotone formula F' over 2n variables by replacing each literal $\neg x_i$ in F by a new variable y_i. Then $a_1, \ldots, a_n$ is a satisfying assignment for F iff $a_1, \ldots, a_n, \neg a_1, \ldots, \neg a_n$ is a satisfying asssignment for F'. (Note that F' may have other satisfying assignments not of this form.) Now let:

$$A = (x_1 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$$
$$B = (x_1 \vee y_1) \wedge \ldots \wedge (x_n \vee y_n)$$
$$G1 = A \vee (B \wedge F')$$
$$G2 = A$$

We claim that G1 is inequivalent to G2 iff F is satisfiable, completing the reduction of SAT to MF-INEQ. To see this, define three properties of assignments:

1. Low. For some k, $x_k = y_k = 0$.
2. High. For some k, $x_k = y_k = 1$.
3. Good. For all k, $x_k \neq y_k$.

Clearly every assignment that is neither low nor high is good. The remarks above show that if F' is satisfied by a good assignment, then F is satisfiable.

Now, for any assignment, B=0 iff the assignment is low, and A=1 iff the assignment is high. Hence, G1 agrees with G2 on any low or high assignment. For any good assignment, A=0 and B=1, so G1 = F' and G2 = 0. Hence, G1 agrees with G2 on all assignments iff they agree on all good assignments iff F' is not satisfiable.

II. Correctness Proofs for First-Order Strictness

To aid the proofs that follow, we need to more precisely define the environments **env'** and **senv'** as fixpoints of the **whererec** clauses. These environments should be viewed as vectors of functions that satisfy the respective systems of mutually recursive equations. Therefore we can talk about solutions to the system as instances of these environments. Considering the system as a whole, let **<env,senv>** be one such solution. Then starting with the program:

$$\{\ f_1(x_1,\ldots,x_m) = e_1,$$
$$f_2(x_1,\ldots,x_m) = e_2,$$
$$\ldots$$
$$f_n(x_1,\ldots,x_m) = e_n\ \}$$

we define the functional **Tau** by:

$$\mathbf{Tau}\ \langle env_a, senv_a\rangle = \langle env_b, senv_b\rangle, \text{ where:}$$

$$env_b = [\ \lambda(v_1,\ldots,v_m).\ E[\![e_1]\!]\ env_a[v_1/x_1,\ldots,v_m/x_m]\ /\ f_1,\ \ldots,\ \lambda(v_1,\ldots,v_m).\ E[\![e_n]\!]\ env_a[v_1/x_1,\ldots,v_m/x_m]\ /\ f_n\ ]$$

$$senv_b = [\ \lambda(\hat{x}_1,\ldots,\hat{x}_m).\ N[\![e_1]\!]\ senv_a[\hat{x}_1/x_1,\ldots,\hat{x}_m/x_m]\ /\ f_1,\ \ldots,\ \lambda(\hat{x}_1,\ldots,\hat{x}_m).\ N[\![e_n]\!]\ senv_a[\hat{x}_1/x_1,\ldots,\hat{x}_m/x_m]\ /\ f_n\ ]$$

The desired solution can then be found iteratively in the standard way; i.e., by creating Kleene's ascending chain of approximations, starting with the "least," or bottom element:

$$\langle env_0, senv_0\rangle \text{ where}$$
$$env_0 = [\ \lambda(v_1,\ldots,v_m).\ \bot_D\ /\ f_1,\ \ldots,\ \lambda(v_1,\ldots,v_m).\ \bot_D\ /\ f_n\ ]$$
$$senv_0 = [\ \lambda(\hat{x}_1,\ldots,\hat{x}_m).\ \bot_{Sv}\ /\ f_1,\ \ldots,\ \lambda(\hat{x}_1,\ldots,\hat{x}_m).\ \bot_{Sv}\ /\ f_n\ ]$$

where $\bot_D$ and $\bot_{Sv}$ are the bottom elements in the domains **D** and **Sv**, respectively (thus $\bot_{Sv}$ is simply the universal set containing all identifiers of interest). The next solution is:

$$\langle env_1, senv_1\rangle = \mathbf{Tau}\ \langle env_0, senv_0\rangle$$

and generally:

$$\langle env_i, senv_i\rangle = \mathbf{Tau}\ \langle env_{i-1}, senv_{i-1}\rangle,\ i>0$$

Note that $\langle env_i, senv_i\rangle$ is less defined than $\langle env_j, senv_j\rangle$ whenever $i<j$. We then define **<env',senv'>** (defined earlier using **whererec**) as the least upper bound of this ascending chain, and it is thus the least fixpoint of the system.

Theorem 1: (Safety). Let **env'** $=$ **Ep** $[\![p]\!]$ and **senv'** $=$ **S** $[\![p]\!]$ for some program **p**:

$$\begin{aligned} p = \{\ & f_1(x_1,\ldots,x_m) = e_1, \\ & f_2(x_1,\ldots,x_m) = e_2, \\ & \ldots \\ & f_n(x_1,\ldots,x_m) = e_n\ \} \end{aligned}$$

Then $x \in senv'[\![f_i]\!](\hat{x}_1,\ldots,\hat{x}_m) \Rightarrow env'[\![f_i]\!](d_1,\ldots,d_m)=\bot,\ i=1,\ldots,n$ where $d_j=\bot$ whenever $x \in \hat{x}_j$, $j=1,\ldots,m$.

Proof: (using fixpoint induction)

Let **Psi** be the predicate:

$$\begin{aligned} & \mathbf{Psi}\ \langle env,senv\rangle = \\ & \quad x \in senv[\![f_i]\!](\hat{x}_1,\ldots,\hat{x}_m) \Rightarrow \\ & \qquad env[\![f_i]\!](d_1,\ldots,d_m)=\bot,\ i=1,..,n \\ & \qquad \text{where } d_j=\bot \text{ whenever } x \in \hat{x}_j,\ j=1,\ldots,m. \end{aligned}$$

Consider first the least element:

$$\mathbf{Psi}\ \langle env_0,senv_0\rangle = (x \in \bot_N \Rightarrow \bot=\bot)$$

which is trivially true.

Now suppose **Psi** is true for some element $\langle env_{k-1},senv_{k-1}\rangle$ in the ascending chain. Then consider $\langle env_k,senv_k\rangle = \mathbf{Tau}\ \langle env_{k-1},senv_{k-1}\rangle$:

$$\mathbf{Psi}\ \langle env_k,senv_k\rangle$$

$$\begin{aligned} &= x \in senv_k[\![f_i]\!](\hat{x}_1,\ldots,\hat{x}_m) \Rightarrow env_k[\![f_i]\!](d_1,\ldots,d_m)=\bot,\ i=1,\ldots,n \\ &\qquad \text{where } d_j=\bot \text{ whenever } x \in \hat{x}_j \end{aligned}$$

$$\begin{aligned} &= x \in N[\![e_i]\!]senv_{k-1}[\hat{x}_1/x_1,\ldots,\hat{x}_m/x_m] \Rightarrow \\ &\qquad E[\![e_i]\!]env_{k-1}[d_1/x_1,\ldots,d_m/x_m] = \bot,\ i=1,\ldots,n \qquad (1) \\ &\qquad \text{where } d_j=\bot \text{ whenever } x \in \hat{x}_j \end{aligned}$$

the proof of which requires structural induction on e_i. First let $a = env_{k-1}[d_1/x_1,\ldots,d_m/x_m]$ and $b = senv_{k-1}[\hat{x}_1/x_1,\ldots,\hat{x}_m/x_m]$, where $d_j=\bot$ whenever $x \in \hat{x}_j$. Then either:

1. e_i is a constant. Then the lhs of (1) must be false, and so the implication is trivially true.
2. e_i is a bound variable. Then there is some l such that (1) becomes: $x \in \hat{x}_l \Rightarrow d_l=\bot$, which is true because of the qualification that $d_j=\bot$ whenever $x \in \hat{x}_j$, $j=1,\ldots,m$.
3. $e_i = e_1 \rightarrow e_2,e_3$. For the lhs of (1) to be true, either:
   a. $x \in N[\![e_1]\!]b$. Then by the (structural) induction hypothesis, $E[\![e_1]\!]a = \bot$ and by definition of the conditional, $E[\![e_i]\!]a = \bot$. Thus the implication (1) holds.
   b. $(x \in N[\![e_2]\!]b)$ AND $(x \in N[\![e_3]\!]b)$. Then by a similar application of the (structural) induction hypothesis, $E[\![e_i]\!]a = (pred \rightarrow \bot,\bot) = \bot$ and thus implication (1) follows.
4. $e_i = f(e_1,\ldots,e_n)$. Then (1) becomes:

$$\mathbf{x} \in \mathbf{b}[\![\mathbf{f}]\!](\mathbf{N}[\![\mathbf{e}_1]\!]\mathbf{b},\ldots,\mathbf{N}[\![\mathbf{e}_n]\!]\mathbf{b}) \Rightarrow$$
$$\mathbf{a}[\![\mathbf{f}]\!](\mathbf{E}[\![\mathbf{e}_1]\!]\mathbf{a},\ldots,\mathbf{E}[\![\mathbf{e}_n]\!]\mathbf{a}) = \bot$$

But $\mathbf{a}[\![\mathbf{f}]\!] = \mathbf{env}_{k-1}[\![\mathbf{f}]\!]$ and $\mathbf{b}[\![\mathbf{f}]\!] = \mathbf{senv}_{k-1}[\![\mathbf{f}]\!]$, since $\mathbf{f} \notin \mathbf{Bv}$, leading to:

$$\mathbf{x} \in \mathbf{senv}[\![\mathbf{f}]\!](\mathbf{N}[\![\mathbf{e}_1]\!]\mathbf{b},\ldots,\mathbf{N}[\![\mathbf{e}_n]\!]\mathbf{b}) \Rightarrow$$
$$\mathbf{env}[\![\mathbf{f}]\!](\mathbf{E}[\![\mathbf{e}_1]\!]\mathbf{a},\ldots,\mathbf{E}[\![\mathbf{e}_n]\!]\mathbf{a}) = \bot$$

Note now that by the (structural) induction hypothesis, $\mathbf{E}[\![\mathbf{e}_i]\!]\mathbf{a} = \bot$ whenever $\mathbf{x} \in \mathbf{N}[\![\mathbf{e}_i]\!]\mathbf{b}$. But then the (fixpoint) induction hypothesis immediately applies, and the implication (1) holds.

5. $\mathbf{e}_i = \mathbf{p}(\mathbf{e}_1,\ldots,\mathbf{e}_n)$. This depends on the correctness of **Ps**, which we take as given.

Thus implication (1) holds, and the theorem follows. □

The following corollary immediately follows from the above case analysis:

Corollary 1:

$$\mathbf{Psi} < \mathbf{env},\mathbf{senv} > \Rightarrow$$
$$(\ \mathbf{x} \in \mathbf{N}[\![\mathbf{e}]\!]\mathbf{senv} \Rightarrow \mathbf{E}[\![\mathbf{e}]\!]\mathbf{env}[\bot/\mathbf{x}] = \bot\ )$$

Note that if our analysis was perfect we could prove that:

$$\mathbf{x} \in \mathbf{senv}'[\![\mathbf{f}_i]\!](\hat{\mathbf{x}}_1,\ldots,\hat{\mathbf{x}}_m) \text{ iff } \mathbf{env}'[\![\mathbf{f}_i]\!](\mathbf{d}_1,\ldots,\mathbf{d}_m) = \bot,\ \mathbf{i}=\mathbf{1},\ldots,\mathbf{n}$$
$$\text{where } \mathbf{d}_j = \bot \text{ whenever } \mathbf{x} \in \hat{\mathbf{x}}_j,\ \mathbf{j}=\mathbf{1},\ldots,\mathbf{m}$$

i.e., that the implication goes both ways. But we know that there does not exist such a perfect analysis, because if there did it would constitute a direct solution to the halting problem.

Theorem 2: (Termination). If $\bot_{Sv}$ is finite, then the standard iterative technique of determining $\mathbf{S}[\![\mathbf{p}]\!]$ always terminates in a finite number of steps, for all **p**.

Proof: Rather obvious: The strategy terminates when one iteration yields the same solution as the previous one. Since $\bot_{Sv}$ is finite, and the approximations are monotonically decreasing, a fixpoint must be reached in a finite number of steps. □

References

1. Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4th ACM Sym. on Prin. of Prog. Lang., ACM, 1977, pp. 238-252.

2. Damas, L. and Milner, R. Principle type schemes for functional languages. 9th ACM Sym. on Prin. of Prog. Lang., ACM, Aug., 1982.

3. Gordon, J.C.. *The Denotational Description of Programming Languages*. Springer-Verlag, New York, 1979.

4. Hudak, P. ALFL Reference Manual and Programmers Guide. Research Report YALEU/DCS/RR-322, Second Edition, Yale University, Oct., 1984.

5. Hudak, P. and Kranz, D. A combinator-based compiler for a functional language. 11th ACM Sym. on Prin. of Prog. Lang., ACM, Jan., 1984, pp. 121-132.

6. Hudak, P. and Goldberg, B. Distributed execution of functional programs using serial combinators. To appear in Proceedings of 1985 Int'l Conf. on Parallel Proc. and IEEE Trans. on Computers (October 1985), Aug., 1985.

7. Hudak, P. and Goldberg, B. Serial combinators: "optimal" grains of parallelism. To appear in IFIP Int'l Conference on Functional Programming Languages and Computer Architecture, Sept, 1985.

8. Johnsson, T. Detecting when call-by-value can be used instead of call-by-need. Laboratory for Programming Methodology Memo 14, Chalmers University of Technology, Dept. of Computer Science, Oct., 1981.

9. Keller, R.M. FEL programmer's guide. AMPS TR 7, University of Utah, March, 1982.

10. Mycroft, A. The theory and practice of transforming call-by-need into call-by-value. Proc. of Int. Sym. on Programming, Springer-Verlag LNCS Vol. 83, 1980, pp. 269-281.

11. Turner, D.A. SASL language manual. University of St. Andrews, 1976.