Higher-Order Strictness Analysis in Untyped Lambda Calculus

Paul Hudak
Jonathan Young

Yale University
Department of Computer Science
Hudak@Yale, Young@Yale

Abstract

A function is said to be *strict* in one of its formal parameters if, in all calls to the function, either the corresponding actual parameter is evaluated, or the call does not terminate. 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 infer strictness properties of functions expressed in the lambda calculus. Our analysis improves on previous work in that (1) a set-theoretic characterization of strictness is used that permits treatment of *free variables*, which in turn permits a broader range of interpretations, and (2) the analysis provides an effective treatment of higher-order functions. We also prove a result due to Meyer [15]: the problem of first-order strictness analysis is complete in deterministic exponential time. However, because the size of most functions is small, the complexity seems to be tractable in practice.

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

1. Introduction

A function is said to be *strict* in one of its formal parameters if, in all calls to the function, either the corresponding actual parameter is evaluated, or the call does not terminate. More formally, a function $\mathbf{f}(x_1,x_2,\ldots,x_n)$ is strict in x_i if $\mathbf{f}(x_1,\ldots,x_{i-1},\perp,x_{i+1},\ldots,x_n)=\perp$ 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 [8], SASL [18], and FEL [14]), where knowing that a function is strict in a certain argument allows one to compute its value ahead of time, thus avoiding the overhead of a "closure," "self-modifying thunk" [9], "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 identify subexpressions that may be safely computed in parallel, with a potentially large reduction in overall execution time [10, 11].

As an example, consider the function **f(x,y,z) = if p(x) then y else z**, which is strict in **x** (assuming **p** is), but not **y** or **z**. Thus a language using lazy evaluation may result in a more efficient program. On the other hand, **f(x,y,z) = if p(x) then y+z else z** is strict in **x** (if **p** is), and also **z**. In most implementations one would like to compute **x** and **z** *prior* to calling **f**, to avoid the overhead of a closure.

A more interesting example derives from the simple factorial function:

fac(n) = if n=0 then 1 else n*fac(n-1)

In introductory programming one is taught that this is an elegant way to express factorial, but the unbounded nature of the stack makes it perhaps inefficient. A better approach would be to make it tail-recursive by introducing an "accumulator," as in:

fac(n,acc) = if n=0 then acc
else fac(n-1,n*acc)

which a suitable optimizing compiler will convert into a loop. However, in a language using lazy evaluation as the default function call strategy, a closure will have to be created to delay the evaluation of the expression **n*acc**, and those closures will not get invoked until the very end of the recursion. A little thought should convince the reader that this is no more efficient (in fact, probably less so) than the original solution! It is only through a suitable strictness analysis that a compiler can infer that the second version of **fac** is strict in **acc** (as well as **n**), at which point transformation into a loop pays off.

In this paper we present a strategy that allows one to infer strictness properties of functions expressed in the lambda calculus. The strategy improves on previous work in that: (1) a set-theoretic characterization of strictness is used that permits treatment of *free variables*, which in turn permits a broader range of interpretations, and (2) the analysis provides an effective treatment of higher-order functions, which is crucial for any implementation of a programming language that treats functions as "first-class citizens." We also prove a result due to Meyer [15]: the problem of first-order strictness analysis is complete in deterministic exponential time. However, because the size of most functions is small, the complexity seems to be tractable in practice.

2. Preliminaries

Before one can begin to speak of strictness properties, one needs to establish some preliminary notion of syntax and (standard) semantics.

2.1. Syntax

Instead of adhering to the conventional syntax for the lambda calculus, we use a "syntactic sugaring" that has become rather 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 $\mathbf{f}(\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n) = \mathbf{body}$ to define the n-ary function **f** (equivalently, in unnamed form, $\lambda(\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_n).\mathbf{body}$). 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 $\mathbf{f}(\mathbf{e}_1,\mathbf{e}_2,\ldots,\mathbf{e}_n)$, where **f** may be either a user-defined function or primitive operator (informally we sometimes use infix notation for primitive functions, as in $\mathbf{e}_1+\mathbf{e}_2$ -- the context should make the meaning clear). We also take as primitive a conditional operator, which permits expressions of the form $\mathbf{pred} \rightarrow \mathbf{con},\mathbf{alt}$, 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:

$\mathbf{c} \in \mathbf{Con}$ constants
$\mathbf{x} \in \mathbf{Bv}$ bound variables
$\mathbf{f} \in \mathbf{Fn}$ function variables
$\mathbf{p} \in \mathbf{Pf}$ primitive function names
$\mathbf{e} \in \mathbf{Exp}$ expressions, defined by:

$$\mathbf{e} ::= \mathbf{c} \mid \mathbf{x} \mid \mathbf{e}_1 \rightarrow \mathbf{e}_2,\mathbf{e}_3 \mid \mathbf{f}(\mathbf{e}_1,\ldots,\mathbf{e}_k) \mid \mathbf{p}(\mathbf{e}_1,\ldots,\mathbf{e}_k)$$

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

$$\mathbf{pr} ::= \{\ \mathbf{f}_1(\mathbf{x}_1,\ldots,\mathbf{x}_{k_1}) = \mathbf{e}_1,\ \ldots,\ \mathbf{f}_n(\mathbf{x}_1,\ldots,\mathbf{x}_{k_n}) = \mathbf{e}_n\ \}$$

where $\mathbf{k}_i$ is the arity of $\mathbf{f}_i$.

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 that an important restriction made 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 First-Order Semantics

2.2.1. Standard Semantic Domains

$\mathbf{Bas} = \mathbf{Int}+\mathbf{Bool}$	Basic values, integers and booleans.
$\mathbf{Fun} = \mathbf{Bas}^n \rightarrow \mathbf{Bas}$	First-order functions.
$\mathbf{D} = \mathbf{Bas}+\mathbf{Fun}+\{\mathbf{error}\}$	Denotable values.
$\mathbf{Env} = (\mathbf{Bv}+\mathbf{Fn}) \rightarrow \mathbf{D}$	Environments.

2.2.2. Auxiliary Semantic Functions

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

Conditional:

$$\mathbf{e}_1 \rightarrow \mathbf{e}_2,\mathbf{e}_3 = \begin{cases} \bot, & \text{if } \mathbf{e}_1 = \bot \\ \mathbf{error}, & \text{if not } \mathbf{Bool?}(\mathbf{e}_1) \\ \mathbf{e}_2, & \text{if } \mathbf{e}_1 = \mathbf{true} \\ \mathbf{e}_3, & \text{otherwise.} \end{cases}$$

Plus standard definitions for arithmetic and boolean operators.

2.2.3. Standard Semantic Functions

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

$\mathbf{K}$: $(\mathbf{Con}+\mathbf{Pf}) \rightarrow \mathbf{D}$, mapping constants to semantic values.
$\mathbf{E}$: $\mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{D}$, giving meaning to expressions.
$\mathbf{Ep}$: $\mathbf{Prog} \rightarrow \mathbf{Env}$, giving meaning to programs.

$\mathbf{K}[\![\mathbf{n}]\!] = \mathbf{n}$, if **n** is an integer
$\mathbf{K}[\![\mathbf{true}]\!] = \mathbf{true}$
$\mathbf{K}[\![\mathbf{false}]\!] = \mathbf{false}$

$\mathbf{K}[\![+]\!] = \lambda(\mathbf{x},\mathbf{y}).(\mathbf{Int?}\ \mathbf{x}\ \mathbf{AND}\ \mathbf{Int?}\ \mathbf{y}) \rightarrow \mathbf{x}+\mathbf{y}, \mathbf{error}$
$\mathbf{K}[\![\mathbf{and}]\!] = \lambda(\mathbf{x},\mathbf{y}).(\mathbf{Bool?}\ \mathbf{x}\ \mathbf{AND}\ \mathbf{Bool?}\ \mathbf{y}) \rightarrow \mathbf{x}\ \mathbf{AND}\ \mathbf{y}, \mathbf{error}$
$\mathbf{K}[\![\mathbf{cons}]\!] = \lambda(\mathbf{x},\mathbf{y}).\langle \mathbf{x},\mathbf{y} \rangle$
and so on for other primitive functions.

$$E[\![c]\!]env = K[\![c]\!]$$
$$E[\![x]\!]env = env[\![x]\!]$$
$$E[\![e_1 \rightarrow e_2,e_3]\!]env = E[\![e_1]\!]env \rightarrow E[\![e_2]\!]env, E[\![e_3]\!]env$$
$$E[\![f(e_1,...,e_k)]\!]env = env[\![f]\!](E[\![e_1]\!]env,...,E[\![e_k]\!]env)$$
$$E[\![p(e_1,...,e_k)]\!]env = K[\![p]\!](E[\![e_1]\!]env,...,E[\![e_k]\!]env)$$

$$Ep[\![\{ f_1(x_1,...,x_{k_1}) = e_1, ..., f_n(x_1,...,x_{k_n}) = e_n \}]\!] = env'$$
$$\text{whererec } env' = [\lambda(v_1,...,v_{k_1}).E[\![e_1]\!]env'[v_1/x_1,...,v_{k_1}/x_{k_1}]/f_1, ..., \lambda(v_1,...,v_{k_n}).E[\![e_n]\!]env'[v_1/x_1,...,v_{k_n}/x_{k_n}]/f_n]$$

Note that the "meaning" of a program is an *environment* containing values for all of the top-level functions f_1 through 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 $f(x_1,...,x_k) = body$ is strict in x_i whenever the expression **body** is guaranteed to evaluate x_i, according to the following (recursive) set of rules:

1. The evaluation of a parameter **x** always evaluates **x**.
2. The evaluation of $p(e_1,...,e_k)$ depends on **p**. For example, if it is a strict binary operator such as +, then both e_1 and e_2 are always evaluated; if it is a "sequential" operator such as **and**, then only e_1 is always evaluated.
3. The evaluation of $e_1 \rightarrow e_2,e_3$ always results in the evaluation of e_1, and also all variables that are evaluated in *both* e_2 and e_3.
4. The evaluation of $f(e_1,...,e_k)$ always results in the evaluation of 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 in [13], except that there a boolean domain is used that indicates whether or not a *particular* variable will be evaluated.

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

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

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 of Mycroft [16] does exactly that, but using a boolean domain as mentioned above. 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 **Need**[\![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 **Need**[\![f(x,y,z)]\!] = **Need**[\![x → y,z]\!] = **Need**[\![x]\!] ∪ (**Need**[\![y]\!] ∩ **Need**[\![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, **Need**[\![g(a,b)]\!] = **Need**[\![f(a,b,(b+1))]\!] = **Need**[\![a]\!] ∪ (**Need**[\![b]\!] ∩ **Need**[\![b+1]\!]). Clearly, **Need**[\![b+1]\!] = **Need**[\![b]\!], so **Need**[\![g(a,b)]\!] = **Need**[\![a]\!] ∪ **Need**[\![b]\!], as intuitively inferred.

We can formalize the above analysis by deriving an abstract interpretation [5, 17] of the original functions to obtain a new set of mutually recursive equations that capture the properties of interest. Rather than give an "informal" abstract interpretation as suggested above, we provide an alternative, or "non-standard," semantics that captures the properties of interest. We introduce the semantic function **N** that formally captures the intent of **Need** as used above, and having type **Exp** → **Senv** → **Sv**, where **Senv** is an environment containing strictness properties of free variables, and **Sv** is the powerset of **V** (the set of all variables of interest). More formally:

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

V, the set of variables of interest.
Sv, the powerset of **V**.
Sfun = $\mathbf{Sv^n}$ → **Sv**, the function space mapping sets of strict variables to other sets.
Senv = (**Bv**+**Fn**) → (**Sv**+**Sfun**), the strictness environment.

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

Kn: $\mathbf{Pf} \to (\mathbf{Sfun}+\mathbf{Sv})$, strictness properties of primitive functions.

N: $\mathbf{Exp} \to \mathbf{Senv} \to \mathbf{Sv}$, as intuitively described earlier.

Np: $\mathbf{Prog} \to \mathbf{Senv}$, the meaning of programs.

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

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

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

and so on for other primitive functions.

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

$$\mathbf{Np}[\![\{\, f_1(x_1,\ldots,x_{k_1}) = e_1,\ \ldots,\ f_n(x_1,\ldots,x_{k_n}) = e_n \,\}]\!] = \mathbf{senv}'$$
$$\mathbf{whererec\ senv}' = [\, \lambda(\hat{x}_1,\ldots,\hat{x}_{k_1}).\mathbf{N}[\![e_1]\!]\mathbf{senv}'[\ldots,\hat{x}_i/x_i,\ldots],\ \ldots,\ \lambda(\hat{x}_1,\ldots,\hat{x}_{k_n}).\mathbf{N}[\![e_n]\!]\mathbf{senv}'[\ldots,\hat{x}_i/x_i,\ldots]\,]$$

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]

$$f(x,y,z,p,q) = p>0 \to (p=1 \to (z=0 \to x,y),\ f(z,z,0,p-1,x)),\ f(0,0,z,1,y)$$

[1]This interesting example is a variation of one due to Simon Peyton Jones.

It follows from the above definitions that:

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

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

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

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:

$\hat{x}_1 \sqsubseteq \hat{x}_2$ iff $\hat{x}_1 \supseteq \hat{x}_2$
and $f_1 \sqsubseteq f_2$ iff $f_1(\hat{x}) \sqsubseteq f_2(\hat{x})$ 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 $\perp_{\mathbf{Sv}}$, and it is simply the set **V**, and the least defined function $\cup_n$ in each n-ary function space simply returns $\perp_{\mathbf{Sv}}$. 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):

$$\hat{f}^0(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) = \cup_5(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) = \mathbf{V}$$

$$\hat{f}^1(\hat{x},\hat{y},\hat{z},\hat{p},\hat{q}) = \hat{p} \cup ((\hat{z} \cup (\hat{x} \cap \hat{y})) \cap \mathbf{V} \cap \mathbf{V}) = \hat{p} \cup \hat{z} \cup (\hat{x} \cap \hat{y})$$

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

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

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

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

In this example it was obvious when the least upper bound in the chain was reached, but how is this done in general? In other words, how do we determine when $\hat{f}^i = \hat{f}^{i+1}$? As it turns out, this problem is NP-complete,

and 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.

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 $\{p,z\}$.

Of course, it would be convenient if there existed an algorithm to compute the fixpoint in less than exponential time. However, strictness functions are easily seen as recursive monotone boolean functions, under the interpretation of set variables as boolean variables, set union as boolean or, and set intersection as boolean and. Appendix I shows that evaluating even one such function at any point must take at least an exponential amount of time in the length of the function definition (more precisely, it is complete in deterministic exponential time). Thus we can do no better than the above strategy in the general case. Fortunately, in most applications user-defined functions are rather small, and do not grow with program size, so this method may be practical despite its exponential nature. The interested reader should refer to [4] for a useful discussion of the pragmatics of strictness analysis.

3.4. 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 ith 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 property 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 ith argument. Our analysis is thus only an *approximation*, but that is the best that we can hope for.

Appendix II contains detailed proofs for the following theorems:

Theorem 1: (First-Order Safety). Let $\mathbf{env'} = \mathbf{Ep}[\![\mathbf{pr}]\!]$ and $\mathbf{senv'} = \mathbf{Np}[\![\mathbf{pr}]\!]$ for some program **pr**:

$$\mathbf{pr} = \{\ f_1(x_1,\ldots,x_{k_1}) = e_1,\ \ldots,\ f_n(x_1,\ldots,x_{k_n}) = e_n\ \}$$

Then

$$x \in \mathbf{senv'}[\![f_i]\!](\hat{x}_1,\ldots,\hat{x}_{k_i}) \Rightarrow$$
$$\mathbf{env'}[\![f_i]\!](d_1,\ldots,d_{k_i}) = \perp,\ i=1,\ldots,n$$
$$\text{where } d_j = \perp \text{ whenever } x \in \hat{x}_j,\ j=1,\ldots,k_i.$$

The proof of Theorem 1 involves a classical use of structural and fixpoint induction.

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

$$x \in \mathbf{senv'}[\![f_i]\!](\hat{x}_1,\ldots,\hat{x}_{k_i}) \text{ iff}$$
$$\mathbf{env'}[\![f_i]\!](d_1,\ldots,d_{k_i}) = \perp,\ i=1,\ldots,n$$
$$\text{where } d_j = \perp \text{ whenever } x \in \hat{x}_j,\ j=1,\ldots,k_i.$$

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: (First-Order Termination). If $\perp_{Sv}$ 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 Higher-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:

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

Our analysis so far 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 higher-order functions of this sort.

4.1. Preliminaries -- Standard Higher-Order Semantics

We now consider all functions to be "curried" (including primitive ones) and thus our function definitions might look like: $\mathbf{f}\ x_1\ x_2\ \ldots\ x_n = \mathbf{body}$. However, we use the more verbose lambda calculus notation $\mathbf{f} = \lambda x_1.\lambda x_2.\ \ldots\ \lambda x_n.\mathbf{body}$ (or equivalenty, $\mathbf{f} = \lambda x_1\ x_2$

... x_n.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:

$c \in Con$ constants, including primitive functions
$x,f \in V$ variables
$eg \in EqGrp$ equation groups, defined by:

$$eg ::= \{ f_1 = e_1, \ldots, f_n = e_n \}$$

$e \in Exp$ expressions, defined by:

$$e ::= c \mid x \mid f \mid e_1 \rightarrow e_2,e_3 \mid e_1\ e_2 \mid \lambda x.e \mid eg$$

$pr \in Prog$ is the set of programs, defined by:

$$pr ::= eg$$

4.1.1. Standard Semantic Categories

$Bas = Int + Bool$, domain of basis values.
$D = Bas + (D \rightarrow D)$, domain of denotable values.
$Env = V \rightarrow D$, domain of environments.

4.1.2. Standard Semantic Functions

$K: Con \rightarrow D$
$E: Exp \rightarrow Env \rightarrow D$
$Ep: Prog \rightarrow D$

$$E[\![c]\!]env = K[\![c]\!]$$
$$E[\![x]\!]env = env[\![x]\!]$$
$$E[\![f]\!]env = env[\![f]\!]$$
$$E[\![e_1 \rightarrow e_2,e_3]\!]env = E[\![e_1]\!]env \rightarrow E[\![e_2]\!]env, E[\![e_3]\!]env$$
$$E[\![e_1\ e_2]\!]env = (E[\![e_1]\!]env)\ (E[\![e_2]\!]env)$$
$$E[\![\lambda x.e]\!]env = \lambda v.E[\![e]\!]env[v/x]$$
$$E[\![\ \{ f_1 = e_1, \ldots, f_n = e_n \}\ ]\!] = E[\![e_1]\!]env'$$
$$\text{where rec } env' = env[\, E[\![e_1]\!]/f_1, \ldots, E[\![e_n]\!]/f_n\,]$$
$$Ep[\![pr]\!] = E[\![pr]\!]null\text{-}env$$

4.2. Strictness Pairs

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 also 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. This suggests that the strictness property should perhaps be captured by an object of type T, say, where $T = Sv \times (T \rightarrow Sv) \times (T \rightarrow T \rightarrow Sv) \times \ldots$[2] Note in particular that the abstract functions take arguments of type T, since in the higher-order case functions may be passed as arguments.

[2]Indeed, this was the domain of *strictness ladders* used in [12].

However, a more concise and completely equivalent representation is the domain of *strictness pairs*, SP, defined by:

$$SP = Sv \times (SP \rightarrow SP)$$

We often write a particular strictness pair as $<sv, sf>$, and we use the following subscript notation to select elements from the pair:

$$<sv, sf>_v = sv$$
$$<sv, sf>_f = sf$$

With every expression **exp** in a "strictness environment" **senv**, we associate a strictness pair $S[\![exp]\!]senv$ that provides strictness properties of **exp** both as an isolated value and as a function to be applied. We write $S_v[\![exp]\!]senv$ for $(S[\![exp]\!]senv)_v$, and $S_f[\![exp]\!]senv$ for $(S[\![exp]\!]senv)_f$. Intuitively, $S_v[\![exp]\!]senv$ is a set of strict variables much like $N[\![exp]\!]senv$ in our previous analysis. But in addition we now have information that captures **exp**'s behavior as a function. In particular, $N[\![exp\ e]\!]senv = S_v[\![exp]\!]senv \cup ((S_f[\![exp]\!]senv)\ (S[\![e]\!]senv))$, since when evaluating a function call one must evaluate the function, and then apply it to its argument. Similar results are obtained for repeated (i.e., curried) applications. Note that the *entire strictness pair* $S[\![e]\!]senv$ is passed to $S_f[\![exp]\!]senv$, 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. 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 (Higher-Order Strictness)

V, variables of interest
Sv, the powerset of **V**
$SP = Sv \times (SP \rightarrow SP)$, domain of strictness pairs
$Senv = V \rightarrow SP$, the "strictness environments"

For clarity, we write $e_1 \sqcap e_2$ to mean $\lambda\hat{x}.<(e_1\ \hat{x})_v \cap (e_2\ \hat{x})_v, (e_1\ \hat{x})_f \sqcap (e_2\ \hat{x})_f>$. We also define a special error element $serr = \lambda\hat{x}.<\emptyset, serr>$, to be used when an expression has been applied "too many times."

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

Ks: Con $\rightarrow$ **SP**, maps constants to SP.
S: Exp $\rightarrow$ **Senv** $\rightarrow$ **SP**, maps expressions to s-pairs.
Sp: Prog $\rightarrow$ **Senv**, gives meaning to programs.

$Ks[\![c]\!] = <\emptyset, serr>$, if **c** is integer or boolean
$Ks[\![+]\!] = <\emptyset, \lambda\hat{x}.<\emptyset, \lambda\hat{y}.<\hat{x}_v \cup \hat{y}_v, serr>>>$
$Ks[\![=]\!] = Ks[\![+]\!]$
$Ks[\![AND]\!] = <\emptyset, \lambda\hat{x}.<\emptyset, \lambda\hat{y}.<\hat{x}_v, serr>>>$
and so on for other primitive functions.

$$S[\![c]\!]s = Ks[\![c]\!]$$
$$S[\![x]\!]s = s[\![x]\!]$$
$$S[\![f]\!]s = s[\![f]\!]$$

$$S[\![e_1 \rightarrow e_2, e_3]\!]s =$$
$$<S_v[\![e_1]\!]s \cup (S_v[\![e_2]\!]s \cap S_v[\![e_3]\!]s),$$
$$S_f[\![e_2]\!]s \sqcap S_f[\![e_3]\!]s>$$
$$S[\![e_1\ e_2]\!]s = <S_v[\![e_1]\!]s \cup sv, sf>$$
where $<sv, sf> = ((S_f[\![e_1]\!]s)\ (S[\![e_2]\!]s))$
$$S[\![\lambda x.e]\!]s = <\emptyset, \lambda\hat{x}.S[\![e]\!]s[\hat{x}/x]>$$

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

$$Sp[\![\{\ f_1 = e_1, \ldots,$$
$$f_n = e_n\ \}]\!]s = s'$$
$$\text{whererec } s' = s[\ S[\![e_1]\!]s'/f_1, \ldots,$$
$$S[\![e_n]\!]s'/f_n\ ]$$

Thus the abstract "meaning" of a program is again a "strictness environment," but that now binds the top-level functions to *strictness pairs*. As in our earlier analysis, we refer informally to the strictness pair of one of these functions as $\hat{f}$.

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

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

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

$$\hat{f} = S[\![\lambda x.x]\!] = <\emptyset, \lambda\hat{x}.\hat{x}>$$
$$\hat{g} = S[\![\lambda x.\ +\ x\ 1]\!]$$
$$= <\emptyset, \lambda\hat{x}.S[\![+\ x\ 1]\!]>$$
$$= <\emptyset, \lambda\hat{x}.<\hat{x}_v, serr>>$$

$$S[\![(=\ a\ 0) \rightarrow f, g]\!] = <\hat{a}_v \cup (\hat{f}_v \cap \hat{g}_v), \hat{f}_f \sqcap \hat{g}_f>$$
$$S[\![((=\ a\ 0) \rightarrow f, g)b]\!] =$$
$$<\hat{a}_v \cup (\hat{f}_v \cap \hat{g}_v) \cup sv, sf>$$
where $<sv, sf> = (\hat{f}_f \sqcap \hat{g}_f)\hat{b}$

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

$$S[\![((=\ a\ 0) \rightarrow f, g)b]\!] = <\hat{a}_v \cup \hat{b}_v, serr>$$

so finally:

$$\hat{h} = S[\![\lambda a\ b.((=\ a\ 0) \rightarrow f, g)b]\!]$$
$$= <\emptyset, \lambda\hat{a}.<\emptyset, \lambda\hat{b}.<\hat{a}_v \cup \hat{b}_v, serr>>>$$

This indicates that the function **h** is strict in both of its arguments, as was intuitively inferred earlier.

4.3. Correctness

As with first-order strictness, we would like to prove certain correctness properties of the higher-order analysis. To make the presentation clear, we first define some auxiliary functions and concepts.

Definition: Let SAP_n be a special "apply" operator for strictness pairs that gives meaning to the application of a strictness pair to **n** strictness pairs. It is defined by:

$$SAP_n(sp, sp_1, \ldots, sp_n) =$$
$$sp_v, \text{ if } n=0$$
$$sp_v \cup SAP_{n-1}(\ sp_f\ sp_1, sp_2, \ldots, sp_n\ ),$$
$$\text{if } n > 0$$

By convention, $SAP_n(sp) = \emptyset$ if $n < 0$.

Definition: Let AP_n be a similar "apply" operator for elements in the standard domain, defined by:

$$AP_n(e, e_1, \ldots, e_n) =$$
$$e, \text{ if } n=0$$
$$AP_{n-1}(\ e\ e_1, e_2, \ldots, e_n\ ), \text{ if } n > 0$$

AP_n is used to give symmetry to our presentation; note that $AP_n(\ e, e_1, \ldots, e_n\ )$ is really just $e\ e_1\ e_2 \ldots e_n$.

Next we define our interest in safety, which is a bit more complex than in the first-order case, because now we must consider expressions that may evaluate to functions. We say that $sp \in SP$ is *safe at level* n for value $e \in E$ (with respect to a variable **v**) if for all $m \leq n$, $s_i \in SP$,] and $e_i \in D$ (i=1,...,m) such that s_i is safe at level n-1 for e_i,[3] we have:

$$v \in SAP_m(\ sp, s_1, \ldots, s_m\ )$$
$$\Rightarrow AP_m(\ e, e_1, \ldots, e_m\ ) = \perp_D$$

Furthermore, we say that a strictness pair **sp** is *safe* for a value **e** (with respect to **v**) if for all $n \geq 0$, **sp** is safe at level **n** for **e**.

Finally, we wish to define a point-wise safety property for environments. We say that a strictness environment **senv** is *safe* for a standard environment **env** (with respect to a variable **v**) if $\forall x \in V$, **senv**$[\![x]\!]$ is safe for **env**$[\![x]\!]$ (with respect to **v**). Two environments related in this way are said to *correspond*. This leads us to:

Theorem 3: (Higher-Order Safety) For all expressions **exp**, variables **v**, and corresponding environments **senv** and **env** (with respect to **v**),

$S[\![exp]\!]senv$ is safe for $E[\![exp]\!]env$.

Proof: See Appendix III.

4.4. Computing the Least Fixpoint of Strictness Pairs

Given our higher-order analysis, how does one apply it to a particular program? That is, how does one compute the fixpoint of the resulting mutually recursive equations defining strictness pairs? The domain $SP = Sv \times (SP \rightarrow SP)$ seems to have unbounded "depth," making the computation seemingly difficult. Fortunately, the second element in a pair almost always degenerates to **serr** at some point, which can be represented compactly. Further, strictness pairs are almost never applied more than a finite number of times. Thus one may use the standard technique of starting with an initial approximation $\perp_{SP}$ for all strictness pairs, and iterating in the normal way to refine the approximations to whatever degree is necessary for the particular application.

The complexity of the resulting analysis is, of course, no better than in the first-order case. Indeed, it is worse, for in general it is not guaranteed to terminate! The reason is that there is occasion when a strictness pair needs to be applied an infinite number of times. Consider, for example, the function $f = \lambda x.\ f\ x\ x$:

$$\hat{f} = <\emptyset, \lambda\hat{x}.<\hat{f}_v \cup (\hat{f}_f\ \hat{x})_v \cup ((\hat{f}_f\ \hat{x})_f\ \hat{x})_v, ((\hat{f}_f\ \hat{x})_f\ \hat{x})_f>>$$ [4]

Note that early S_V elements in the nested pairs depend on "deeper" $SP \rightarrow SP$ elements, creating a circularity in computing $\hat{f}$. This aspect of untyped lambda calculus is an unfortunate one, considering that most programs have little need for functions such as this.[5] Our only solution to this problem currently is to impose a weak type discipline that disallows functions whose type is of arbitrary "order" or "depth". In particular, most versions of typed lambda calculus provide the necessary constraints [2], and schemata of this kind have been studied extensively elsewhere [6, 7].

4.5. Other Interpretations

Assume x does not occur free in e. Then note that $S[\![e\ x]\!] = <\hat{e}_v \cup sv, sf>$ where $<sv, sf> = \hat{e}_f\ \hat{x}$, and thus:

$$S[\![\lambda x.e\ x]\!] = <\emptyset, \lambda\hat{x}.<\hat{e}_v \cup sv, sf>>$$

However, by Eta-conversion $\lambda x.e\ x \equiv e$, yet:

$$S[\![e]\!] = \hat{e} = <\hat{e}_v, \hat{e}_f>$$

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:

$$S[\![g]\!] = \hat{g} = <\emptyset, \lambda\hat{x}.<\emptyset, \lambda\hat{y}.<\hat{x}_v \cup \hat{y}_v, serr>>>$$

For here note that $S[\![g\ e_1]\!] = <\emptyset, \lambda\hat{y}.<S_v[\![e_1]\!] \cup \hat{y}_v, serr>>$ and thus $g\ e_1$ does *not* evaluate e_1 (since $S_v[\![g\ e_1]\!] = \emptyset$), even though $g\ e_1\ e_2$ does.

Although this interpretation of strictness is reasonable, it is easy to imagine situations where one would want $g\ e_1$ to evaluate e_1 whenever $g\ e_1\ e_2$ does. Indeed, the only time one would normally evaluate $g\ e_1$ is when it is about to be applied, so evaluating 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?** $(g\ \perp) = \perp$ instead of **true**. We can alter our analysis to conform to this new interpretation by simply changing the definition of S when applied to lambda expressions, from:

$$S[\![\lambda x.e]\!]s = <\emptyset, \lambda\hat{x}.S[\![e]\!]s[\hat{x}/x]>$$

to:

$$S[\![\lambda x.e]\!]s = <S_v[\![e]\!]s[\perp_{SP}/x], \lambda\hat{x}.S[\![e]\!]s[\hat{x}/x]>$$

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

$$S[\![g]\!] = <\emptyset, \lambda\hat{x}.<\hat{x}_v, \lambda\hat{y}.<\hat{x}_v \cup \hat{y}_v, serr>>>$$

so that $S_v[\![g\ e_1]\!] = S_v[\![e_1]\!]$, which means that e_1 *is* evaluated when $g\ e_1$ is. This result highlights the generality of the strictness pair approach, stemming from it's treatment of free variables.[6]

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 Ks:

$$Ks[\![+]\!] = <\emptyset, \lambda\hat{x}.<\hat{x}_v, \lambda\hat{y}.<\hat{x}_v \cup \hat{y}_v, serr>>>$$

4.6. 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 $f(x_1, x_2, \ldots, x_n) = exp$ 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 $f'\ x_1\ x_2\ \ldots\ x_n = exp'$, 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 higher-order cases, respectively. Then what we wish to prove is:

Theorem 5: $N[\![exp]\!]se = S_v[\![exp']\!]se'$

That is, the first element of the strictness pair provides all of the information that the first-order analysis provided. A review of the definitions for N and S should convince the reader of this, but the details are omitted

---

[4] Since $S[\![f\ x]\!] = <\hat{f}_v \cup (\hat{f}_f\ \hat{x})_v, (\hat{f}_f\ \hat{x})_f>$, and $S[\![(f\ x)\ x]\!] = <\hat{f}_v \cup (\hat{f}_f\ \hat{x})_v \cup ((\hat{f}_f\ \hat{x})_f\ \hat{x})_v, ((\hat{f}_f\ \hat{x})_f\ \hat{x})_f>$.

[5] Note that the function $g = \lambda x.g$ does not have the circularity problem mentioned above, since "deep" elements depend only on more "shallow" ones, yet g is equivalent to f in the Beta-theory of the lambda calculus -- their Bohm trees are identical.

[6] Note that by giving a unique name to each node in a parse-tree, one can use the same strategy to determine all *subexpressions* that will be evaluated. This may be useful for compiler optimizations.

here.

4.7. A Final Example

The observant reader will have noticed that we did not provide a strictness pair for **cons** in the definition of **Ks**. It turns out that one can define **cons**, **car**, and **cdr** as higher-order functions in the pure lambda calculus, and get the "lazy" behavior that we desire. The definitions are:

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

Applying our analysis to these functions is a good test of its effectiveness, since higher-order functions get used in several ways. Indeed, the analysis is able to determine that the function **f** defined by:

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

is strict in **p** *and* **x**. The details of the analysis are left to the reader.

5. Acknowledgements

Over the past year many people have contributed in various ways to the overall content of this paper. Thanks first of all to Simon Peyton Jones, whose enlightening visit inspired us to pursue this topic further. We are also indebted to Sam Kamin, who first pointed out that our original domain of strictness ladders could be simplified. Albert Meyer was the first to state the complexity result in Appendix I [15]; given his result, our proof was developed independently in conjunction with Neil Immerman; Dana Angluin and Michael Fischer also provided invaluable help.

I. Complexity Result for First-Order Strictness

Definition: A recursive monotone boolean function (RMBF) is an equation $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n) = \mathbf{body}$, where **body** is an expression with syntax:

$$\mathbf{exp} ::= \mathbf{exp} \wedge \mathbf{exp} \mid \mathbf{exp} \vee \mathbf{exp} \mid \mathbf{0} \mid \mathbf{1} \mid \mathbf{f}(\mathbf{exp}_1,\ldots,\mathbf{exp}_n) \mid \mathbf{x}_i\ (i=1,\ldots,n)$$

The semantics are that **f** is the least fixpoint of this equation on the boolean domain $0 \sqsubseteq 1$, with the standard interpretation of $\wedge$ and $\vee$. That is, if we view **body** as a function from $(2^n \rightarrow 2) \rightarrow (2^n \rightarrow 2)$, then **f** is the fixpoint found by iterating **body** applied to **zero**, the function which maps all arguments to zero:

$$\mathbf{zero},\ \mathbf{body(zero)},\ \mathbf{body(body(zero))},\ \ldots$$

(Clearly this sequence terminates at a unique fixpoint, since there are a finite number of such funtions and **body** is monotonic.)

Definition: (RMBF) An *instance* of RMBF is an equation $\mathbf{eq} = \mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n)=\mathbf{body}$ and a set of n arguments $\mathbf{a} = \{\mathbf{a}_1,\ldots,\mathbf{a}_n\}$. $(\mathbf{eq},\mathbf{a}) \in$ RMBF iff $\mathbf{f}(\mathbf{a}_1,\ldots,\mathbf{a}_n)=1$ under the above semantics.

Definition: *Deterministic exponential time* is defined by:

$$\mathrm{EXPTIME}[n] = \bigcup_{c>0} \mathrm{TIME}[c^n]$$

(See [1] for the definition of TIME classes.)

Theorem: RMBF is complete in deterministic exponential time in the length of the instance **(eq, a)**. That is, evaluating a RMBF at any point takes, in the worst case, an exponential amount of time as a function of the length of the function definition.

Proof: Clearly it takes no longer than exponential time to evaluate a RMBF $\mathbf{f}(\mathbf{x}_1,\ldots,\mathbf{x}_n)=\mathbf{body}$ at any point. Observe that the number of arguments (n) is bounded by the length of the function, l, and write down a truth table of size 2^n, all zero. Iteratively find **body(g)** where **g** is the function represented by the truth table until the table ceases to change. This is the truth table for **f**. We have taken at most 2^n iterations, evaluating $\mathbf{body(g)}(\mathbf{x}_1,\ldots,\mathbf{x}_n)$ at 2^n places for each iteration. Clearly, evaluating $\mathbf{body(g)}(\mathbf{x}_1,\ldots,\mathbf{x}_l)$ takes at most l lookups in **g**'s table, or $l*n*2^n$, so we took total time $T < 2^n*2^n*l*n*2^n < c^l$ for $n > 8$ and sufficiently large l. (Remember that $n \leq l$).

Now we must show that RMBF is EXPTIME-hard. This is shown by simulating an alternating turing machine [3] with O(n) scratch space by a recursive monotone boolean function of length O(n). The function applied to a set of arguments will return 1 if and only if the turing machine accepts on the corresponding input.

Recall from [3] that an alternating turing machine is a tuple:

$M = (k, Q, \Sigma, \Gamma, \delta, q_0, g)$, where
k is the number of work tapes,
Q is a finite set of states,
Σ is a finite input alphabet.
Γ is a finite work tape alphabet,
$\delta \subseteq (Q \times \Gamma^k \times \Sigma) \times (Q \times \Gamma^k \times \{\text{left, right}\}^{k+1})$
is the next move relation,
$q_0 \in Q$ is the initial state,
$g: Q \rightarrow \{$ AND, OR, NOT, accept, reject $\}$

If g(q) = AND (respectively, OR, NOT, accept, reject), then q is said to be a universal (respectively, existential, negating, accepting, rejecting) state.

Quoting [3], "we require of δ that an accepting or rejecting configuration have no successors, universal and existential configurations have at least one, and negating configurations have exactly one." (Recall that a nondeterministic turing machine has only existential, accepting, and rejecting states, while a deterministic turing machine has at most one successor for each configuration.)

Suppose that A is an ATM which uses at most O(n) space on input x ($|x| = n$). Note that by Theorem 2.5 in

[3], we can assume that A has no negating states. In addition, we assume that A does not use its input tape, since, in light of the fact that we have O(n) space, we could simulate the input tape on the work tape by copying it there first. So we will take k=1 and $\Gamma=\{0, 1\}$. An ID (or configuration) for A looks like ($q \in Q$, $j \in \{1,...,n\}$, $T=(t_1,t_2,...,t_n)$), where T is the contents of the work tape.

We will simulate A by a RMBF C such that C(ID) = 1 just when A halts after being started in state ID.

Let C($q_1,...,q_q$, $p_1,...,p_n$, $\bar{p}_1,...,\bar{p}_n$, $t_1,...,t_n$, $\bar{t}_1,...,\bar{t}_n$) be our function. The arguments to C will represent the IDs as follows: ID (q_i, p_j, $T=(t_1,...,t_n)$) will correspond to:

C(0,...,q_i=1,...,0,
0,...,p_j=1,...,0,
1,...,$\bar{p}_j$=0,...,1,
$t_1,...,t_n$,
$\neg t_1,...,\neg t_n$)

Let ZERO = $\vee$ (p_j & $\bar{t}_j$),
and ONE = $\vee$ (p_j & t_j)

Clearly, if the variables represent a valid ID, then ZERO is true iff the tape head reads a zero, while ONE is true iff the tape head reads a one.

Let CODE[q,b] (for $q \in Q$, $b \in \{0,1\}$) be
1 if q is an accepting state
0 if q is a rejecting state
$\wedge(\vee)$ C(...q'...,
...p_j = ($p_{j+1} \wedge$(dir=left)) $\vee$
($p_{j-1} \wedge$(dir=right))...,
...$\bar{p}_j$ = ($\bar{p}_{j+1} \wedge$(dir=left)) $\vee$
($\bar{p}_{j-1} \wedge$(dir=right))...,
...t_j = ($\bar{p}_j \wedge t_j$) $\vee$ ($p_j \wedge$ b')...,
...$\bar{t}_j$ = ($\bar{p}_j \wedge \bar{t}_j$) $\vee$ ($p_j \wedge \neg$b')...
)

If q is a universal (existential) state where the large AND or OR is over all ((q,b),(q',b',dir)) in δ.

Now, C(ID) =
[ONE $\wedge$ ($\vee$ $q_i \wedge$ CODE[q_i,1])] $\vee$
[ZERO $\wedge$ ($\vee$ $q_i \wedge$ CODE[q_i,0])]
(where the OR is over all $q_i \in Q$).

It should be clear that if C is called with arguments representing a valid ID, then C only makes recursive calls with arguments representing valid IDs, so we can define the C "transition relation" ID $\xrightarrow{c}$ ID' if C(ID) calls C(ID') recursively in one step. It should be clear that the transition relation $\xrightarrow{c}$ is equivalent to the ATM transition relation $\xrightarrow{a}$ defined in [3] (because C preserves the state, position, and tape contents during each transition). In addition, by construction C correctly deals with accepting and rejecting states and quantified transitions, and so C(ID) is 1 iff the ATM accepts. (The inductive proof is left to the reader, who should note that the semantics of an ATM "accepting" are themselves a least fixpoint construction.)

Now we note that the size of C is O(n), where n was the size of the input to the ATM. This follows because |ONE| = |ZERO| = O(n), and |CODE[q_i,b]| = O(n), so |C| = |ONE| + |ZERO| + 2q * |CODE[q_i,b]| = O(n).[7]

Thus RMBF is complete for ASPACE, which by [3] is the same as EXPTIME.

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:

$\{..., f_i(x_1,...,x_{k_i}) = e_i, ...\}$

we define the functional **Tau** by:

Tau $<env_a,senv_a> = <env_b,senv_b>$,
where:
env_b =
$[..., \lambda(v_1,...,v_{k_i}).E[\![e_i]\!]\,env_a[v_1/x_1,...,v_{k_i}/x_{k_i}] / f_i, ...]$
$senv_b$ =
$[..., \lambda(\hat{x}_1,...,\hat{x}_{k_i}).N[\![e_i]\!]\,senv_a[\hat{x}_1/x_1,...,\hat{x}_{k_i}/x_{k_i}] / f_i, ...]$

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:

$<env_0,senv_0>$ **where**
$env_0 = [..., \lambda(v_1,...,v_{k_i}).\perp_D/f_i, ...]$
$senv_0 = [..., \lambda(\hat{x}_1,...,\hat{x}_{k_i}).\perp_{Sv}/f_i, ...]$

[7] Unfortunately, under a strict interpretation of input length, this proof is not quite correct. If we charge for each character in the function definition, including subscripts, we may need O(nlog(n)) characters to write the whole function definition, because the subscript n takes log(n) space (written in binary). Although our proof still works, we should really be more rigorous and state what type of reduction we are using to prove completeness. It is clear that a polynomial-time (say $O(n^2)$) transducer could output C given a description of the machine A, so the proof holds under a polynomial-time reduction. We believe that a log-space reduction would not be difficult.

where $\perp_D$ and $\perp_{Sv}$ are the bottom elements in the domains D and Sv, respectively (thus $\perp_{Sv} = V$). The next solution is:

$$<env_1,senv_1> = Tau\ <env_0,senv_0>$$

and generally:

$$<env_i,senv_i> = Tau\ <env_{i-1},senv_{i-1}>$$

Note that $<env_i,senv_i>$ is less defined than $<env_j,senv_j>$ 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: (First-Order Safety). Let $env' = Ep[\![pr]\!]$ and $senv' = Np[\![pr]\!]$ for some program pr:

$$pr = \{..., f_i(x_1,...,x_{k_i}) = e_i, ...\}$$

Then:

$$x \in senv'[\![f_i]\!](\hat{x}_1,...,\hat{x}_{k_i}) \Rightarrow$$
$$env'[\![f_i]\!](d_1,...,d_{k_i}) = \perp,\ i=1,...,n$$
where $d_j = \perp$ whenever $x \in \hat{x}_j$, $j=1,...,k_i$.

Proof: (using fixpoint induction)

Let **Psi** be the predicate:

$$Psi\ <env,senv> =$$
$$x \in senv[\![f_i]\!](\hat{x}_1,...,\hat{x}_{k_i}) \Rightarrow$$
$$env[\![f_i]\!](d_1,...,d_{k_i}) = \perp,\ i=1,...,n$$
where $d_j = \perp$ whenever $x \in \hat{x}_j$, $j=1,...,k_i$.

Consider first the least element:

$$Psi\ <env_0,senv_0> = (x \in \perp_N \Rightarrow \perp = \perp)$$

which is trivially true.

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

$$Psi\ <env_k,senv_k>$$

$$= x \in senv_k[\![f_i]\!](\hat{x}_1,...,\hat{x}_m) \Rightarrow$$
$$env_k[\![f_i]\!](d_1,...,d_m) = \perp,\ i=1,...,n$$
where $d_j = \perp$ whenever $x \in \hat{x}_j$

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

the proof of which requires structural induction on e_i. First let $a = env_{k-1}[d_1/x_1,...,d_m/x_m]$ and $b = senv_{k-1}[\hat{x}_1/x_1,...,\hat{x}_m/x_m]$, where $d_j = \perp$ 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: $x \in \hat{x}_l \Rightarrow d_l = \perp$, which is true because of the qualification that $d_j = \perp$ whenever $x \in \hat{x}_j$, $j=1,...,k_i$.
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, $E[\![e_1]\!]a = \perp$ and by definition of the conditional, $E[\![e_i]\!]a = \perp$. 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, $E[\![e_i]\!]a = (pred \rightarrow \perp,\perp) = \perp$ and thus implication (1) follows.
4. $e_i = f(e_1,...,e_n)$. Then (1) becomes:
   $x \in b[\![f]\!](N[\![e_1]\!]b,...,N[\![e_n]\!]b) \Rightarrow$
   $a[\![f]\!](E[\![e_1]\!]a,...,E[\![e_n]\!]a) = \perp$
   But $a[\![f]\!] = env_{k-1}[\![f]\!]$ and $b[\![f]\!] = senv_{k-1}[\![f]\!]$, since $f \notin Bv$, leading to:
   $x \in senv[\![f]\!](N[\![e_1]\!]b,...,N[\![e_n]\!]b) \Rightarrow$
   $env[\![f]\!](E[\![e_1]\!]a,...,E[\![e_n]\!]a) = \perp$
   Note now that by the (structural) induction hypothesis, $E[\![e_i]\!]a = \perp$ whenever $x \in N[\![e_i]\!]b$. But then the (fixpoint) induction hypothesis immediately applies, and the implication (1) holds.
5. $e_i = p(e_1,...,e_n)$. This depends on the correctness of **Kn**, which we take as given.

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

Corollary 1:

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

Theorem 2: (Termination). If $\perp_{Sv}$ is finite, then the standard iterative technique of determining $S[\![p]\!]$ 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 $\perp_{Sv}$ is finite, and the approximations are monotonically decreasing, a fixpoint must be reached in a finite number of steps. □

III. Correctness Proof for Higher-Order Strictness

Theorem 3: (Higher-Order Safety) For all expressions **exp**, variables **v**, and corresponding environments **senv** and **env** (with respect to **v**),

$S[\![exp]\!]senv$ is safe for $E[\![exp]\!]env$.

Proof: We proceed by structural induction on **exp**.

(1) **exp** is a constant: Trivially true.

(2) exp is a bound variable v. Then $S[\![v]\!]senv = senv[\![v]\!]$ and $E[\![v]\!]env = env[\![v]\!]$, but since env and senv correspond, $senv[\![v]\!]$ is safe for $env[\![v]\!]$ (wrt v).

(3) exp is a function application f e. First recall:

$$S[\![f\ e]\!]senv = \langle SAP_1(S[\![f]\!]senv, S[\![e]\!]senv), ((S[\![f]\!]senv)_f\ S[\![e]\!]senv)_f\rangle$$

$$E[\![f\ e]\!]env = (E[\![f]\!]env)\ (E[\![e]\!]env).$$

Then we proceed by first fixing $n \geq 0$, and showing that $S[\![f\ e]\!]senv$ is safe at level n for $E[\![f\ e]\!]env$. To do this we must show that for $m \leq n$, and s_i safe at n-1 for e_i, (i=1,...,m), then: $v \in SAP_m(S[\![f\ e]\!]senv, s_1,...,s_m) \Rightarrow AP_m(E[\![f\ e]\!]env, e_1,...,e_m) = \perp_D$.

$v \in SAP_m(S[\![f\ e]\!]senv, s_1,...,s_m)$

$\Rightarrow v \in (S_v[\![f\ e]\!]senv \cup SAP_{m-1}((S[\![f\ e]\!]senv)_f\ s_1, s_2,...,s_m))$

(by defn of SAP_n)

$\Rightarrow v \in (SAP_1(S[\![f]\!]senv, S[\![e]\!]senv) \cup SAP_{m-1}(((S[\![f]\!]senv)_f\ S[\![e]\!]senv)_f\ s_1, s_2,...,s_m))$

(by defn of S)

$\Rightarrow v \in (SAP_1(S[\![f]\!]senv, S[\![e]\!]senv) \cup SAP_m((S[\![f]\!]senv)_f\ S[\![e]\!]senv, s_1,...,s_m))$

(because $SAP_{m-1}(sp_f\ s_1, ..., s_m) \subseteq SAP_m(sp, s_1,...,s_m)$)

$\Rightarrow v \in (SAP_1(S[\![f]\!]senv, S[\![e]\!]senv) \cup SAP_{m+1}(S[\![f]\!]senv, S[\![e]\!]senv, s_1,...,s_m))$

$\Rightarrow$ either $AP_1(E[\![f]\!]env, E[\![e]\!]env) = \perp_D$ or $AP_{m+1}(E[\![f]\!]env, E[\![e]\!]env, e_1,...,e_m) = \perp_D$

(since $S[\![f]\!]senv$ is safe for $E[\![f]\!]env$ and $S[\![e]\!]senv$ is safe for $E[\![e]\!]env$)

Then clearly the result:

$$AP_m(E[\![f\ e]\!]env, e_1,...,e_m) = \perp_D$$

follows by a case-by-case analysis (since $\perp_D\ e = \perp_D$).

(4) exp is a lambda abstraction λv.e. Recall:

$$S[\![\lambda v.e]\!]senv = \langle \emptyset, \lambda\hat{v}.S[\![E]\!]senv[\hat{v}/v]\rangle$$

$$E[\![\lambda v.e]\!]env = \lambda x.E[\![e]\!]env[x/v]$$

We need to show that $S[\![exp]\!]senv$ is safe for $E[\![exp]\!]env$. That is, for all n, $m \leq n$, and s_i safe at n-1 for e_i (i=1,...,m):

$$v \in SAP_m(S[\![\lambda v.e]\!]senv, s_1,...,s_m) \Rightarrow AP_m(E[\![\lambda v.e]\!]env, e_1,...,e_m) = \perp_D$$

Clearly this is true for n=0, since:

$$SAP_0(S[\![\lambda v.e]\!]senv) = S_v[\![\lambda v.e]\!]senv = \emptyset$$

On the other hand, for $n > 0$, we have:

$SAP_m(S[\![\lambda v.e]\!]senv, s_1,...,s_m)$

$= S_v[\![\lambda v.e]\!]senv \cup SAP_{m-1}((S[\![\lambda v.e]\!]senv)_f\ s_1,...,s_m)$

$= \emptyset \cup SAP_{m-1}((S[\![\lambda v.e]\!]senv)_f\ s_1,...,s_m)$

$= SAP_{m-1}(S[\![E]\!]senv[s_1/v], s_2,...,s_m)$

But since senv and env correspond, and s_1 is safe at n-1 for e_1, we have $senv[s_1/v]$ corresponds to $env[e_1/v]$ (wrt v), and thus $S[\![E]\!]senv[s_1/v]$ is safe for $E[\![E]\!]env[e_1/v]$. Further:

$v \in SAP_{m-1}(S[\![E]\!]senv[s_1/v], s_2,...,s_m)$

$\Rightarrow AP_{m-1}(E[\![E]\!]env[e_1/v], e_2,...,e_m) = \perp_D$

$\Rightarrow AP_{m-1}(E[\![\lambda v.E]\!]env\ e_1, e_2,...,e_m) = \perp_D$

$\Rightarrow AP_m(E[\![\lambda v.E]\!]env, e_1, e_2,...,e_m) = \perp_D$

(5) exp is an equation group $\{..., f_i = e_i, ...\}$:

$$S[\![\{..., f_i = e_i, ...\}]\!]senv = S[\![e_1]\!]senv'$$

whererec $senv' = senv[S[\![e_i]\!]senv'/f_i]$

$$E[\![\{..., f_i = e_i, ...\}]\!]env = E[\![e_1]\!]env'$$

whererec $env' = env[E[\![e_i]\!]senv'/f_i]$

We must show that $S[\![exp]\!]senv$ is safe for $E[\![exp]\!]env$, which we do by fixpoint induction. First recall $\perp_{SP} = \langle V, \lambda sp.\perp_{SP}\rangle$ is the bottom element of SP. Then let:

$$senv'_0 = \lambda x.x{=}f_i \rightarrow \perp_{SP}, senv[\![x]\!]$$

$$env'_0 = \lambda x.x{=}f_i \rightarrow \perp_D, env[\![x]\!]$$

Clearly $senv'_0$ is safe for env'_0. Now let:

$$senv'_n = \lambda x.x{=}f_i \rightarrow S[\![e_i]\!]senv'_{n-1}, senv[\![x]\!]$$

$$env'_n = \lambda x.x{=}f_i \rightarrow E[\![e_i]\!]env'_{n-1}, env[\![x]\!]$$

Then we claim that $senv'_n$ is safe for env'_n for all n, since $senv'_{n-1}$ is safe for env'_{n-1} and thus by the structural induction hypothesis $S[\![e_i]\!]senv'_{n-1}$ is safe for $E[\![e_i]\!]env'_{n-1}$. Then by fixpoint induction senv' is safe for env', and thus $S[\![exp]\!]senv'$ is safe for $E[\![exp]\!]env'$. □

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D.. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2. Burn, G.L., Hankin, C.L., and Abramsky, S. The theory and practice of strictness analysis for higher order functions. DoC 85/6, Imperial College of Science and Technology, Department of Computing, April, 1985.

3. Chandra, A.K., Kozen, D.C., and Stockmeyer, L.J. "Alternation". *JACM 28*, 1 (Jan. 1981).

4. Clack, C., and Peyton Jones, S.L. Strictness analysis -- a practical approach. Functional Programming Languages and Computer Architecture, Sept, 1985, pp. 35-49.

5. 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.

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

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

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

9. 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.

10. Hudak, P. and Goldberg, B. Distributed execution of functional programs using serial combinators. Proceedings of 1985 Int'l Conf. on Parallel Proc. (and IEEE Trans. on Computers October 1985), Aug., 1985, pp. 831-839.

11. Hudak, P. and Goldberg, B. Serial combinators: "optimal" grains of parallelism. Functional Programming Languages and Computer Architecture, Sept, 1985, pp. 382-388.

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

13. 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.

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

15. Meyer, A.R. Complexity of program flow-analysis for strictness.

16. 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.

17. Mycroft, A. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* Ph.D. Th., Univ. of Edinburgh, 1981.

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