Collecting Interpretations of Expressions
(Preliminary Version)

Paul Hudak

August 1986

Yale University
Department of Computer Science

Research Report YALEU/DCS/RR-497

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Collecting Interpretations of Expressions (Preliminary Version)

Paul Hudak

August 1986

Yale University
Department of Computer Science
Research Report YALEU/DCS/RR-497
Box 2158 Yale Station
New Haven, CT 06520
Arpanet: hudak@yale

Abstract

A *collecting interpretation of expressions* is an interpretation of a program that allows one to answer questions of the sort: "What are all possible values to which the expression *exp* might evaluate during program execution?" Answering such questions for functional programs is akin to traditional *data flow analysis* of imperative programs, and when used in the context of *abstract interpretation*, allows one to infer properties that approximate the run-time behavior of expression evaluation. In this paper collecting interpretations of expressions are developed for the standard semantics of three abstract functional languages: (1) a first-order language with call-by-value semantics, (2) a first-order language with call-by-name semantics, and (3) a higher-order language with call-by-name semantics (i.e., the full untyped lambda calculus with constants). It is argued that the method is simpler (for example, no powerdomain construction is needed) yet more expressive than existing methods (indeed, it is the first collecting interpretation for either lazy or higher-order programs).

This research was supported in part by the National Science Foundation under Grants DCR-8403304 and DCR-8451415, and the Department of Energy under Grant DE-FG02-86ER25012.

1 Introduction

Abstract interpretation [5,6] has been shown to be an effective methodology for expressing many compile-time analyses of programs. Its chief attraction is that it expresses a compile-time analysis as an *abstraction* of, or approximation to, the standard (or possibly non-standard) semantics of the source language. This results in a unified framework within which one can reason about compile-time analyses, and allows correctness properties to be proven straightforwardly. The most widely known example of applied abstract interpretation is strictness analysis on flat domains[4,11,17], but other applications include strictness analysis on non-flat domains[12,15], reference counting[10], sharing of partial applications[9], various data-flow analyses [5,20], and even applications in logic programming.

The Cousots' seminal work [5,6] on abstract interpretation concentrated on inferring properties of imperative programs. However, recent interest in purely functional languages[1] has honed interest in abstract interpretation of such languages. This began with Mycroft's thesis [16] and has recently flourished in scope and application (see the forthcoming book [1] for a summary of recent results).

In this paper we investigate an idea closely related to abstract interpretation, namely a *collecting interpretation*. Roughly speaking, a collecting interpretation is a program analysis that collects properties about *program points*, where the points are not considered in isolation, but rather *in the context of a particular program*. A collecting interpretation thus bears a strong resemblance to traditional *data flow analysis*, and when used together with abstract interpretation provides useful compile-time information that may not otherwise be obtainable. Examples of the formal treatment of a collecting interpretation in a denotational setting include the Cousots' original *static semantics*, the *minimum function graph* (mfg) semantics in [14], and the collecting interpretations used in [10,19,20].

The collecting interpretations investigated in this paper are considered within the framework of functional programs, where the notion of a "program point" shall refer, quite simply, to an *expression*, and thus the result is called a *collecting interpretation of expressions*. Furthermore, since there is no "store" component in a typical denotational semantics of a functional language, we must state what it is we are collecting. The answer is simply either concrete or abstract *values* and *environments*. Thus the collecting interpretations developed here answer the following sort of question: "What are all possible values to which the expression *exp* might evaluate during program execution?"

Three collecting interpretations of expressions are presented in this paper, one for each of three abstract functional languages:

- $\hat{\mathcal{E}}_{1a}$, for first-order languages with call-by-value semantics (Section 4).

[1] Of which there are now many, including ML, ALFL, Hope, Ponder, Orwell, FEL, SASL, KRC and Miranda.

- $\hat{\mathcal{E}}_{1n}$, for first-order languages with call-by-name semantics (Section 5).
- $\hat{\mathcal{E}}_h$, for higher-order languages with call-by-name semantics (Section 6).

The first of these is equivalent in power to an mfg semantics [14]; the second two are new developments. In all three cases the methodology used is relatively straightforward, and is easily related to the standard semantics.

In the remaining sections several applications of the theory will be discussed.

2 Notation

Domains as used here are *cpos* – chain-complete partial orders with a unique least element called "bottom." For a domain D the bottom element is written $\perp_D$, or just $\perp$ when the domain is clear from context. $A* \rightarrow B$ denotes the domain $B + (A \rightarrow B) + (A \rightarrow A \rightarrow B) + \cdots$. We write "$d \in D = Exp$" to define the domain (or set) D with "canonical" element d. $\mathcal{P}(S)$ denotes the powerset of S, and $\{\}$ is the empty set. Unless stated otherwise, we will treat $\mathcal{P}(S)$ as a domain, ordered pointwise by set inclusion.

Lambda expressions of the form "$\lambda x.\ exp$" carry with them implicit type information that is usually clear from context. When it is not clear, the notation "$\lambda x : D.\ exp : E$" will be used, having the standard meaning of a function in $(D \rightarrow E)$. Similarly, all domain/subdomain coercions will be omitted when clear from context. It is convenient to write "n-ary" lambda expressions as "$\lambda x_1\ x_2\ \ldots\ x_n.\ exp$," and when $n = 0$ we interpret this to mean just *exp*.

Double brackets are used to surround syntactic objects, as in $\mathcal{E}_h[\![exp]\!]$; square brackets are used for environment update, as in $env[e/x]$; and angle brackets are used for tupling, as in $\langle e_1, e_2, e_3 \rangle$. The notation $env[e_i/x_i]$ is shorthand for $env[e_1/x_1,\ \ldots,\ e_n/x_n]$, where the subscript bounds are inferred from context. Thus new environments are created by $\perp[e_i/x_i]$. The ith component of a tuple t is written $t \downarrow i$; alternatively, tuples may be "destructured" in let or lambda expressions. For example, "let $\langle x, y \rangle = exp$ in *body*" is equivalent to "let $x = exp{\downarrow}1,\ y = exp{\downarrow}2$ in *body*."

3 Preliminary Discussion

3.1 Previous Work

Collecting interpretations for expressions have not been studied very extensively before.[2] One obvious approach would be to extend the Cousots' original work so that expressions

[2]Although [16] contains something called a collecting interpretation, it is not a collecting interpretation as we have defined here.

in the presence of side-effects would cause changes to the state, and the states could be collected accordingly. However, we are interested in a more direct approach, and for interpretations based on both applicative-order and normal-order evaluation.

The most promising approach to date for a true collecting interpretation of expressions is Jones and Mycroft's *minimal function graph* (or *mfg*) semantics, which collects, for every function in a first-order program with call-by-value semantics, all argument tuples and results that could occur during program execution. They do this by essentially simulating the call/return behavior of function calls, by extending the base domain so as to contain an extra bottom element; one bottom represents the fact that no demand for the value was made, and the other bottom represents non-termination in the classical sense.[3] However, if one is interested primarily in the value *returned* from the call (which seems all to be required by a collecting interpretation), then it seems unnecessary to introduce this extra level of detail into the semantics. This will be discussed in more detail later.

A significant difficulty that has been faced by researchers attempting to formulate collecting interpretations for the applicative idiom is the apparent need to "lift" a function, say $f : D \rightarrow E$, to a function $f' : PD(D) \rightarrow PD(E)$, where $PD(...)$ is a *powerdomain* constructor. An early attempt to get these functions to behave properly (in particular, to preserve monotonicity and continuity) can be found in [16], but problems with that approach led to an improvement in [18]. This in turn was improved upon in [21], where powerdomains were abandoned in favor of a category-theoretic approach.

The approach presented in this paper avoids these problems by completely obviating the need to "lift" functions using powerdomains. The resulting approach seems to be much simpler, does not have deep semantical problems rooted in powerdomains or categories, and has allowed us to extend the current state of research in collecting interpretations in the following ways:

- A collecting interpretation for *expressions* is developed rather than for functions as a whole (the latter being what an mfg interpretation accomplishes).
- Extensions are made to languages with lazy evaluation.
- Extensions are made to languages with higher-order functions.

3.2 Intuitive Overview

Intuitively, what we desire as the "answer" to a program is an object, call it *cache*, such that $cache[\![exp]\!]$ returns the set of all possible values that exp could evaluate to during program execution. Doing this we run immediately into a small technical difficulty, namely finding

[3]This same approach was used in [10], but there non-termination was not an issue, so the two "bottoms" were synonymous.

a way to uniquely reference each expression. We solve this problem by assuming that each expression has a unique *label* from a primitive syntactic domain *Lab*. A labelled expression is written $[\![l.e]\!]$, where $l \in Lab$, and we define the syntactic functions *expr* and *label* such that $expr[\![l.e]\!] = [\![e]\!]$ and $label[\![l.e]\!] = [\![l]\!]$. We often omit the label from an expression when its presence is not needed. (Labels are similar to *occurrences* and *places* as used in [3,7,10,16,20].)

So now our cache should have functionality $Lab \rightarrow \mathcal{P}(D)$, if we assume D to be the domain of values we are collecting. Although $\mathcal{P}(D)$ denotes the *powerset* of D (i.e., not a powerdomain), we will treat it as a domain, ordered pointwise by set inclusion, whose bottom element $\perp_{\mathcal{P}(D)}$ is the empty set $\{\}$. That the empty set is the appropriate bottom element can be made clear by a simple example. Consider the program:

$$\begin{aligned} pr = [\![\ \{\ & f_1 = \text{if } true \text{ then } l_1.f_2(1) \text{ else } l_2.f_2(2), \\ & f_2 = \lambda x.x\ \}\ ]\!] \end{aligned}$$

and suppose *cache* is the result of a collecting interpretation of expressions for *pr*. Then $cache(l_1) = \{1\}$, but $cache(l_2) = \{\}$, because $[\![f_2(2)]\!]$ is never called during program execution. Thus, unlike most other domains, the bottom element $\perp_{\mathcal{P}(D)}$ does not denote non-termination (although elements of $\mathcal{P}(D)$ may contain $\perp_D$, indicating that one possible outcome is non-termination), but rather indicates the absence of any result at all.

This point becomes even more important in a language with lazy evaluation. In particular, just because one occurrence of a bound variable is evaluated doesn't mean that another is, and that is one reason why labels are necessary; i.e., to distinguish the different occurrences. For example, in the program *pr* above, if f_2 were really defined by:

$$f_2 = \lambda x.\ \text{if } (l_3.x = 1) \text{ then } (l_4.x + 1) \text{ else } (l_5.x + 2)$$

then $cache(l_3) = cache(l_4) = \{1\}$, but $cache(l_5) = \{\}$. This is not merely an artifact of the analysis, but rather a very deliberate behavior, since the same sort of differences within a suitable abstraction can provide exploitable compile-time information for use by the industrious compiler writer.

3.3 A Motivating Example

We conclude this section by presenting one motivating application of this work to the ever-popular field of *strictness analysis*. Consider the typical definition of a *map* function such that *map f lst* builds a new list from *lst* by applying *f* to each of *lst*'s elements. Higher-order strictness analysis will tell us strictness properties of *map*, but *only as a function of f's strictness properties.* Thus despite strictness analysis, the compiler-writer is not free to turn *f*'s application in the body of *map* from call-by-name to call-by-value, because at compile-time *f* is unknown. However, a collecting interpretation might be able to help in two different ways:

1. It could determine that all possible functions bound to f *in the body of map* were strict, thus allowing the optimization mentioned.

2. It could determine that all possible functions bound to f *at a particular application of map* were strict, thus allowing an optimized version of *map* to be used there, and presumably a more conservative *map* to be used elsewhere.

The strictness analysis research community has for the most part ignored this problem, although it has been pointed out in [8], where empirical studies have indicated that higher-order strictness analysis (as opposed to just first-order) makes no significant impact on program performance (and the above problem is the primary reason why).

4 First-Order Language, Applicative-Order Semantics

For this section and the next two, standard semantic functions such as $\mathcal{E}_{1a}$ ("1st-order, *a*pplicative-order"), $\mathcal{E}_{1n}$ ("1st-order, *n*ormal-order"), and $\mathcal{E}_h$ ("*h*igher-order") will be defined. Their counterparts in the collecting interpretations will be denoted using a "top-hat," as in $\hat{\mathcal{E}}_{1a}$, $\hat{\mathcal{E}}_{1n}$, and $\hat{\mathcal{E}}_h$.

We begin our development with a collecting interpretation of expressions for a first-order language with applicative-order reduction semantics. The reason for starting with such a restricted language is that it is essentially the same language for which an mfg semantics was developed by Jones and Mycroft. From this starting point we will next consider lazy evaluation, and then higher-order functions.

The abstract syntax of a first-order language can be given as follows:

$$
\begin{array}{rcll}
l & \in & Lab & \text{labels} \\
k, p & \in & Con & \text{constants} \\
x & \in & Bv & \text{bound variables} \\
f & \in & Fv & \text{function variables} \\
e & \in & Exp & \text{expressions, where } e \leftarrow l.k \mid l.x \mid l.p(e_1 ... e_n) \mid l.f(e_1 ... e_n) \\
pr & \in & Prog & \text{programs, where } pr \leftarrow \{\ f_i(x_1 ... x_n) = e_i\ \}
\end{array}
$$

Note that all expressions are labelled; we assume that every label in a program $pr \in Prog$ is unique. A program is a set of mutually-recursive first-order equations. For simplicity we assume that f_1 is always a function of no arguments, and thus a program is "run" by evaluating $f_1()$. There are two standard ways of interpreting such programs, depending on whether one wishes to model applicative-order or normal-order reduction in the lambda-calculus, and corresponding more colloquially to call-by-value or call-by-name evaluation, respectively. In this section we consider an applicative-order semantics; in the next we consider normal-order.

4.1 Standard Applicative-Order Semantics for First-Order Programs

We assume a domain D whose structure depends on the base types implied by Con. For example, if integers and truth values were the only base types then $D = Int + Bool$. Now define two environment domains, one for bound variables, the other for function names:

$$
\begin{array}{rcll}
bve \in Bve & = & Bv \to D & \text{(bound variable environments)} \\
fve \in Fve & = & Fv \to (D* \to D) & \text{(function variable environments)}
\end{array}
$$

and then define the semantic functions $\mathcal{E}_{1a}$ and $\mathcal{P}_{1a}$ by:

$$
\begin{array}{lll}
\mathcal{E}_{1a}: & Lab \to Bve \to Fve \to D & \text{(gives meaning to expressions)} \\
\mathcal{P}_{1a}: & Prog \to D & \text{(gives meaning to programs)}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_{1a}\ lab\ bve\ fve = \text{case } expr(lab) \text{ of} \\
\qquad \begin{array}{rl}
[\![k]\!]: & \mathcal{A}_{1a}[\![k]\!] \\
[\![x]\!]: & bve[\![x]\!] \\
[\![p(e_1 ... e_n)]\!]: & \mathcal{K}_{1a}[\![p]\!](\mathcal{E}_{1a}\ label[\![e_1]\!]\ bve\ fve,\ ...,\ \mathcal{E}_{1a}\ label[\![e_n]\!]\ bve\ fve) \\
[\![f(e_1 ... e_n)]\!]: & fve[\![f]\!](\mathcal{E}_{1a}\ label[\![e_1]\!]\ bve\ fve,\ ...,\ \mathcal{E}_{1a}\ label[\![e_n]\!]\ bve\ fve)
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}_{1a}[\![\{\ f_i(x_1 ... x_n) = e_i\}]\!] = fve[\![f_1]\!] \\
\qquad \text{whererec } fve = \bot[\ strict(\lambda(y_1 ... y_n).\ \mathcal{E}_{1a}\ label[\![e_i]\!]\ \bot[y_j/x_j]\ fve)\ /\ f_i\ ]
\end{array}
$$

Note that the meaning of program is just the meaning of f_1, which is assumed to be a function of no arguments, as discussed earlier. The function *strict* is similar to that used in [22], and essentially makes its functional argument return bottom if it is applied to any bottom arguments. $\mathcal{A}_{1a}$ and $\mathcal{K}_{1a}$ give meaning to atoms and primitive functions, respectively, and are assumed to be given. Except for the presence of labels in the syntax, this semantics is very straightforward and conventional.

4.2 Applicative-Order Collecting Interpretation for First-Order Programs

We now define a collecting interpretation of expressions that is consistent with the standard semantics just defined.

$$
\begin{array}{lcl}
Bve & = & Bv \to D \\
Fve & = & Fv \to (Ans* \to Ans) \\
Ans & = & D \boxtimes Cache \\
Cache & = & Lab \to \mathcal{P}(D)
\end{array}
$$

$$
\begin{array}{ll}
\hat{\mathcal{E}}_{1a}: & Lab \to Bve \to Fve \to Ans \\
\hat{\mathcal{P}}_{1a}: & Prog \to Ans \\
\hat{\mathcal{K}}_{1a}: & Con \to (Ans* \to Ans)
\end{array}
$$

$$
\begin{array}{rl}
\hat{\mathcal{E}}_{1a}\ lab\ bve\ fve = & \text{case } expr(lab) \text{ of} \\
[\![k]\!]: & \langle \mathcal{A}_{1a}[\![k]\!], \bot[\{\mathcal{A}_{1a}[\![c]\!]\}/lab]\rangle \\
[\![x]\!]: & \langle bve[\![x]\!], \bot[\{bve[\![x]\!]\}/lab]\rangle \\
[\![p(e_1 ... e_n)]\!]: & \text{let } e'_i = \hat{\mathcal{E}}_{1a}\ label[\![e_i]\!]\ bve\ fve,\ i = 1, ..., n \\
& \quad \langle d, c\rangle = \hat{\mathcal{K}}_{1a}[\![p]\!](e'_1 ... e'_n) \\
& \text{in } \langle d, c \sqcup \bot[\{d\}/lab]\rangle \\
[\![f(e_1 ... e_n)]\!]: & \text{let } e'_i = \hat{\mathcal{E}}_{1a}\ label[\![e_i]\!]\ bve\ fve,\ i = 1, ..., n \\
& \quad \langle d, c\rangle = fve[\![f]\!](e'_1 ... e'_n) \\
& \text{in } \langle d, c \sqcup \bot[\{d\}/lab]\rangle
\end{array}
$$

$$
\begin{array}{l}
\hat{\mathcal{P}}_{1a}[\![\{\ f_i(x_1 ... x_n) = e_i\}]\!] = fve[\![f_1]\!] \\
\text{whererec } fve = \bot[\ strict'(\lambda(\langle d_1, c_1\rangle ... \langle d_n, c_n\rangle). \\
\qquad\qquad\qquad\qquad \text{let } \langle d, c\rangle = \hat{\mathcal{E}}_{1a}\ label[\![e_i]\!]\ \bot[d_j/x_j]\ fve \\
\qquad\qquad\qquad\qquad \text{in } \langle d, c \sqcup c_1 \sqcup ... \sqcup c_n\rangle \qquad\qquad)/\ f_i\ ]
\end{array}
$$

where *strict'* is analogous to *strict* in the standard semantics, but checks for bottom only in the first element of each tuple argument, and where $c_1 \sqcup c_2$ denotes the standard least-upper-bound of c_1 and c_2. As used here, that means $\lambda id.\ (c_1\ id) \cup (c_2\ id)$. For now treat the domain construction $D \boxtimes Cache$ as just $D \times Cache$; we return to its precise definition later.

The equations for $\hat{\mathcal{E}}_{1a}$ should be fairly self-explanatory. $\hat{\mathcal{E}}_{1a}\ lab\ bve\ fve$ returns a pair containing the standard denotation together with a cache containing a "history" of the evaluation of $expr(lab)$. This history is gathered by adding to the cache the value of every expression as it is computed. It is fairly easy to prove that this semantics is consistent with the standard one.

5 First-Order Language, Normal-Order Semantics

We next consider an abstract language whose syntax is identical to that given in the last section, but which we now interpret using normal-order semantics (i.e., lazy evaluation).

5.1 Standard Normal-Order Semantics for First-Order Programs

This semantics is identical to that given earlier, except that the equation for fve:

$$fve = \bot[\ strict(\lambda(y_1...y_n).\ \mathcal{E}_{1a}[\![e_i]\!]\ \bot[y_j/x_j]\ fve)\ /\ f_i\ ]$$

is changed to:

$$fve = \bot[\ (\lambda(y_1...y_n).\ \mathcal{E}_{1a}[\![e_i]\!]\ \bot[y_j/x_j]\ fve)\ /\ f_i\ ]$$

In other words, the functions are not forced to be strict.

5.2 Normal-Order Collecting Interpretation for First-Order Programs

The key change to the collecting interpretation strategy derives from the observation that we must not merge the cache resulting from the evaluation of an argument to the cache resulting from a function call, until the corresponding bound variable is evaluated (if in fact it is ever evaluated). This change is easily made by adding the argument cache to the bound variable environment, and then extracting the necessary information when the variable is evaluated. Note the change in functionality of Bve and Fve.

$$\begin{array}{lcll} Bve & = & Bv \to Ans & \star \\ Fve & = & Fv \to (Ans* \to Ans) & \\ Ans & = & D \boxtimes Cache & \\ Cache & = & Lab \to \mathcal{P}(D) & \end{array}$$

$$\begin{array}{ll} \hat{\mathcal{E}}_{1n}: & Lab \to Bve \to Fve \to Ans \\ \hat{\mathcal{P}}_{1n}: & Prog \to Ans \\ \hat{\mathcal{K}}_{1n}: & Con \to (Ans* \to Ans) \end{array}$$

$$\begin{array}{rll}
\multicolumn{3}{l}{\hat{\mathcal{E}}_{1n}\ lab\ bve\ fve = \text{case } expr(lab) \text{ of}} \\
[\![k]\!]: & \langle \mathcal{A}_{1n}[\![k]\!], \bot[\{\mathcal{A}_{1n}[\![k]\!]\}/lab]\rangle & \\
[\![x]\!]: & \text{let } \langle d, c\rangle = bve[\![x]\!] & \\
 & \text{in } \langle d, c \sqcup \bot[\{d\}/lab]\rangle & \star \\
[\![p(e_1...e_n)]\!]: & \text{let } e_i' = \hat{\mathcal{E}}_{1n}\ label[\![e_i]\!]\ bve\ fve & \\
 & \quad \langle d, c\rangle = \hat{\mathcal{K}}_{1n}[\![p]\!](e_1'...e_n') & \\
 & \text{in } \langle d, c \sqcup \bot[\{d\}/lab]\rangle & \\
[\![f(e_1...e_n)]\!]: & \text{let } e_i' = \hat{\mathcal{E}}_{1n}\ label[\![e_i]\!]\ bve\ fve & \\
 & \quad \langle d, c\rangle = fve[\![f]\!](e_1'...e_n') & \\
 & \text{in } \langle d, c \sqcup \bot[\{d\}/lab]\rangle &
\end{array}$$

$$\begin{array}{l} \hat{\mathcal{P}}_{1n}[\![\{\ f_i(x_1...x_n) = e_i\}]\!] = fve[\![f_1]\!] \\ \quad \text{whererec } fve = \bot[(\lambda(y_1...y_n).\ \hat{\mathcal{E}}_{1n}\ label[\![e_i]\!]\ \bot[y_j/x_j]\ fve)\ /\ f_i\ ] \quad \star \end{array}$$

The three lines marked with a star indicate the only changes from the previous collecting interpretation. In some sense the result is actually simpler than the previous one, since there is no need to "force" the merging of the argument caches, just as in the new standard semantics there is no need to "force" the strict evaluation of arguments.

6 Higher-Order Language, Normal-Order Semantics

We now arrive at a language with the full power of untyped lambda calculus with constants. Its abstract syntax is given by:

$$\begin{array}{rcll} l & \in & \mathit{Lab} & \text{labels} \\ k & \in & \mathit{Con} & \text{constants} \\ x, f & \in & \mathit{Id} & \text{identifiers, either bound variables or function names} \\ e & \in & \mathit{Exp} & \text{expressions, where } e \leftarrow l.k \mid l.x \mid l.f \mid l.(\lambda x.e) \mid l.(e_1\ e_2) \\ pr & \in & \mathit{Prog} & \text{programs, where } pr \leftarrow \{\ f_i = e_i\ \} \end{array}$$

and again we assume that all labels in a program $pr \in \mathit{Prog}$ are unique.

6.1 Standard Normal-Order Semantics for Higher-Order Programs

We will again assume a domain D whose structure depends on Con, but now it will typically be the solution of a reflexive domain equation such as $D = \mathit{Int} + \mathit{Bool} + (D \to D)$.

$$\mathit{env} \in \ \mathit{Env} = \mathit{Id} \to D$$

$$\begin{array}{ll} \mathcal{E}_h : & \mathit{Lab} \to \mathit{Env} \to D \\ \mathcal{P}_h : & \mathit{Prog} \to D \end{array}$$

$$\begin{array}{l} \mathcal{E}_h\ \mathit{lab}\ \mathit{env} = \text{case } \mathit{expr}(\mathit{lab}) \text{ of} \\ \quad \begin{array}{rl} [\![k]\!] : & \mathcal{K}_h[\![k]\!] \\ [\![x]\!] : & \mathit{env}[\![x]\!] \\ [\![\lambda x.e]\!] : & \lambda y.\ \mathcal{E}_h\ \mathit{label}[\![e]\!]\ \mathit{env}[y/x] \\ [\![e_1\ e_2]\!] : & (\mathcal{E}_h\ \mathit{label}[\![e_1]\!]\ \mathit{env})\ (\mathcal{E}_h\ \mathit{label}[\![e_2]\!]\ \mathit{env}) \end{array} \end{array}$$

$$\begin{array}{l} \mathcal{P}_h[\![\{\ f_i = e_i\ \}]\!] = \mathit{env}[\![f_1]\!] \\ \qquad \text{whererec } \mathit{env} = \bot[\ \mathcal{E}_h\ \mathit{label}[\![e_i]\!]\ \mathit{env}\ /\ f_i\ ] \end{array}$$

As is the first-order semantics, this semantics is quite conventional.

6.2 Normal-Order Collecting Interpretation for Higher-Order Programs

The introduction of higher-order functions necessarily complicates our collecting interpretation somewhat, because now we must take into account the fact that the application of the value of some expression might induce other values to be added to the cache. We solve this problem in a way similar to our solution of other higher-order inferencing strategies [9,11] – that is, we add a higher-order function to the domain of our answers. In particular, the domain $Ans = D \boxtimes Cache$ in the previous analysis becomes $Ans = D \boxtimes Cache \boxtimes (Ans \to Ans)$ in the new analysis, and the environments must map identifiers to Ans. The result follows.

$$\begin{array}{lcl} Env & = & Id \to Ans \\ Ans & = & D \boxtimes Cache \boxtimes (Ans \to Ans) \\ Cache & = & Lab \to \mathcal{P}(D) \end{array}$$

$$\begin{array}{ll} \hat{\mathcal{E}}_h: & Lab \to Env \to Ans \\ \hat{\mathcal{P}}_h: & Prog \to Ans \\ \hat{\mathcal{K}}_h: & Con \to Ans \to Ans \end{array}$$

$$\begin{array}{rll}
\multicolumn{3}{l}{\hat{\mathcal{E}}_h\ lab\ env = \text{case } expr(lab) \text{ of}} \\
[\![k]\!]: & \text{let } d = \mathcal{K}_h[\![k]\!] & \\
 & \text{in } \langle d, \bot[\{d\}/lab], \hat{\mathcal{K}}_h[\![k]\!]\rangle & \\
[\![x]\!]: & \text{let } \langle d, ds, f\rangle = env[\![x]\!] & \\
 & \text{in } \langle d, ds \sqcup \bot[\{d\}/lab], f\rangle & \\
[\![\lambda x.e]\!]: & \text{let } f = \lambda z{:}Ans.\ \hat{\mathcal{E}}_h\ label[\![e]\!]\ env[z/x] & \\
 & \quad\ d = \lambda y{:}D.\ \mathcal{E}_h\ label[\![e]\!]\ (\lambda id.\ (env\ id)\downarrow 1)[y/x] & \\
 & \text{in } \langle d, \bot[\{d\}/lab], f\rangle & \\
[\![e_1\ e_2]\!]: & \text{let } \langle d_1, c_1, f_1\rangle = \hat{\mathcal{E}}_h\ label[\![e_1]\!]\ env & \\
 & \quad\ \langle d_2, c_2, f_2\rangle = e_2' = \hat{\mathcal{E}}_h\ label[\![e_2]\!]\ env & \\
 & \quad\ \langle d, c, f\rangle = f_1\ e_2' & (\text{note}: d = d_1\ d_2) \\
 & \text{in } \langle d, c_1 \sqcup c \sqcup \bot[\{d\}/lab], f\rangle &
\end{array}$$

$$\begin{array}{l} \hat{\mathcal{P}}_h[\![\{f_i = e_i\}]\!] = env[\![f_1]\!] \\ \quad \text{whererec } env = \bot[\ \hat{\mathcal{E}}_h\ label[\![e_i]\!]\ env\ /\ f_i\ ] \end{array}$$

Note that in the equation for $[\![\lambda x.e]\!]$ there is a call to $\mathcal{E}_h$ in an environment derived from env that "simulates" the standard environment. Although this lone call to $\mathcal{E}_h$ is used only to create the D-value for $[\![\lambda x.e]\!]$, it can actually be removed in the following (albeit devious) way. First define g recursively by:

$$g = \lambda y{:}D.\ \langle y, \bot, \lambda\langle d, c, f\rangle.\ g\ (y\ d)\rangle$$

Thus g essentially "coerces" its argument in D into an element in Ans. We then redefine d in the equation for $[\![\lambda x.e]\!]$ as:

$$d = \lambda y{:}D.\ (f\ (g\ y))\downarrow 1$$

In this way $\hat{\mathcal{E}}_h$ is being used not only to cache results, but also to simulate the standard semantics completely, as we did for the first-order cases. A point related to all this is the observation that in the equation for $[\![e_1\ e_2]\!]$, the equality $d = (d_1\ d_2)$ holds.

For completeness, we provide a partial specification of $\hat{\mathcal{K}}_h$:

$$\begin{aligned}\hat{\mathcal{K}}_h[\![if]\!] = \lambda\langle d_p, c_p, f_p\rangle.\ &\text{if } d_p \text{ then } \langle\ \lambda c\ a.\ c,\ c_p,\\ &\qquad\qquad \lambda\langle d_c, c_c, f_c\rangle.\ \langle\ \lambda a.\ d_c,\ c_c,\\ &\qquad\qquad\qquad \lambda\hat{a}.\ \langle d_c, \perp, f_c\rangle\rangle\rangle\\ &\text{else } \langle\ \lambda c\ a.\ a,\ c_p,\\ &\qquad \lambda\langle d_c, c_c, f_c\rangle.\ \langle \lambda a.\ a,\ \perp,\ \lambda\hat{a}.\hat{a}\rangle\rangle\end{aligned}$$

$$\begin{aligned}\hat{\mathcal{K}}_h[\![+]\!] = \lambda\langle d_1, c_1, f_1\rangle.\ &\langle\ \lambda d.\ d_1 + d,\ c_1,\\ &\quad \lambda\langle d_2, c_2, f_2\rangle.\ \langle d_1 + d_2,\ c_2,\ err\rangle\rangle\end{aligned}$$

6.3 Correctness

In what sense are our collecting interpretations "correct"? In this section we explore answers to this question for the higher-order analysis – similar results hold for the first-order case.

First of all, there is the question of obtaining a deterministic result – that is, a unique least fixpoint of the semantic equations. It so happens that if the domain construction $D_1 \boxtimes D_2$ is interpreted to mean $D_1 \times D_2$, the conventional cross product of two domains, then a monotonicity problem arises, because as approximations (such as $\perp$) become refined to true answers, the weaker elements must drop out of the cache – the "dropping out" is what causes the non-monotonicity. This same problem arises in applications involving non-determinism, and the typical solution is to resort to a suitable powerdomain construction. Fortunately, there is a far simpler solution in our context: Simply define the special domain construction such that if $\langle d_1, c_1\rangle, \langle d_2, c_2\rangle \in (D \boxtimes Cache)$, then $\langle d_1, c_1\rangle \sqsubseteq \langle d_2, c_2\rangle$ iff $d_1 \sqsubseteq d_2$. In other words, we essentially *ignore* the cache when computing the fixpoint. The cache that results will then need to be shown "correct" independently, but for now let us first state a small theorem about the fixpoint analysis:

Theorem 1: For all finite programs $pr \in Prog$, $(\hat{\mathcal{P}}_h\ pr)\downarrow 1 = \mathcal{P}_h\ pr$.

Proof: By structural induction on pr, and fixpoint induction on the semantic equations (details are omitted in this summary). The proof depends upon the following property of $\hat{\mathcal{K}}_h$:

$$(\hat{\mathcal{K}}_h[\![c]\!]\ e_1 ... e_n)\downarrow 1 = \mathcal{K}_h[\![c]\!]\ (e_1\downarrow 1)...(e_n\downarrow 1),\ n > 0$$

which is easily proved for the partial specifications of $\mathcal{K}_h$ and $\hat{\mathcal{K}}_h$ given, and is assumed to be true of the remaining specifications. □

Now the question returns to what can be said about the cache. This turns out to be a subtle problem, since one can define several kinds of caches, depending on one's intuition about what it means for a program to be "evaluated." A full discussion of this issue is beyond the scope of this paper, but we wish to point out that there are at least three kinds of collecting interpretations that we think are worth distinguishing, and their behavior can be captured by considering the expression $bot_1 + bot_2$, where the bot_i are arbitrary expressions that happen not to terminate in the "current" environment. Then the three types of collecting interpretations can be described as:

1. A *sequential* collecting intepretation is one that mimics a sequential interpreter. For the above example such a collection would result in a cache that has values for subexpressions in bot_1, but not bot_2, assuming left-to-right evaluation.

2. A *parallel* collecting interpretation is one that mimics a parallel interpreter – that is, one that evaluates in parallel all arguments that it "knows are needed." Thus for the above example "contributions" from both bot_1 and bot_2 should appear in the cache.

3. A *dependent* collecting interpretation is one that includes in the cache only those values that can effect the final answer. Interestingly, in the above example there is no single value that can effect the outcome (since the outcome is always bottom) and thus the cache should be empty!

All three of the collecting interpretations defined in this paper are *parallel* ones. This was done primarily for simplicity, since one has to add machinery to check for bottom values in order to achieve one of the other two.

From this discussion it should be clear that the behavior of a collecting interpretation can be pretty much whatever we *define* it to be. However, there is at least one notion of correctness that we think should be captured by any collecting interpretation; namely, if a program terminates with an atomic result (i.e., not a function), then the cache should contain any values that the result "depends" on. This notion of dependence can be formalized in the following way:

Let G be the functional describing $\mathcal{E}_h$; i.e., $\mathcal{E}_h = fix\ G$ (the precise definition of G is easily derived from the equation defining $\mathcal{E}_h$ given earlier). Then define G' by:

$$\begin{aligned} G'\ E\ lab'\ env' = \ & \text{if } (lab' = lab) \wedge (env' = env) \\ & \text{then } \perp \\ & \text{else } G\ E\ lab'\ env' \end{aligned}$$

and let $\mathcal{E}_h' = fix\ G'$. Thus $\mathcal{E}_h'$ is just like $\mathcal{E}_h$ except at point $\langle lab, env\rangle$, where it returns the value $\perp$. Further, let $\mathcal{P}_h'$ be derived from $\mathcal{E}_h'$ just as $\mathcal{P}_h$ is derived from $\mathcal{E}_h$.

Definition: A program pr is said to *depend on* the value of $lab.exp$ in the environment env if and only if $\mathcal{P}'_h \neq \mathcal{P}_h$.

In other words, if we can change the behavior of a program pr by causing exp in environment env to diverge, then it must be the case that pr depends on that evaluation.[4] This leads to a second theorem:

Theorem 2: Given any finite program $pr \in Prog$, let $\langle d, c, f \rangle = \mathcal{P}_h\ pr$, and assume $d \neq \perp$ and $d \notin (D \to D)$. Then if pr depends on the value of $lab.exp$ in environment env, then $(\mathcal{E}_h\ lab\ env) \in c(lab)$.

Proof: (Omitted.)

The reason for the constraint $d \notin (D \to D)$ relates to the definition of dependence, and will be discussed more fully in a future paper.

7 Discussion

We should point out that one may collect not only all *values* that a particular expression evaluates to, but also all *environments* that it was evaluated in. This is a straightforward extension of any of the collecting interpretations given, and similar to saving all argument tuples in an mfg interpretation. It may be useful in the following sense: suppose $l_1.e_1$ and $l_2.e_2$ are expressions within the same lexical environment, and let $S_1 = cache(l_1)$ and $S_2 = cache(l_2)$. Then if we wish to ask what all *pairs* of values possessed by e_1 and e_2 are during program execution, the best answer we can currently give is $S_1 \times S_2$; i.e., the cartesian product of the two sets. But this may be inaccurate in that certain of those pairs may not have really occurred. This is exactly the same distinction made between the "independent attribute" and "relational attribute" methods discussed in [13] and later in [16].

There are two ways to fix this problem, both rather straightforward. The first method collects, in addition to the *value* of an expression, the *bound variable environment* in effect when the expression is evaluated. That is, the functionality of the cache is changed to $Lab \to \mathcal{P}(D \times Bve)$. Thus the answer to the previous question would be:

$$S = \{\ \langle d_1, d_2 \rangle \mid \langle d_1, bve_1 \rangle \in S_1,\ \langle d_2, bve_2 \rangle \in S_2,\ and\ bve_1 = bve_2\ \}$$

The necessary changes to accomplish this for each of the collecting interpretations given earlier are straightforward, and the details are left to the reader.

Alternatively, one could name each lexical environment explicitly (say with names from some syntactic domain Lex), and return a cache with functionality: $Lex \to \mathcal{P}(Lab \to D)$.

[4]This can be thought of as a generalized notion of "strictness" [11].

In this setting the answer to the previous question would be:

$$S = \{\ \langle f\ l_1,\ f\ l_2\rangle \mid f \in (cache\ lex)\}$$

where lex is the name of the lexical environment containing e_1 and e_2. This method has the advantage of avoiding the test $bve_1 = bve_2$,[5] but it provides no more power than the previous method. For this reason, and since the changes necessary to invoke this method are only slightly more complex than the previous method, the details are omitted.

8 Abstract Collecting Interpretations of Expressions

Abstractions, of course, may be made of any of the previous collecting interpretations. Preliminary versions of such abstractions have been developed for strictness analysis (thus solving the problem mentioned in Section 3.3) and reference counts [10]; a future paper will detail these applications. Another application may be found in [2].

References

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

[2] A. Bloss and P. Hudak. Path semantics. *submitted to the Third Workshop on the Mathematical Foundations of Programming Language Semantics*, November 1986.

[3] A. Bloss and P. Hudak. Variations on strictness analysis. In *Sym. on Lisp and Functional Programming*, pages 132–142, ACM, August 1986.

[4] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1985.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Sym. on Prin. of Prog. Lang.*, pages 238–252, ACM, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Sym. on Prin. of Prog. Lang.*, pages 269–282, ACM, 1979.

[5] Actually, this equality test is too strong in the higher-order case, where it is only necessary that the environments be *comparable* to account for the possibility of their being nested.

[7] V. Donzeau-Gouge. Denotational definition of properties of program computations. In *Program Flow Analysis: Theory and Applications*, pages 343–379, Prentice-Hall, 1981.

[8] Jon Fairbairn and Stuart C. Wray. Code generation techniques for functional languages. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 94–104, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986.

[9] B. Goldberg and P. Hudak. *Inferring sharing properties of partial applications in higher-order functional languages.* Research Report in preparation, Yale University, Department of Computer Science, August 1986.

[10] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Sym. on Lisp and Functional Programming*, pages 351–363, ACM, August 1986.

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

[12] J. Hughes. Strictness detection in non-flat domains. In *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1986.

[13] N.D. Jones and S.S. Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to dijkstra. In *Program Flow Analysis: Theory and Applications*, pages 380–393, Prentice-Hall, 1981.

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

[15] G. Lindstrom. Static evaluation of functional programs. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 196–206, ACM, June 1986. Published as SIGPLAN Notices Vol. 21, No. 7, July 1986.

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

[17] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. of Int. Sym. on Programming*, pages 269–281, Springer-Verlag LNCS Vol. 83, 1980.

[18] A. Mycroft and F. Nielson. Strong abstract interpretation using powerdomains. In *Proc. ICALP, Springer Verlag LNCS No. 154*, pages 536–547, 1983.

[19] F. Nielson. *Abstract Interpretation Using Domain Theory.* PhD thesis, University of Edinburgh, October 1984.

[20] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[21] P. Panangaden and P. Mishra. *A category theoretic formalism for abstract interpretation.* Technical Report UUCS-84-005, University of Utah, 1984.

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

Don't copy this.

Collecting Interpretations of Expressions
(Preliminary Version)

Paul Hudak

August 1986

Yale University
Department of Computer Science
Research Report YALEU/DCS/RR-497
Box 2158 Yale Station
New Haven, CT 06520
Arpanet: hudak@yale