Collecting Interpretations of Expressions

PAUL HUDAK
Yale University
and
JONATHAN YOUNG
Massachusetts Institute of Technology Laboratory for Computer Science

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 an expression might evaluate during program execution?" Answering such questions in a denotational framework is akin to traditional data flow analysis and, when used in the context of abstract interpretation, allows one to infer properties that approximate the run-time behavior of expression evaluation.

Exact collecting interpretations of expressions are developed for three abstract functional languages: a strict first-order language, a nonstrict first-order language, and a nonstrict higher order language (the full untyped lambda calculus with constants). It is argued that the method is simple (in particular, no powerdomains are needed), natural (it captures the intuitive operational behavior of a cache), yet more expressive than existing methods (it is the first exact collecting interpretation for either nonstrict or higher order languages). Correctness of the interpretations with respect to the standard semantics is shown via a generalization of the notion of strictness. It is further shown how to form abstractions of these exact interpretations, using as an example a collecting strictness analysis which yields compile-time information not previously captured by conventional strictness analyses.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classification—*applicative languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*denotational semantics*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms: Languages, Theory

1. INTRODUCTION

In recent years there has been a trend toward using formal programming language semantics, usually denotational semantics, as a basis for designing compilers. This has the advantage of providing a formal basis with which to

This paper is an expanded version of one that appeared in the *Proceedings of the 1988 ACM Symposium on Principles of Programming Languages*. The research was supported in part by the National Science Foundation under grant DCR-8451415 and in part by the Department of Energy under grant DE-FG02-86ER25012.

Authors' addresses: P. Hudak, Department of Computer Science, Yale University, Box 2158, Yale Station, New Haven, CT 06520; J. Young, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

reason about the correctness of compilers, as well as offering opportunities to build "language independent" compilers, and indeed "compiler compilers."

Besides using formal semantics to guide code generation, formal semantics has also been used to guide optimization methods. The idea behind this approach is as follows: First an optimization is shown to depend on a particular dynamic property of a program, and this dependence is expressed formally using either the standard denotational semantics of the language or possibly a nonstandard one (i.e., an operational semantics). These dynamic properties are of course almost always undecideable, and so, just as with traditional program analysis, approximations (or "abstractions") are required to make the analysis computable, thus allowing safe but effective use of the optimizations. In the framework of denotational semantics this analysis technique has become known as abstract interpretation and was first introduced by Cousot and Cousot in an imperative language framework [3, 4].

Abstract interpretation has been shown to be an effective methodology for realizing many compile-time analyses of programs. Beginning with the Cousots' seminal work in an imperative language setting, many applications have been proposed and implemented, mostly within the functional language paradigm. Examples include strictness analysis on flat domains [2, 10, 18, 19], strictness analysis on nonflat domains [11, 16, 24], reference counting [9], sharing of partial applications [7], various data-flow analyses [3, 21], and applications in logic programming [15, 17]. An excellent summary of recent results may be found in Abramsky and Harkin [1].

Surprisingly, despite the preponderance of work on abstract interpretation for functional languages, one of the Cousots' original goals has been for the most part ignored: the desire to "collect properties about program points." We discuss this issue in detail in Section 2, and in Sections 3 through 5 we present a *collecting interpretation of expressions* for several classes of functional languages, similar in power to interprocedural dataflow analysis of imperative programs, and satisfying for functional languages the Cousots' original goal for imperative languages. Intuitively, such an interpretation simulates execution of a program, recording the result of evaluating each expression in every context that arises.

We first present *exact* (and thus generally uncomputable) collecting interpretations, in which most common difficulties with powerdomains can be avoided. In Section 6 we show the uniqueness of the collecting interpretations, and, through a novel generalization of the notion of function strictness, we establish the correctness of the interpretations with respect to the standard semantics. Given the exact collecting interpretation as a theoretical basis, we then show in Section 7 how it can be abstracted, using as an example strictness analysis, thus yielding useful compile-time information which has not previously been available to denotational semantics based compilers.

1.1 Previous Work

We begin with a brief survey of previous work, while attempting to clarify some of the inconsistent terminology found in the literature.

The Cousots' original paper [3] on abstract interpretation presented a static semantics which associated with each arc in a program control flow graph a set of possible run-time environments which could be associated with that arc under all possible executions of the program. This idea is consistent with our own, except that the term "static semantics" is more commonly used today to refer to the static constraints imposed by a particular language's standard semantics (such as type-checking). In addition, the Cousots' work was done in an imperative language framework where the idea of "program point" is well defined (whereas in a functional framework it is not), and the work was restricted to a flat value domain whose approximations are supersets or subsets of the actual set of values (thus avoiding problems with powerdomains). The Cousots' work was notable in that it was the first to show that most program flow analyses can be unified under the framework of abstract interpretation.

Mycroft [18] later applied the Cousots' ideas to the problem of performing strictness analysis of first-order functional programs. For such an analysis the notion of a "program point" was abandoned, settling instead on abstract properties of functions. To formalize this, functions with type $D_1 \to D_2$ were "lifted" to functions of type $P(D_1) \to P(D_2)$, where P, because of domain considerations, must be a powerdomain constructor. A suitable abstraction of this semantics leads to a strictness analysis.

Nielson [21] coined the term *collecting semantics* for a semantics which describes all possible states at some program point, as with the Cousots' analysis. Since this term is highly descriptive and is consistent with our own use, we have adopted it. Nielson goes on to form abstractions of the collecting semantics, which are called static semantics in the same sense as the Cousots' work, and in which functions are "lifted" in the same way as with Mycroft's work. Nielson's work was elaborated upon considerably in [20].

In related work, Jones and Mycroft [14] designed a minimal function graph (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. This is achieved by simulating the call/return behavior of function calls. In addition, they extend the base value 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 nontermination in the classical sense. This same approach was used in [9] to capture abstractions to an exact reference counting semantics, but there nontermination was not an issue so the two "bottoms" were synonymous.

1.2 Overview

The collecting interpretations investigated in this paper are considered within the framework of functional languages, where the notion of a "program point" refers, quite simply, to an *expression*, and thus the result is called a *collecting interpretation of expressions*. We develop three such interpretations: (1) $\tilde{\mathscr{E}}_{1s}$, for a strict first-order language, (2) $\tilde{\mathscr{E}}_{1n}$, for a nonstrict first-order language, and (3) $\tilde{\mathscr{E}}_{h}$, for a nonstrict higher-order language (the full untyped

lambda calculus with constants). $\tilde{\mathscr{E}}_{1s}$ is similar in power to an MFG semantics, whereas $\tilde{\mathscr{E}}_{1n}$ and $\tilde{\mathscr{E}}_{h}$ are new developments. In all three cases the methodology used is relatively straightforward and is easily related to the standard semantics. Also, although functional languages are used as the source, the technique is easily generalized to denotational interpretations of other (i.e., imperative) languages as well.

One of the key aspects of our technique is that it obviates the need to "lift" functions to corresponding ones over the powerdomains of the original domains, as described earlier. Thus we are able to avoid many of the problems and (if nothing else) technical complexities of handling powerdomains. As some indication of this complexity, the abstract interpretation found in Mycroft [18] contains a subtle error due to powerdomain considerations; see Nielson [20] for a discussion of the problem and a proposed solution.

1.3 Notation

Most of our notation is that of standard denotational semantics. A domain D is a *pointed cpo*—a chain-complete partial order with a least element $\perp_D$ (called "bottom") and whose binary ordering relation is denoted $\sqsubseteq_D$. The infix least upper bound (lub) operator for the domain D is written $\sqcup_D$; its prefix form, which computes the lub of a set of elements, is denoted $\bigsqcup_D$. Thus we have that for all $d \in D$, $\perp_D \sqsubseteq_D d$ and $\perp_D \sqcup d = d$. Domain subscripts are often omitted, as in $\perp \sqsubseteq d$, when they are clear from context.

The notation "$d \in D = \cdots$" defines the domain (or set) D with "typical element" d, where $\cdots$ provides the domain specification usually via some combination of the following domain constructions: $D_1 \xrightarrow{t} D_2$ denotes the domain of all *total* functions from D_1 to D_2, and $D_1 \to D_2$ is the domain of all *continuous functions* from D_1 to D_2 (the latter is contained within the former). $D_1 + D_2$ and $D_1 \times D_2$ denote the separated sum and product, respectively, of the domains D_1 and D_2. All domain/subdomain coercions are omitted when clear from context.

The ordering on functions $f, f' \in D_1 \xrightarrow{t} D_2$ is defined in the standard way: $f \sqsubseteq f' \Leftrightarrow (\forall d \in D_1) f d \sqsubseteq f' d$. A function $f \in D_1 \xrightarrow{t} D_2$ is *monotonic* iff it satisfies $(\forall d, d' \in D_1) d \sqsubseteq d' \Rightarrow f d \sqsubseteq f d'$; it is *continuous* if in addition it satisfies $f(\bigsqcup \{d_i\}) = \bigsqcup \{f d_i\}$ for any chain $\{d_i\} \subseteq D_1$. A function $f \in D_1 \to D_2$ is said to be *strict* if $f \perp_{D_1} = \perp_{D_2}$. An element $d \in D$ is a *fixpoint* of $f \in D \to D$ iff $f d = d$; it is the *least fixpoint* if for every other fixpoint d', we have that $d \sqsubseteq d'$. The notation f^n denotes the n-way composition of f.

Angle brackets are used for tupling. If $d_i \in D_i$, for $1 \le i \le n$, then $\langle d_1, \ldots, d_n \rangle \in D_1 \times \cdots \times D_n$. The domain $\mathbf{1}$ contains the single element $\langle\rangle$. A^* denotes the domain $\mathbf{1} + A + (A \times A) + \cdots$ with implicit coercions. It is convenient to write n-ary curried functions as $\lambda x_1\ x_2 \cdots x_n \cdot exp$, and when $n = 0$ this is interpreted to mean just exp. Syntactic objects are consistently enclosed in double brackets, as in $[\![exp]\!]$. Square brackets are used for environment update, as in $env[e/[\![x]\!]]$, which is equivalent to the function $\lambda id.$ if $id = [\![x]\!]$, then e else $env\ id$. 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. "New" environments are created by $\perp[e_i/[\![x_i]\!]]$.

There are several well-known facts concerning domains that we state without proof.

THEOREM 1. *Every continuous function* $f \in D \to D$ *has a least fixpoint,* $fix\, f$, *where* $fix = \lambda f.\ \sqcup\{f^n(\perp) \mid n \geq 0\}$.

THEOREM 2. *The composition of monotonic* [*continuous*] *functions is also monotonic* [*continuous*].

THEOREM 3. *The domain constructors* $\to$, $+$, *and* $\times$, *together with their operators, preserve monotonicity and continuity.*

2. ON THE NATURE OF COLLECTING INTERPRETATIONS

Intuitively, what we desire as the "collecting interpretation" of a program is a function, call it *cache*, such that $cache[\![exp]\!]$ returns the set of all possible values that the expression $[\![exp]\!]$ could evaluate to during program execution. Doing this runs immediately into a small technical difficulty, namely, finding a way to uniquely reference each expression. We solve this problem by assuming that each expression has a unique *label* from a primitive, flat syntactic domain *Lab*. A labeled expression is written $[\![l.e]\!]$, where $l \in Lab$, but we often omit the label when its presence is not needed. (Labels serve the same purpose as *occurrences* and *places* as used in [5, 18, 21].) Thus, our cache should have functionality $Lab \to \mathcal{P}(D)$, if D is assumed to be the domain of values to be collected. $\mathcal{P}(D)$ denotes the *powerset* of D (i.e., not a powerdomain). We will, however, make use of the fact that $\mathcal{P}(D)$ is also 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 for a collecting interpretation can be made clear by a simple example. Consider the program *pr* given by

$$[\![\ \text{if } \mathit{true} \text{ then } l_1.(f\,1) \text{ else } l_2.(f\,2),\ \text{where } f = \lambda x.\, x\ ]\!]$$

and suppose *cache* is the result of a collecting interpretation of expressions for *pr*. Then $cache(l_1) = \{1\}$, but $cache(l_2) = \{\}$, because $[\![f\,2]\!]$ is "never used"[1] during program execution. Thus, unlike most other domains, the bottom element $\perp_{\mathcal{P}(D)} = \{\}$ does not denote nontermination, but rather indicates the absence of any evaluation of the expression at all. On the other hand, elements of $\mathcal{P}(D)$ may contain $\perp_D$, indicating that one possible value of the expression is nontermination. This aspect of our collecting interpretation contrasts with, for example, the semantics of nondeterminism, where *at least one result* is (usually) expected, and thus the bottom element is typically $\{\perp\}$.[1]

The above behavior becomes even more pronounced in a language with lazy evaluation. In particular, just because one occurrence of a bound variable is evaluated does not mean that another is, and that is one reason why

[1] We make this phrase precise in Section 6.2.

labels are necessary—to distinguish the different occurrences. For example, in the program *pr* above, if f were defined by

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

then $cache(l_3) = cache(l_4) = \{1\}$, but $cache(l_5) = \{\}$.

2.1 Sets as Domains Can Be Problematical

The use of sets in a collecting interpretation seems rather intuitive, and relatively innocuous. However, monotonicity (and thus continuity) is easily lost, as demonstrated by the following simple example:

$$setify: D \to \mathcal{P}(D)$$
$$setify\, d = \{d\}.$$

The function *setify* simply maps its single argument into a singleton set, but note that it is not monotonic: even though $d_1 \sqsubset d_2$, $setify\, d_1 \not\sqsubseteq setify\, d_2$, since $\{d_1\} \not\subseteq \{d_2\}$. In a fixpoint semantics this can be devastating, since the usual least fixpoint construction depends critically on monotonicity.

This unfortunate behavior is what prompted the development of *powerdomains*, which generally fix the problem either by "flattening" the domain sufficiently (for example, by discarding $\perp$), or by preserving both the partial order on the sets and the partial order on the elements of the sets by quotienting the powerset based on the equivalence classes induced by a combined partial order (actually a preorder). However, there are several ways of combining the partial orders, and doing so in the obvious ways regains monotonicity but not necessarily continuity, thus requiring further nontrivial refinements. (A good overview of powerdomains may be found in Schmidt [22].)

The resulting general powerdomain constructions are relatively complex, may be difficult to reason about, and none can be said to be general enough to satisfy the requirements of all situations in which sets of results are needed. Indeed, as discussed earlier, in a collecting interpretation we would like for the least element to be the empty set, whereas in conventional powerdomain constructions meeting our partial order constraints, the least element is the singleton set containing bottom. Powerdomains that collapse elements that are "downwards closed" are also inadequate, since a proper collecting interpretation must keep those elements distinct (since they may represent two different results). For these reasons, extending Jones and Mycroft's MFG semantics, for example, in the obvious way to lazy and higher order programs run immediately into nontrivial powerdomain problems [12].

Our strategy for extending collecting interpretations to lazy and higher order programs is to avoid the use of powerdomains altogether; more specifically:

(1) We do not insist that all functions be monotonic or continuous on all arguments. This leaves on us the burden of proving the existence of a least fixpoint, but the following technique makes such a proof relatively straightforward.

(2) We effectively separate the recursive specification of the standard interpretation from that of the collecting interpretation.[2] This eliminates the need for a "combined partial order," and guarantees a unique least fixpoint in each recursive definition independently, using standard arguments.

2.2 Collecting Interpretations Are Operational

There are several interesting aspects of collecting interpretations which suggest that they are very operational in nature. A discussion of these characteristics is worthwhile prior to a formal development of the collecting interpretations themselves.

Collecting Interpretations Model Interpreters. It is possible to define several kinds of collecting interpretations, depending on one's intuition about what it means for an expression to be "evaluated." For example, consider the expression $[\![bot_1 + bot_2]\!]$, where the $[\![bot_i]\!]$ are arbitrary expressions which happen not to terminate in the "current" environment. With this example alone, it is possible to distinguish three kinds of collecting interpretations, defined intuitively below:

(1) A *sequential* collecting interpretation is one which mimics a sequential interpreter. For the above example, values for $[\![bot_1]\!]$ and its subexpressions would be collected, but not for $[\![bot_2]\!]$, if we assume left-to-right evaluation.

(2) A *parallel* collecting interpretation is one which mimics a parallel interpreter—that is, one which evaluates in parallel all arguments in a function call. 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 which includes only those values which contribute effectively to the final answer. Interestingly, in the above example there are no such values (since the outcome is always bottom) and thus the collection should be empty.

All three of the collecting interpretations defined in this paper are *parallel* ones. This is done primarily for simplicity, since extra machinery is needed to check for bottom values in the other two cases.

Functionals Capture Operational Behavior. Consider a standard semantics $\mathscr{E}$ with the following clause:

$$\cdots$$

$$\mathscr{E}[\![e_1 + e_2]\!]env = plus(\mathscr{E}[\![e_1]\!]env)(\mathscr{E}[\![e_2]\!]env)$$

$$\cdots$$

[2] In a previous version of this paper [9] we essentially used *predomains* to effect the separation, but we feel that the current approach is simpler.

versus one called $\mathscr{E}'$ which has an additional clause to recognize the expression $[\![1 + 1]\!]$:

$$\begin{aligned} &\cdots \\ &\mathscr{E}'[\![1 + 1]\!]env = 2 \\ &\mathscr{E}'[\![e_1 + e_2]\!]env = plus(\mathscr{E}[\![e_1]\!]env)(\mathscr{E}[\![e_2]\!]env) \\ &\cdots \end{aligned}$$

Assuming that $plus(\mathscr{E}[\![1]\!]env)(\mathscr{E}[\![1]\!]env) = 2$, these two semantics can easily be shown to be equivalent (i.e., $\mathscr{E} = \mathscr{E}'$). On the other hand, intuition tells us that the collecting interpretations of these should somehow be different, in that the two subexpressions in $[\![1 + 1]\!]$ "get evaluated" in $\mathscr{E}$, but not in $\mathscr{E}'$. How do we capture this operational difference?

To answer this question, note that the *functionals* describing $\mathscr{E}$ and $\mathscr{E}'$ are *not* equivalent. Even though $fix\, G = \mathscr{E} = \mathscr{E}' = fix\, G'$, there is in general no basis for concluding that $G = G'$. In the above case G and G' are different exactly in their behavior at the point $[\![1 + 1]\!]$.[3] This suggests that the correctness of a collecting interpretation should be stated in terms of the functional that *describes* the standard semantics, rather than in terms of the standard semantic function itself.

Higher order and Nonstrict Collecting Interpretations. Consider the following function definition in a language with nonstrict semantics: $[\![f = \lambda x.g(x) + (1 + 1)]\!]$. Several issues arise:

(1) Should the collecting interpretation of the *value* of $[\![f]\!]$ include the fact that $[\![1 + 1]\!]$ is evaluated? For most interpreters the answer is *no*, since the creation of a function is normally thought of as a "closure" whose body is not evaluated until the function is applied. This is a subtle operational concern that distinguishes normal-order evaluation from, say, weak-head-normal-order evaluation or Plotkin's notion of "call-by-value."

(2) The eventual *application* of $[\![f]\!]$ should contribute to the collecting interpretation in whatever context $[\![f]\!]$ is being applied, reflecting the evaluation of the function body.

(3) When $[\![f]\!]$ is applied, the expression to which $[\![x]\!]$ is bound may or may not contribute to the collecting interpretation (since the language has nonstrict semantics), and in this example depends entirely on whether or not $[\![g]\!]$ evaluates its argument.

Consideration of these issues suggests two related solutions: First, a higher order object is needed to capture the proper behavior of a function in a

[3] To see this, note that both G and G' have the form: $\lambda E\, exp\, env. \cdots$. To show they are different, let $E = \perp_{Exp \to Env \to D} = \lambda exp\, env. \perp_D$. Then $G\, E[\![1 + 1]\!]env = \perp_D$, whereas $G'\, E[\![1 + 1]\!]env = 2$.

collecting interpretation. Second, to capture nonstrict semantics a mechanism for delaying contributions of argument evaluation is required.

As a final comment we point out that, although a standard semantics should certainly be expressible as an *abstraction* of any collecting interpretation, the discussions in this section suggest that it may be difficult to justify a particular collecting interpretation as being *standard*. This highlights their operational nature. On the other hand, there is at least one notion of correctness that we think should be captured by any collecting interpretation, based on a notion of *dependence*, and described formally in Section 6.2.

3. STRICT FIRST-ORDER LANGUAGE

In this section and the next two, standard semantic functions such as $\mathscr{E}_{1s}$ ("first order, strict), $\mathscr{E}_{1n}$ ("first order, nonstrict), and $\mathscr{E}_{h}$ ("higher order") are defined. Their counterparts in the collecting interpretations are annotated with a "tilde," as in $\tilde{\mathscr{E}}_{1s}$, $\tilde{\mathscr{E}}_{1n}$, and $\tilde{\mathscr{E}}_{h}$, and domain elements specific to the collecting interpretation are annotated similarly, as in *env*.

We begin the development with a collecting interpretation of expressions for a strict first-order language. The reason for starting with such a restricted language is that it is essentially the same language for which the MFG semantics was developed by Jones and Mycroft.

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

$l \in Lab$ labels,

$k, p \in Con$ first-order constants and primitive functions,

$x \in Bv$ bound variables,

$f \in Fv$ function variables,

$e \in Exp$ expressions, where $e ::= l.k \mid l.x \mid l.p(e_1, \ldots, e_n) \mid l.f(e_1, \ldots, e_n)$,

$pr \in Prog$ programs, where $pr ::= \{e; f_i(x_1, \ldots, x_n) = e_i\}$.

Note that all expressions are labeled; we further assume that every label in a program is unique. A program pr is an expression e to be evaluated in the context of a finite set of mutually recursive first-order equations $f_i(x_1, \ldots, x_n) = e_i$. There are two standard ways of interpreting such programs, depending on whether one wishes a strict or nonstrict semantics, and corresponding roughly to applicative-order and normal-order reduction, respectively, in the lambda calculus. In this section a strict semantics is considered; in the next, nonstrict.

3.1 Standard Strict First-Order Semantics

Assume a domain D whose structure depends on the base types implied by *Con*. For example, if integers and truth values are the only base types, then $D = Int + Bool$. Now define two environment domains, one for bound

variables, the other for function names:

$$bve \in Bve = Bv \to D$$
(bound variable environments)
$$fve \in Fve = Fv \to D* \to D$$
(function variable environments)

and then define the semantic functions $\mathscr{E}_{1s}$ and $\mathscr{P}_{1s}$ by

$\mathscr{E}_{1s}$: $Exp \to Bve \to Fve \to D$ (gives meaning to expressions)
$\mathscr{P}_{1s}$: $prog \to D$ (gives meaning to programs)
$\mathscr{K}_{1s}$: $Con \to D* \to D$ (gives meaning to constants)

$$\mathscr{E}_{1s}[\![l.e]\!]\ bve\ fve = \text{case } [\![e]\!] \text{ of}$$
$$[\![k]\!]: \mathscr{K}_{1s}[\![k]\!]$$
$$[\![x]\!]: bve[\![x]\!]$$
$$[\![p(e_1, \ldots, e_n)]\!]: \mathscr{K}_{1s}[\![p]\!]\ \langle(\mathscr{E}_{1s}[\![e_1]\!]\ bve\ fve), \ldots, (\mathscr{E}_{1s}[\![e_n]\!]\ bve\ fve)\rangle$$
$$[\![f(e_1, \ldots, e_n)]\!]: fve[\![f]\!]\ \langle(\mathscr{E}_{1s}[\![e_1]\!]\ bve\ fve), \ldots, (\mathscr{E}_{1s}[\![e_n]\!]\ bvd\ fve)\rangle$$

$$\mathscr{P}_{1s}[\![\{e;\ f_i(x_1, \ldots, x_n) = e_i\}]\!] = \mathscr{E}_{1s}[\![e]\!]\ \perp\ fve$$
$$\text{whererec } fve = \perp\ [strict_n(\lambda\langle d_1, \ldots, d_n\rangle.\mathscr{E}_{1s}[\![e_i]\!]\ \perp[d_j/x_j]fve)/f_i].$$

For an n-ary function f, "$strict_n f$" is an n-ary function just like f except that it is strict in each of its arguments ($strict_n$ is a generalization of *strict* as used in Stoy [23]). $\mathscr{K}_{1s}$ gives meaning to constants (including primitive functions) and is assumed to be given. Except for the presence of labels in the syntax, this semantics is straightforward and quite conventional.

3.2 Strict First-Order Collecting Interpretation

We now define a collecting interpretation of expressions that is consistent with the standard semantics just defined. First some domain definitions:

$$bve \in Bve = Bv \to D$$
$$fve \in Fve = Fv \to D* \to D$$
$$\widetilde{fve} \in \widetilde{Fve} = Fv \to D* \xrightarrow{t} Cache* \to Cache$$
$$c \in Cache = Lab \to \mathscr{P}(D)$$

To help in the updating of caches, we define the following utility function:

$$upd: Cache \to Lab \to D \xrightarrow{t} Cache$$
$$upd\ c\ lab\ d = c \sqcup \perp[\{d\}/lab]$$

When used with caches, note that $c_1 \sqcup c_2 = \lambda l.\ (c_1\ l) \cup (c_2\ l)$, and thus

$$upd\ c\ lab\ d = \lambda l.\ (c\ l) \cup (\text{if } l = lab \text{ then } \{d\} \text{ else } \{\}).$$

It is easy to show that *upd* is monotonic in its first two arguments, but not its third (where its behavior is similar to that of *setify*). We return to this issue in Section 6.1, where correctness properties are formally addressed.

Now the collecting interpretation itself:

$$\tilde{\mathscr{E}}_{1s}: Exp \to Bve \xrightarrow{t} Fve \xrightarrow{t} \widetilde{Fve} \to Cache$$

$$\tilde{\mathscr{P}}_{1s}: Prog \to Cache$$

$$\tilde{\mathscr{K}}_{1s}: Con \to D^* \xrightarrow{t} Cache^* \to Cache$$

$$\tilde{\mathscr{E}}_{1s}[\![l.e]\!]\, bve\, fve\, \widetilde{fve} =$$

$$\begin{aligned}
&\text{let } d = \mathscr{E}_{1s}[\![l.e]\!]\, bve\, fve \text{ in case } [\![e]\!] \text{ of}\\
&\qquad [\![k]\!]: upd \perp l\, d\\
&\qquad [\![x]\!]: upd \perp l\, d\\
&\qquad [\![p(e_1, \ldots, e_n)]\!]: \text{let } d_i = \mathscr{E}_{1s}[\![e_i]\!]\, bve\, fve,\ i = 1, \ldots, n\\
&\qquad\qquad\qquad\qquad c_i = \tilde{\mathscr{E}}_{1s}[\![e_i]\!]\, bve\, fve\, \widetilde{fve},\ i = 1, \ldots, n\\
&\qquad\qquad\qquad\qquad c = \tilde{\mathscr{K}}_{1s}[\![p]\!]\langle d_1, \ldots, d_n\rangle\langle c_1, \ldots, c_n\rangle\\
&\qquad\qquad\qquad\qquad \text{in } upd\, c\, l\, d\\
&\qquad [\![f(e_1, \ldots, e_n)]\!]: \text{let } d_i = \mathscr{E}_{1s}[\![e_i]\!]\, bve\, fve,\ i = 1, \ldots, n\\
&\qquad\qquad\qquad\qquad c_i = \tilde{\mathscr{E}}_{1s}[\![e_i]\!]\, bve\, fve\, \widetilde{fve},\ i = 1, \ldots, n\\
&\qquad\qquad\qquad\qquad c = \widetilde{fve}[\![f]\!]\langle d_1, \ldots, d_n\rangle\langle c_1, \ldots, c_n\rangle\\
&\qquad\qquad\qquad\qquad \text{in } upd\, c\, l d
\end{aligned}$$

$$\tilde{\mathscr{P}}_{1s}[\![\{e; f_i(x_1, \ldots, x_n) = e_i\}]\!] = \tilde{\mathscr{E}}_{1s}[\![e]\!] \perp fve\, \widetilde{fve}$$

$$\begin{aligned}
&\text{whererec } fve = \perp[(\lambda\langle d_1, \ldots, d_n\rangle\langle c_1, \ldots, c_n\rangle.\\
&\qquad\qquad\qquad \text{let } c = \tilde{\mathscr{E}}_{1s}[\![e_i]\!] \perp [d_j/x_j]\, fve\, \widetilde{fve}\\
&\qquad\qquad\qquad \text{in } c \sqcup c_1 \sqcup \cdots \sqcup c_n)/f_i]\\
&\qquad fve = \perp[strict_n(\lambda\langle d_1, \ldots, d_n\rangle.\ \mathscr{E}_{1_s}[\![e_i]\!]\\
&\qquad\qquad \perp[d_j/x_j]\, fve)/f_i]
\end{aligned}$$

$\tilde{\mathscr{E}}_{1s}[\![l.e]\!]\, bve\, fve\, \widetilde{fve}$ returns a cache containing a "history" of the evaluation of $[\![l.e]\!]$. This history is gathered by adding to the cache the standard value (computed using $\mathscr{E}_{1s}$) of every expression as it is evaluated. More specifically, note that each clause for $\tilde{\mathscr{E}}_{1s}$ returns a value of the form $upd\, c\, l\, d$, where only c varies. In other words, the value d of the expression $[\![l.e]\!]$ is added to the cache c at point l, where c is either the empty cache $\perp_{Cache}$ for a constant or formal parameter, or the cache returned from the application of a primitive or program-defined function. The latter cache in turn represents the history of the computation induced by the function call, which can be seen by studying the construction of the environment $\widetilde{fve}$.

A partial specification of $\tilde{\mathscr{K}}_{1s}$ follows:

$$\tilde{\mathscr{K}}_{1s}[\![if]\!] = \lambda\langle d_p, d_c, d_a\rangle\langle c_p, c_c, c_a\rangle.\ c_p \sqcup (\text{if } d_p \text{ then } c_c \text{ else } c_a)$$

$$\tilde{\mathscr{K}}_{1s}[\![+]\!] = \lambda\langle d_1, d_2\rangle\langle c_1\, c_2\rangle . c_1 \sqcup c_2$$

It is fairly easy to prove that this semantics is consistent with the standard one, and we do so for the higher order case in Section 6. Also important, however, is the correctness and uniqueness of the cache itself. In this regard, note in the specification the separation of the environments *fve* and $\widetilde{fve}$. *fve* in fact does not depend on $\widetilde{fve}$ at all, and $\widetilde{fve}$ depends only on the *exact* value of *fve*. In other words, a construction of $\widetilde{fve}$ by the usual fixpoint iteration technique involves approximations to $\widetilde{fve}$, but not to $\widetilde{fve}$ (and therefore not to *D*); this avoids the problems faced with powerdomains that were discussed earlier. Despite the fact that not all of the domains are continuous, the functionals involved in the recursions are continuous, and a least fixpoint is guaranteed. A formal proof of this (for the higher order case, which subsumes this one) is given in Section 6.

4. NONSTRICT FIRST-ORDER LANGUAGE

We next consider a language whose syntax is identical to that given in the last section, but which is now interpreted using a nonstrict semantics.

4.1 Standard Nonstrict First-Order Semantics

The standard semantics is identical to that given earlier, except that the equation for *fve* is changed to

$$fve = \bot\left[\left(\lambda\langle d_1, \ldots, d_n\rangle.\ \mathscr{E}_{1n}[\![e_i]\!] \perp \left[d_j/x_j\right] fve\right)/f_i\right]$$

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

4.2 Nonstrict First-Order Collecting Interpretation

The key change to the collecting interpretation derives from the observation that the cache resulting from the evaluation of an argument must not be merged with the cache resulting from a function call, unless the corresponding bound variable is actually evaluated. This change is easily made by introducing the domain $\widetilde{Bve}$ which maps bound variables to their respective caches, and then extracting the cache at the time the variable is evaluated.

$$\widetilde{bve} \in \widetilde{Bve} = Bv \to D$$

$$bve \in Bve = Bv \to Cache$$

$$fve \in Fve = Fv \to D^* \to D$$

$$\widetilde{fve} \in \widetilde{Fve} = Fv \to D^* \xrightarrow{t} Cache^* \to Cache$$

$$c \in Cache = Lab \to \mathscr{P}(D)$$

$$\tilde{\mathscr{E}}_{1n}: Exp \to Bve \xrightarrow{t} Fve \xrightarrow{t} \widetilde{Bve} \to \widetilde{Fve} \to Cache$$

$$\tilde{\mathscr{P}}_{1n}: Prog \to Cache$$

$$\tilde{\mathscr{K}}_{1n}: Con \to D^* \xrightarrow{t} Cache^* \to Cache$$

$$\tilde{\mathscr{E}}_{1n}[\![l.e]\!]\, bve\, fve\, \widetilde{bve}\, \widetilde{fve} =$$
$$\text{let } d = \mathscr{E}_{1n}[\![l.e]\!]\, bve\, fve \text{ in case } [\![e]\!] \text{ of}$$
$$[\![k]\!]: upd\, \perp\, l\, d$$
$$[\![x]\!]: upd\, bve[\![x]\!]\, l\, d$$
$$[\![p(e_1, \ldots, e_n)]\!]: \text{let } d_i = \mathscr{E}_{1n}[\![e_i]\!]\, bve\, fve,\ i = 1, \ldots, n \quad \star$$
$$c_i = \tilde{\mathscr{E}}_{1n}[\![e_i]\!]\, bve\, fve\, \widetilde{bve}\, \widetilde{fve},\ i = 1, \ldots, n$$
$$c = \tilde{\mathscr{K}}_{1n}[\![p]\!]\langle d_1, \ldots, d_n\rangle\langle c_1, \ldots, c_n\rangle$$
$$\text{in } upd\, c\, l\, d$$
$$[\![f(e_1, \ldots, e_n)]\!]: \text{let } d_i = \mathscr{E}_{1n}[\![e_i]\!]\, bve\, fve,\ i = 1, \ldots, n$$
$$c_i = \tilde{\mathscr{E}}_{1n}[\![e_i]\!]\, bve\, fve\, \widetilde{bve}\, \widetilde{fve},\ i = 1, \ldots, n$$
$$c = \widetilde{fve}[\![f]\!]\langle d_1, \ldots, d_n\rangle\langle c_1, \ldots, c_n\rangle$$
$$\text{in } upd\, c\, l\, d$$
$$\tilde{\mathscr{P}}_{1n}[\![\{e; f_i(x_1, \ldots, x_n) = e_i\}]\!] = \tilde{\mathscr{E}}_{1n}[\![e]\!] \perp fve \perp \widetilde{fve}$$
$$\text{whererec } \widetilde{fve} = \perp\big[(\lambda\langle d_1, \ldots, d_n\rangle\langle c_1, \ldots, e_n\rangle. \quad \star$$
$$\tilde{\mathscr{E}}_{1n}[\![e_i]\!] \perp \big[d_j/x_j\big]\, fve \perp\big[c_j/x_j\big]\, \widetilde{fve}\big)/f_i\big]$$
$$fve = \perp\big[(\lambda\langle d_1, \ldots, d_n\rangle.\ \mathscr{E}_{1n}[\![e_i]\!] \perp \big[d_j/x_j\big]\, fve)/f_i\big].$$

Other than minor subscript changes of "$1a$" to "$1n$" and the addition of the extra argument to $\tilde{\mathscr{E}}_{1n}$, 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. The definition of $\tilde{\mathscr{K}}_{1n}$ is a minor variation of $\tilde{\mathscr{K}}_{1s}$,and is left to the reader.

5. NONSTRICT HIGHER ORDER LANGUAGE

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

$l \in Lab$	labels,
$k \in Con$	constants,
$x, f \in Id$	identifiers, either bound variables or function names,
$e \in Exp$	expressions, where $e ::= l.k \mid l.x \mid l.f \mid l.(\lambda x \cdot e) \mid l.(e_1\ e_2)$,
$pr \in Prog$	programs, where $pr ::= \{e; f_i = e_i\}$,

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

5.1 Standard Nonstrict Higher Order Semantics

The structure of the domain D is again assumed to depend on Con, but now will typically be the solution of a reflexive domain equation such as $D = Int + Bool + (D \to D)$.

$$+ \mathit{Bool} + (D \to D).$$

$$\begin{aligned}
&\mathit{env} \in \mathit{Env} = \mathit{Id} \to D\\
&\mathscr{E}_{\mathrm{h}}\colon\ \mathit{Exp} \to \mathit{Env} \to D\\
&\mathscr{P}_{\mathrm{h}}\colon\ \mathit{Prog} \to D\\
&\mathscr{K}_{\mathrm{h}}\colon\ \mathit{Con} \to D\\
&\mathscr{E}_{\mathrm{h}}[\![l.e]\!]\mathit{env} = \text{case } [\![e]\!] \text{ of}\\
&\qquad [\![k]\!]\colon\ \mathscr{K}_{\mathrm{h}}[\![k]\!]\\
&\qquad [\![x]\!]\colon\ \mathit{env}[\![x]\!]\\
&\qquad [\![\lambda x.e]\!]\colon\ \lambda d.\mathscr{E}_{\mathrm{h}}[\![e]\!]\mathit{env}[d/x]\\
&\qquad [\![e_1\ e_2]\!]\colon\ (\mathscr{E}_{\mathrm{h}}[\![e_1]\!]\mathit{env})(\mathscr{E}_{\mathrm{h}}[\![e_2]\!]\mathit{env})\\
&\mathscr{P}_{\mathrm{h}}[\![\{e;\ f_i = e_i\}]\!] = \mathscr{E}_{\mathrm{h}}[\![e]\!]\mathit{env}\\
&\quad \text{whererec } \mathit{env} = \bot[\mathscr{E}_{\mathrm{h}}[\![e_i]\!]\mathit{env}/f_i].
\end{aligned}$$

As is the first-order semantics, this semantics is quite conventional, except for the use of labels.

5.2 Nonstrict Higher Order Collecting Interpretation

The introduction of higher order functions complicates the collecting interpretation somewhat, since the application of an expression might induce other values to be added to the cache. We solve this problem in a way similar to solutions for other higher order analyses [7, 10]—a higher order function is added to the domain of answers which captures the behavior of an expression when it is applied to an argument. The desired domain is not simply $\mathit{Ans} = \mathit{Cache} \times (\mathit{Ans} \to \mathit{Ans})$, however, because in addition to the behavior of the argument under the collecting interpretation, we need to know the value of the argument in the standard semantics. Thus the desired domain is given by

$$\mathit{Ans} = \mathit{Cache} \times \left(D \xrightarrow{t} \mathit{Ans} \to \mathit{Ans}\right)$$

and the collecting environment must map identifiers to Ans.

$$\begin{aligned}
&\mathit{env} \in \mathit{Env} = \mathit{Id} \to D\\
&\widetilde{\mathit{env}} \in \widetilde{\mathit{Env}} = \mathit{Id} \to \mathit{Ans}\\
&\qquad a \in \mathit{Ans} = \mathit{Cache} \times \left(D \xrightarrow{t} \mathit{Ans} \to \mathit{Ans}\right)\\
&\qquad c \in \mathit{Cache} = \mathit{Lab} \to \mathscr{P}(D)\\
&\bar{\mathscr{E}}_{\mathrm{h}}\colon \mathit{Exp} \to \mathit{Env} \xrightarrow{t} \widetilde{\mathit{Env}} \to \mathit{Ans}\\
&\tilde{\mathscr{P}}_{\mathrm{h}}\colon \mathit{Prog} \to \mathit{Ans}\\
&\tilde{\mathscr{K}}_{\mathrm{h}}\colon \mathit{Con} \to D \xrightarrow{t} \mathit{Ans} \to \mathit{Ans}\\
&\tilde{\mathscr{E}}_{\mathrm{h}}[\![l.e]\!]\,\mathit{env}\,\widetilde{\mathit{env}} =\\
&\text{let } d = \mathscr{E}_{\mathrm{h}}[\![l.e]\!]\mathit{env} \text{ in case } [\![e]\!] \text{ of}\\
&\qquad [\![k]\!]\colon \langle \mathit{upd}\ \bot\ l\,d,\ \tilde{\mathscr{K}}_{\mathrm{h}}[\![k]\!]\rangle\\
&\qquad [\![x]\!]\colon \text{let } \langle c, f\rangle = \widetilde{\mathit{env}}[\![x]\!]\\
&\qquad\qquad \text{in } \langle \mathit{upd}\,c\,l\,d,\ f\rangle
\end{aligned}$$

$$
\begin{aligned}
[\![\lambda x.e]\!]&: \text{let } f = \lambda d\, a.\ \tilde{\mathscr{E}}_{\mathrm{h}}[\![e]\!] env[d/x]\,\widetilde{env}[a/x] \\
&\qquad \text{in } \langle upd\ \bot\ l\, d, f\rangle \\
[\![e_1\ e_2]\!]&: \text{let } \langle c, f\rangle = \tilde{\mathscr{E}}_{\mathrm{h}}[\![e_1]\!] env\ \widetilde{env} \\
&\qquad\qquad \langle c', f'\rangle = f(\mathscr{E}_{\mathrm{h}}[\![e_2]\!] env)(\tilde{\mathscr{E}}_{\mathrm{h}}[\![e_2]\!] env\ \widetilde{env}) \\
&\qquad \text{in } \langle upd(c \sqcup c')\, l\, d, f'\rangle \\
\tilde{\mathscr{P}}_{\mathrm{h}}[\![\{e; f_i = e_i\}]\!] &= \tilde{\mathscr{E}}_{\mathrm{h}}[\![e]\!] env\ \widetilde{env} \\
&\text{whererec } env = \bot[(\tilde{\mathscr{E}}_{\mathrm{h}}[\![e_i]\!] env\ \widetilde{env})/f_i] \\
&\qquad\qquad env = \bot[(\mathscr{E}_{\mathrm{h}}[\![e_i]\!] env)/f_i]
\end{aligned}
$$

Given previous discussions, these equations should be self-explanatory. The only subtlety is in the treatment of higher order functions, where the equations for $[\![\lambda x.e]\!]$ and $[\![e_1\ e_2]\!]$ should be considered together. Note that (1) An object of functionality $(D \to Ans \to Ans)$ is constructed to delay the evaluation of a function body, and thus nothing gets evaluated in $[\![\lambda x \cdot e]\!]$ except for the functional value itself. (2) In an application $[\![e_1\ e_2]\!]$ the evaluation of $[\![e_1]\!]$ contributes to the cache, as does the result of "evaluating the body" of the function corresponding to $[\![e_1]\!]$, which for lambda abstractions is generated from the functional object created in (1).

We again provide a partial specification of $\tilde{\mathscr{K}}_{\mathrm{h}}$:

$$
\begin{aligned}
&\tilde{\mathscr{K}}_{\mathrm{h}}[\![if]\!] = \lambda d_p\langle c_p, f_p\rangle.\ \text{if } d_p \text{ then } \langle c_p, \lambda d_c\langle c_c, f_c\rangle.\langle c_c, \lambda d_a\langle c_a, f_a\rangle.\ \langle \bot\ , f_c\rangle\rangle\rangle \\
&\text{else } \langle c_p, \lambda d_c\langle c_c, f_c\rangle.\langle \bot\ , \lambda d_a\langle c_a, f_a\rangle.\langle c_a, f_a\rangle\rangle\rangle \\
&\tilde{\mathscr{K}}_{\mathrm{h}}[\![+]\!] = \lambda d_1\langle c_1, f_1\rangle.\langle c_1, \lambda d_2\langle c_2, f_2\rangle.\ \langle c_2, err\rangle\rangle \\
&\text{where } err = \lambda d\, a.\ \langle \bot\ , err\rangle
\end{aligned}
$$

6. CORRECTNESS

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

6.1 Soundness and Uniqueness

First we prove the existence and uniqueness of the collecting interpretation. To do so it is helpful to rewrite the specification of $\tilde{\mathscr{E}}_{\mathrm{h}}$ as follows:

$$
\begin{aligned}
&\tilde{\mathscr{E}}_{\mathrm{h}} = fix\ \tilde{G} \\
&\tilde{G} = \lambda\tilde{\mathscr{E}}_{\mathrm{h}}. \\
&\quad \lambda[\![l.e]\!] env\ \widetilde{env}. \\
&\qquad \text{let } d = \mathscr{E}_{\mathrm{h}}[\![l.e]\!] env \text{ in } H\ \tilde{\mathscr{E}}_{\mathrm{h}} \\
&\qquad \text{where } H\ E = \text{case } [\![e]\!] \text{ of} \\
&\qquad\qquad [\![k]\!]: \langle upd_{ld}\ \bot\ ,\ \tilde{\mathscr{K}}_{\mathrm{h}}[\![k]\!]\rangle \\
&\qquad\qquad [\mathrm{x}]: \text{let } \langle c, f\rangle = \widetilde{env}[\![x]\!] \\
&\qquad\qquad\qquad \text{in } \langle upd_{ld} c, f\rangle \\
&\qquad\qquad [\![\lambda \mathrm{x.e}]\!]: \text{let } f = \lambda d\, a.E[\![e]\!]\ env[d/x]\widetilde{env}[a/x] \\
&\qquad\qquad\qquad \text{in } \langle upd_{ld}\ \bot\ ,\ f\rangle \\
&\qquad\qquad [\![e_1\ e_2]\!]: \text{let } \langle c, f\rangle = E[\![e_1]\!]\ env\ \widetilde{env} \\
&\qquad\qquad\qquad\qquad \langle c', f'\rangle = f(\mathscr{E}_{\mathrm{h}}[\![e_2]\!]\ env)(E[\![e_2]\!]\ env\ \widetilde{env}) \\
&\qquad\qquad\qquad \text{in } \langle upd_{ld}(c \sqcup c'), f'\rangle
\end{aligned}
$$

LEMMA 1. *For fixed d and l, the function* $upd_{ld} = \lambda c.\ upd\, c\, l\, d$ *is monotonic and continuous.*

PROOF. (a) upd_{ld} is monotonic if $c \sqsubseteq c' \Rightarrow upd\, c\, ld \sqsubseteq upd\, c'\, l\, d$. Expanding *upd*, this amounts to $c \sqsubseteq c' \Rightarrow c \sqcup \bot[\{d\}/l] \sqsubseteq c' \sqcup \bot[\{d\}/l]$, which is true by the definition of $\sqcup$. (b) upd_{ld} is continuous if $\mathrm{upd_{ld}}(\sqcup\{\mathrm{d_i}\}) = \sqcup\{\mathrm{upd_{ld}\ d_i}\}$; that is, $(\sqcup\{\mathrm{d_i}\}) \sqcup \bot[\{\mathrm{d}\}/\mathrm{l}] = \sqcup\{\mathrm{d_i} \sqcup \bot[\{\mathrm{d}\}/\mathrm{l}]\}$. But this is true by commutativity and associativity of $\sqcup$. □

LEMMA 2. *For fixed env,* $\widetilde{env}$, *and* $[\![l \cdot e]\!]$, *the function H (defined above) is monotonic and continuous.*

PROOF. This can be shown via a straightforward analysis as was done with upd_{ld}, but an easier approach is to note that H is constructed solely from monotonic and continuous operators (using Theorem 3 and Lemma 1 for upd_{ld}). Then by Theorem 2 we know immediately that the resulting function must also be monotonic and continuous. □

LEMMA 3. *The function* $\tilde{G}$ *(defined above) is monotonic and continuous.*

PROOF. (1) $\tilde{G}$ is monotonic if $e \sqsubseteq e' \Rightarrow \tilde{G}e \sqsubseteq \tilde{G}e'$, or, expanding $\tilde{G}$, $(\lambda[\![l.e]\!]\, env\, \widetilde{env}.$ let $d = \mathscr{E}_{\mathrm{h}}[\![l.e]\!]env$ in $He) \sqsubseteq (\lambda[\![l.e]\!]\, env\, \widetilde{env}.$ let $d = \mathscr{E}_{\mathrm{h}}[\![l.e]\!]env$ in $H'e')$. This amounts to showing that $He \sqsubseteq He'$, which is true since by Lemma 2 H is monotonic. (2) $\tilde{G}$ is continuous if $\tilde{G}(\sqcup\{e_i\}) = \sqcup\{\tilde{G}e_i\}$. Expanding $\tilde{G}$ on the left-hand side and noting from Lemma 2 that H is continuous yields: $\lambda[\![l.e]\!]\, env\, \widetilde{env}.$ let $d = \mathscr{E}_{\mathrm{h}}[\![l.e]\!]env$ in $\sqcup\{He_i\}$. But this is the same as $\sqcup\{\lambda[\![l.e]\!]\, env\, \widetilde{env}.$ let $d = \mathscr{E}_{\mathrm{h}}[\![l.e]\!]env$ in $He_i\}$, which is just $\sqcup\{\tilde{G}e_i\}$, and the equality has been proven. □

THEOREM 4. $\tilde{\mathscr{E}}_{\mathrm{h}}$ *exists and is unique.*

PROOF. $\tilde{\mathscr{E}}_{\mathrm{h}}$ is defined as $fix\, \tilde{G}$. Since by Lemma 6.3 $\tilde{G}$ is continuous, then by Theorem 1.1 we know that $\tilde{\mathscr{E}}_{\mathrm{h}}$ is well defined. □

THEOREM 5. $\tilde{\mathscr{P}}_{\mathrm{h}}$ *exists and is unique.*

PROOF. By an argument similar to that used in the proof of Lemma 2, it is easy to show that the function $J = \tilde{\mathscr{E}}_{\mathrm{h}}[\![e_t]\!]env$ is continuous. Thus by Theorem 1, $\widetilde{env} = fix\, \lambda\widetilde{env}.\ \bot[(J\, \widetilde{env})/f_t]$ is well defined, and thus so is $\tilde{\mathscr{P}}_{\mathrm{h}}$. □

6.2 Operational Correctness

Now the question returns to what can be said about the values "contained in the cache." This turns out to be a rather subtle problem, for the reason mentioned in Section 2.2: there are several ways to define exactly what it means for a program to be "evaluated." In this section we show that one *can* state a result based on the intuition that if a program terminates with a nonfunctional value, then the cache should contain any values that the result "depends" on. But first the notion of dependence must be formally defined.

Let G be the functional describing $\mathscr{E}_{\mathrm{h}}$; that is, $\mathscr{E}_{\mathrm{h}} = fix\, G$ (the precise definition of G is easily derived from the equations defining $\mathscr{E}_{\mathrm{h}}$ given earlier,

in the same way that we derived $\tilde{G}$ from $\tilde{\mathscr{E}}_h$). Then define

$$\mathscr{E}_h\_maker[\![e]\!]env = fix\big(\lambda E[\![e']\!]env'.\ \text{if } \big([\![e']\!] = [\![e]\!]\big) \wedge \big(env' = env\big) \text{ then } \perp \text{ else } GE[\![e']\!]env'\big).$$

Note that the functional argument to *fix* is "almost" equal to G—the function $\mathscr{E}'_h = \mathscr{E}_h\_maker[\![e]\!]env$ is identical to $\mathscr{E}_h$ except that if it ever encounters the argument pair $\langle [\![e]\!], env\rangle$, it returns $\perp$ (and of course if $\mathscr{E}_h[\![e]\!]env = \perp$, then $\mathscr{E}_h = \mathscr{E}'_h$).[4]

Similary define

$$\mathscr{P}_h\_maker[\![e]\!]env = \lambda[\![\{e_0; f_i = e_i\}]\!].\ \mathscr{E}'_h[\![e_0]\!]env'$$
$$\text{whererec } env' = \perp\,[\mathscr{E}'_h[\![e_i]\!]env'/f_i]$$
$$\mathscr{E}'_h = \mathscr{E}_h\_maker[\![e]\!]env.$$

(Compare this to $\mathscr{P}_h$, defined in Section 5.1.) $\mathscr{P}'_h = \mathscr{P}_h\_maker[\![e]\!]env$ relates to $\mathscr{P}_h$ in the same way that $\mathscr{E}'_h$ relates to $\mathscr{E}_h$.

The important property to note about $\mathscr{E}'_h$ and $\mathscr{P}'_h$ is that the bottom value returned at point $\langle [\![e]\!], env\rangle$ may propagate, thus changing the value of expressions that might "depend on" the result of evaluating $[\![e]\!]$ in environment *env*. This leads to

Definition. An expression $[\![e']\!]$ in environment env' is said to *depend on* $[\![e]\!]$ in environment *env* if and only if $\mathscr{E}'_h[\![e']\!]env' \neq \mathscr{E}_h[\![e']\!]env'$, where $\mathscr{E}'_h = \mathscr{E}_h\_maker[\![e]\!]env$.

Definition. A program $[\![pr]\!]$ is said to *depend on* $[\![e]\!]$ in environment *env* if and only if $\mathscr{P}'_h[\![pr]\!] \neq \mathscr{P}_h[\![pr]\!]$, where $\mathscr{P}'_h = \mathscr{P}_h\_maker[\![e]\!]env$.

In other words, if we can change the behavior of a program $[\![pr]\!]$ by causing $[\![e]\!]$ in environment *env* to diverge, then it must be the case that $[\![pr]\!]$ depends on that evaluation.[5] Note that if $[\![pr]\!]$ diverges, it depends on nothing. Also note the subtlety of the definition when a program evaluates to a functional value—for example, the program $[\![pr]\!] = [\![\{\lambda x \cdot x\}]\!]$ depends on $[\![x]\!]$ in *all* environments $env[d/x]$ in which d is proper (i.e., nonbottom). This is because $[\![pr]\!]$ evaluates to the function $\lambda d.\ \mathscr{E}_h[\![x]\!]\perp[d/x]$. However, as discussed in Section 2.2, our collecting interpretations capture an operational behavior in which a function is a "closure," and thus none of its domain/codomain pairs are computed in evaluating the function itself. Thus we can only state a correctness result with respect to dependence for programs yielding nonfunctional values:

THEOREM 6. *Given any program* $[\![pr]\!] \in Prog$, *let* $\langle c, f\rangle = \tilde{\mathscr{P}}_h[\![pr]\!]$, $d = \mathscr{P}_h[\![pr]\!]$, *and assume* $d \notin (D \to D)$. *If* $[\![pr]\!]$ *depends on* $[\![l \cdot e]\!]$ *in environment env, then* $(\mathscr{E}_h[\![l \cdot e]\!]env) \in c(l)$.

[4]Also noted that the fixpoint is well defined: the argument to *fix* is continuous, even if noncontinuous equality (=) is assumed.

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

PROOF. The proof first depends on a correct definition of $\tilde{\mathscr{K}}_h$. Specifically, let $[\![l.e']\!] = [\![c\, e_1, \ldots, e_n]\!]$, $c \in Con$, $n \geq 1$, $\langle c, f\rangle = \tilde{\mathscr{E}}_h[\![e']\!]env'\,\widetilde{env}$, and assume $\mathscr{E}_h[\![e']\!]env' \notin (D \to D)$. Then if $[\![e']\!]$ in environment env' depends on $[\![e]\!]$ in environment env, it must be the case that $(\mathscr{E}_h[\![e]\!]env) \in c(l)$. We assume that this condition holds.

The remainder of the proof is a structural induction on expressions: it is sufficient to show that if an expression that evaluates to a nonfunctional value depends on one of its *immediate subexpressions*, then the value of that subexpression must appear in the collecting interpretation of the result. From the equations defining $\tilde{\mathscr{E}}_h$, we see that the only nonfunctional expression with substructure is an application $[\![e_1\, e_2]\!]$, for which we identify two subcases: (1) the subexpression on which the overall expression depends is $[\![e_1]\!]$, in which case the result is immediate, since the entire cache for $[\![e_1]\!]$ is added to the result, or (2) the subexpression is $[\![e_2]\!]$, which reduces to showing that $[\![e_1]\!]$ is strict. A straightforward induction then leads to the overall proof. □

It is somewhat dissatisfying that a stronger result (i.e., one involving functional values) cannot be stated. On the other hand, it is difficult to imagine such a stronger result based on the standard semantics, since our interpretation of a function as a closure reflects an operational semantics. Indeed, this difference is analogous to the difference between "weak-head-normal" reduction and "normal-order" reduction in the lambda calculus.

7. ABSTRACT COLLECTING INTERPRETATIONS OF EXPRESSIONS

Collecting interpretations are more than an intellectual exercise or theoretical curiosity. Our motivation stems, in fact, from our work on compilers for functional languages. We present in this section an important application of our work to *strictness analysis*. Recall that a function f is said to be *strict* if $f \perp = \perp$, that is, if the function applied to an argument does not terminate unless the argument itself terminates. The normal problem solved by a *strictness analysis* is to compute a *safe approximation* to function strictness, since the problem itself is clearly undecideable (being a paraphrase of the halting problem). The analysis is "safe" in the sense that if it declares a function to be strict, then indeed it is, but the analysis will not be able to determine every strict function. Based on the result of a strictness analysis, a compiler may convert a relatively expensive "call-by-name" evaluation strategy to a more efficient "call-by-value" evaluation. This has proven to be a valuable optimization in implementations of nonstrict languages.

Now 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 [2, 10] tells us strictness properties of *map*, but *only as a function of its argument's strictness properties*. Thus despite strictness analysis, the compiler-writer is not free to optimize the application of f 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, despite the fact that empirical studies [6] 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 a major reason why. Fortunately, the techniques presented in this paper allow one to infer the necessary properties to invoke either of the optimizations mentioned above, and in fact we have implemented such optimizations in a compiler for a nonstrict functional language [25].

To see how easily the ideas apply, let us first define a higher order strictness analysis (taken from [10]). We begin with the domain definitions:

$$\begin{aligned} V &\quad \text{(variables of interest)},\\ Sv = \mathcal{P}(V) &\quad \text{(set of variables)},\\ Sp = Sv \times (Sp \to Sp) &\quad \text{(strictness pairs)},\\ Senv = V \to Sp &\quad \text{(strictness environments)}. \end{aligned}$$

Here, the strictness interpretation of an expression is a *strictness pair*, which contains a set of variables which were evaluated during the execution of the expression and a function. The function is used to determine the set of variables evaluated when this expression is applied to another expression.

Following [10], the following subscript notation is adopted for strictness pairs: $\langle sv, sf\rangle_v = sv$ and $\langle sv, sf\rangle_f = sf$. A special error element is also defined: $serr = \lambda\hat{x}.\ \langle\{\}, serr\rangle$.

$$\begin{aligned} &K: Con \to Sp && \text{(maps constants to } Sp\text{, and assumed given)},\\ &S: Exp \to Senv \to Sp && \text{(maps expressions to strictness pairs)},\\ &P_s: Prog \to Sp && \text{(gives meaning to programs)}. \end{aligned}$$

$$\begin{aligned} S[\![k]\!]senv &= K[\![k]\!]\\ S[\![x]\!]senv &= senv[\![x]\!]\\ S[\![\lambda x.e]\!]senv &= \langle\{\}, \lambda\hat{x}.\ S[\![e]\!]senv[\hat{x}/x]\rangle\\ S[\![e_1\ e_2]\!]senv &= \langle (S[\![e_1]\!]senv)_v \cup sv, sf\rangle\\ &\qquad \text{where } \langle sv, sf\rangle = (S[\![e_1]\!]senv)_f(S[\![e_2]\!]senv)\\ P_s[\![\{e; f_i = e_i\}]\!] &= S[\![e]\!]senv' \text{ whererec } senv'\\ &= \bot\,[(S[\![e_i]\!]senv')/f_i]. \end{aligned}$$

The strictness analysis given in [10] actually returns an *environment* that contains the strictness properties of the top-level functions, whereas the analysis given above always returns the *empty set*. However, this will not matter, since the collecting interpretation will uncover the desired information, and more. To form this collecting interpretation, simply take the higher order collecting interpretation defined in Section 5.2, and substitute Sp for D, S for $\mathscr{E}_h$, and *senv* for *env*, yielding

$$senv \in Senv = Id \to Sp$$
$$\widetilde{senv} \in \widetilde{Senv} = Id \to Ans$$
$$a \in Ans = Cache \times \left(Sp \xrightarrow{t} Ans \to Ans\right)$$
$$c \in Cache = Lab \to \mathscr{P}(Sp)$$
$$\tilde{\mathscr{E}}_h : Exp \to Senv \xrightarrow{t} Senv \to Ans$$
$$\mathscr{P}_h : Prog \to Ans$$
$$\tilde{\mathscr{K}}_h : Con \to Sp \xrightarrow{t} Ans \to Ans$$

$$\tilde{\mathscr{E}}_h[\![l.e]\!]\, senv\, \widetilde{senv} =$$
$$\text{let } d = S[\![l.e]\!]\, senv \text{ in case } [\![e]\!] \text{ of}$$
$$[\![k]\!] : \langle upd \perp l\, d,\ \tilde{\mathscr{K}}_h[\![k]\!]\rangle$$
$$[\![x]\!] : \text{let } \langle c, f\rangle = \widetilde{senv}[\![x]\!]$$
$$\text{in } \langle upd\, c\, l\, d,\ f\rangle$$
$$[\![\lambda x.e]\!] : \text{let } f = \lambda d\, a.\ \tilde{\mathscr{E}}_h[\![e]\!]\, senv[d/x]\, \widetilde{senv}[a/x]$$
$$\text{in } \langle upd \perp l\, d,\ f\rangle$$
$$[\![e_1\ e_2]\!] : \text{let } \langle c, f\rangle = \tilde{\mathscr{E}}_h[\![e_1]\!]\, senv\, \widetilde{senv}$$
$$\langle c', f'\rangle = f(S[\![e_2]\!]\, senv)\left(\tilde{\mathscr{E}}_h[\![e_2]\!]\, senv\, \widetilde{senv}\right)$$
$$\text{in } \langle upd(c \sqcup c')\, l\, d,\ f'\rangle$$
$$\tilde{\mathscr{P}}_h[\![\{e;\ f_i = e_i\}]\!] = \tilde{\mathscr{E}}_h[\![e]\!]\, senv\, \widetilde{senv}$$
$$\text{whererec } \widetilde{senv} = \perp \left[\left(\tilde{\mathscr{E}}_h[\![e_i]\!]\, senv\, \widetilde{senv}\right)/f_i\right]$$
$$senv = \perp\left[\left(S[\![e_i]\!]\, senv\right)/f_i\right].$$

Of course, a suitable redefinition of $\mathscr{K}_h$ is required as well (to reflect the abstractions performed by K), the details of which are left to the reader.

8. DISCUSSION

It is possible to 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 intepretations given, and similar to saving all argument tuples in an MFG semantics. 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 $S_1 \times S_2$; that is, 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" and "relational" methods of flow analysis discussed in Jones and Muchick [13] and later in Mycroft [18].

To make this extension we can collect, in addition to the *value* of an expression, the *environment* in effect when the expression is evaluated. That is, the functionality of the cache is changed to $Lab \rightarrow \mathcal{P}(D \times Env)$. Thus the answer to the previous question would be

$$S = \{\langle d_1, d_2\rangle \mid \langle d_1, env_1\rangle \in S_1, \langle d_2, env_2\rangle \in S_2, \text{ and } env_1 = env_2\}.$$

The necessary changes to accomplish this for each of the collecting interpretations given earlier are straightforward.

It is perhaps also useful to distinguish between call-by-name evaluation (as we have been using) and true "lazy evaluation" in which the expressions bound to variables during beta reduction are evaluated at most once—the value is somehow "saved" for later use. From the perspective of the standard semantics this difference is not important, since the "answer" is the same regardless. However, most modern implementations of functional languages employ lazy evaluation, and thus modeling its behavior precisely may be important to suitable abstractions of that behavior. The primary difficulty in building such an analysis is the need for a mechanism to save the values of actual parameters as they are computed. Although we have not done this, we believe it is not difficult, and the details are left to the reader.

ACKNOWLEDGMENTS

The authors wish to thank Neil Jones for extensive comments on an earlier draft of this manuscript, Phil Wadler for suggesting an improvement in notation, and to the several referees whose comments helped improve the overall presentation.

REFERENCES

1. Abramsky, S. and Hankin C. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, Chichester, England, 1987.
2. Burn, G. L, Hankin, C. L. and Abramsky, S. The theory of strictness analysis for higher order functions. In *Lecture Notes in Computer Science*, vol. 217: *Programs as Data Objects.* Springer-Verlag, New York, 1985, pp. 42–62.
3. Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages.* ACM, New York, 1977, pp. 238–252.
4. Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages.* ACM, New York, pp. 269–282.
5. Donzeau-Gouge, V. Denotational definition of properties of program computations. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, N.J., pp. 343–379.

6. FAIRBAIRN, J. AND WRAY, S. C. Code generation techniques for functional languages. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, (Cambridge, Mass., Aug. 1986). ACM, New York, 1986, pp. 94–104.
7. GOLDBERG, B., AND HUDAK, P. Detecting sharing of partial applications in functional languages. YALEU/DCS/RR-526, Dept. Computer Science, Yale University, New Haven, Conn., Mar. 1987.
8. HUDAK, P. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings 1986 ACM Conference on LISP and Functional Programming*, ACM, New York, Aug. 1986, pp. 351–363.
9. HUDAK, P. Collecting interpretations of expressions. Res. Rep. YALEU/DCS/RR-497, Dep. Computer Science, Yale University, New Haven, Conn., 1986.
10. HUDAK, P., AND YOUNG, J. Higher order strictness analysis for untyped lambda calculus. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (Jan. 1986), pp. 97–109.
11. HUGHES, J. Strictness detection in non-flat domains. In *Lecture Notes in Computer-Science, vol. 217: Programs as Data Objects*. Springer-Verlag, New York, 1986, pp. 42–62.
12. JONES, N. D. Private communications, June 1987.
13. JONES, N. D., AND MUCHNICK, S. S. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, N. J , 1981, pp. 380–393.
14. JONES, N. D., AND MYCROFT, A. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th Symposium on Principles of Programming Languages*. ACM, New York, 1986, pp. 296–306
15 JONES, N., AND SONDERGAARD, H. A semantics-based framework for the abstract interpretation of 'Prolog.' In *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, England, 1987, pp. 123–142.
16. LINDSTROM, G. Statistic evaluation of functional programs. In *SIGPLAN '86 Symposium on Compiler Construction*. ACM, New York, 1986, pp. 196–206. Published as *SIGPLAN Notices 21*, 7 (July 1986).
17. MELLISH, C. Abstract interpretation of PROLOG programs. In *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, England, 1987, pp. 181–198.
18. MYCROFT, A. Abstract interpretation and optimizing transformations for applicative programs. Ph.D. dissertation, University of Edinburgh, Edinburgh, Scotland, 1981
19. MYCROFT, A. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of International Symposium on Programming, Lecture Notes in Computer Science*, vol. 83, Springer-Verlag, New York, 1980, pp. 269–281.
20. NIELSON, F. Abstract Interpretation Using Domain Theory. Ph.D. dissertation, University of Edinburgh, Edinburgh, Scotland, Oct. 1984.
21. NIELSON, F. A denotational framework for data flow analysis. *Acta Inform. 18* (1982), 265–287.
22. SCHMIDT, D. A. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Boston, Mass., 1986.
23. STOY, J. E *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. M.I.T. Press, Cambridge, Mass , 1977.
24. WADLER, P. AND HUGHES, R. J. M. Projections for strictness analysis. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference, Lecture Notes in Computer Science*, vol. 274. Springer-Verlag, New York, 1987, pp. 385–407.
25. YOUNG, J. The semantic analysis of functional programs: Theory and practice. Ph D. dissertation, Dep. Computer Science, Yale University, New Haven, Conn., 1988.

Received November 1987; revised May 1990; accepted June 1990