A Unified Treatment of Flow Analysis in Higher-Order Languages

Suresh Jagannathan* Stephen Weeks†

Abstract

We describe a framework for flow analysis in higher-order languages. It is both a synthesis and extension of earlier work in this area, most notably [20, 22].

The framework makes explicit use of *flow graphs* for modeling control and data flow properties of untyped higher-order programs. The framework is parameterized, and can express a hierarchy of analyses with different cost/accuracy tradeoffs. The framework is also amenable to a direct, efficient implementation.

We develop several instantiations of the framework, and prove their running-time complexity. In addition, we use the simplest instantiation to demonstrate the equivalence of a 0CFA style analysis[20] and the set-based analysis of [8].

1 Introduction

The flow analysis problem for higher-order programming languages such as Scheme[4] or ML[13] is concerned with tracking data and control flow in the presence of first-class (anonymous) procedures, rich data abstractions (*e.g.,* lists, records, tuples, etc), and references. In the context of these languages, an effective control-flow analyzer can enable a number of important optimizations such as lifetime and escape analysis[15], type recovery[19], safety analysis[14] efficient closure analysis[17], constant folding, and code hoisting.

There is a spectrum of control-flow analyses that can be implemented for these languages. For example, an exact control-flow analyzer is simply an interpreter that preserves all information generated during the evaluation of a program. It has high accuracy but also a prohibitively high cost since its running time is proportional to the running time of the input program. In contrast, a very inexact interprocedural analysis might combine all results of a function applied from all its potential call sites. It has low accuracy since it does not disambiguate among the set of functions callable from a given call site or the set of values applied to a given function. However, it also has relatively low running time complexity.

*Computer Science Division, NEC Research Institute, Princeton, NJ. `suresh@research.nj.nec.com`.

†Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA. `sweeks@cs.cmu.edu`.

To appear in the Proceedings of the 22^{nd} Annual ACM Symposium on Principles of Programming Languages, 1995.

In this paper, we present a framework based on an abstract interpretation[6, 5] of an operational semantics. Section 2 first defines an untyped, higher-order, call-by-value language extended with recursion, conditionals, and references. Section 3 defines an exact semantics that formalizes the notion of *flow graph.* The semantics is similar to a *collecting interpretation*[9] insofar as a flow graph records the history of a computation; however, a flow graph also contains additional dependence information corresponding to the movement of data values.

Section 4 presents a parameterized abstract semantics that is capable of expressing a range of analyses with different cost/accuracy tradeoffs. The abstract semantics manipulates flow graphs which approximate those constructed by the exact semantics. An approximation is based on a finite partition of the nodes in the exact flow graph. Section 5 considers the coarsest possible partition and relates it to set-based analysis[8], an analysis that is also capable of capturing control-flow information.

Section 6 describes a direct implementation of the abstract semantics. We argue that the representation of the abstract state in terms of a flow graph enables such an implementation to be efficient. We also present an informal description of the running-time complexity of any instantiation of the semantics. Section 7 develops several instantiations and proves their running-time complexity. We consider a slight modification to the semantics and a new instantiation in Section 8. Section 9 discusses related work.

2 Language

The language (see Figure 1) contains *simple* expressions, call-by-value function applications, primitive operations, constructor applications, and case expressions. A simple expression is either a constant, variable, lambda expression, or recursive function declaration, and is self-evaluating. Applications are evaluated left to right.

The primitive operations that we consider manipulate first class reference objects. The expression $box(e)$ creates a new reference that contains the value of e, $unbox(e)$ returns the value stored in the reference denoted by e, and $set\text{-}box!(e_1, e_2)$ replaces the value stored in the reference denoted by e_1 with the value of e_2.

Constructor applications are of variable arity, where c ranges over a set of constructor names. A case expression compares the value of its first subexpression against the constructor name and arity of the given pattern. If the value matches, then it is destructured, each x_i is bound,

$$
\begin{array}{lcl}
e & ::= & k \mid x \mid \lambda x.e \mid (rec\ f\ \lambda x.e) \mid (e_1\ e_2) \\
 & \mid & p(e_1, \ldots, e_m) \mid c(e_1, \ldots, e_m) \\
 & \mid & case(e_1, c(x_1, \ldots, x_m) \Rightarrow e_2, y \Rightarrow e_3) \\
k & \in & Const \\
x, y & \in & Var \\
p & \in & Primop = \{box, unbox, set\text{-}box!\} \\
c & \in & Constructor
\end{array}
$$

Figure 1: The language

and evaluation continues with e_2. Otherwise, y is bound to the value and evaluation continues with e_3.

We use the usual conventions of free and bound variables. Note that a case expression binds $x_1, \ldots, x_m$ in e_2 and y in e_3. A program P is an expression with no free variables. We assume that each subexpression occurrence in a program has been assigned a unique label l drawn from an infinite set *Label*. We indicate a labeled expression as either e_l or $[e]_l$.

3 Exact Semantics

Our framework is based on an abstract interpretation[6, 5] of an operational semantics. The exact semantics of a program P is specified by a set of exact states *State* and a transition function $\longrightarrow$, both specific to P. The semantics works directly over the source language; there is no transformation to an intermediate form such as continuation-passing style[1].

The semantics explicitly constructs an *exact flow graph.* For each expression e that is evaluated, a node is added to the graph to the store the value of e. Each time a value "flows" from one node to another, a directed edge is added to the graph to indicate the flow.

3.1 Exact States

The definitions of *State* and its components appears in Figure 2. $A \times A'$ and $A + A'$ denote the cartesian product and disjoint union of A and A', respectively. $A \rightarrow A'$ denotes the set of partial functions from A to A'. A^* denotes the set of finite sequences of elements of A. $\mathcal{P}(A)$ is the powerset of A. Note that *State* is well defined, since each definition refers only to sets defined below it.

An exact state contains components that are similar to those that might be found in a more typical abstract machine (*e.g.,* program counter, environment, stack, heap). The *Label* component of the state corresponds to the program counter. The *BindingEnv* component corresponds to the current environment.

The *Nodes* and *Edges* components of *State* define a graph, where the set of nodes is represented as a partial function mapping a unique identifier (*NodeLabel*) to the value of the node. There are two kinds of nodes that appear in the semantics. The *expression node* $\langle l, b\rangle$ contains the value of e_l evaluated in environment b. The value of free variable x in e_l can be found in the *variable node* $\langle x, b(x)\rangle$. Variable nodes are introduced whenever bindings are constructed, *e.g.,* in function application or case evaluation.

Note that a value in this semantics is a pair $\langle l, b\rangle$ that denotes an expression node in the graph. Since expression nodes are associated with a program expression and a binding environment used in the evaluation of this expression, this representation is sufficient to capture all "concrete" values generated during program evaluation.

Informally, *contours* are used to allocate new dynamic activation frames. In the following, we represent a contour as a list of *call-site* labels, where each call-site label is the label of an application expression. In contour $\langle l_1, \ldots, l_m\rangle$, l_1 denotes the most recent call-site and l_m denotes the least recent. The empty contour is $\langle\rangle$. We write $l{:}\langle l_1, \ldots, l_m\rangle$ to append call-site l onto $\langle l_1, \ldots, l_m\rangle$, yielding $\langle l, l_1, \ldots, l_m\rangle$. Abstractions using call-site information have been proposed elsewhere[10, 20, 18]. Our semantics, however, is not closely tied to this particular contour representation; in In Section 8, we present an alternative and discuss its implications.

3.2 Exact Transition Function

The definition of $\longrightarrow$ for the functional core of the language appears in Figure 3. The rules for primitives, constructors, and conditionals can be found in Appendix A. The definition relies on the auxiliary syntactic function *first*, which maps an expression to the label of its leftmost outermost simple subexpression.

Definition (first)

1. $\mathit{first}(e_l) = l$ if e is simple.
2. $\mathit{first}((e_1\ e_2)) = \mathit{first}(e_1)$
3. $\mathit{first}(p(e_1, \ldots, e_m)) = \mathit{first}(e_1)$
4. $\mathit{first}(c(e_1, \ldots, e_m)) = \mathit{first}(e_1)$
5. $\mathit{first}(case(e_1, c(x_1, \ldots, x_m) \Rightarrow e_2, y \Rightarrow e_3)) = \mathit{first}(e_1)$

■

The initial state for program P is:

$$s_0 = \langle \mathit{first}(P), \lambda x.\ \bot, \langle\rangle, \lambda n.\ \bot, \{\}\rangle.$$

As an explanation of the notation used in Figure 3, $N(l, b) = \bot$ is true iff the partial function N is *not* defined on node label $\langle l, b\rangle$. Similarly, $N(l, b) = v$ is true iff N is defined on node label $\langle l, b\rangle$. We use $f[a_1 \mapsto a_2]$ to denote the function which agrees with f on all arguments except possibly a_1, which it maps to a_2. We use $f[a_1, a_2 \mapsto a_3]$ to abbreviate $f[a_1 \mapsto a_3][a_2 \mapsto a_3]$.

The rules CONST, VAR, LAMDBA, and REC define what happens when the program counter points to an expression that can be immediately evaluated. Each of these rules adds a new node to the graph containing the value of the expression. VAR also adds an edge to record the flow of the value from the variable node to the expression node. REC builds a recursive function by adding a new variable node $\langle f, cn\rangle$ to store the function,

$$
\begin{array}{rcll}
s & \in & \mathit{State} & = \mathit{Label} \times \mathit{BindingEnv} \times \mathit{Contour} \times \mathit{Nodes} \times \mathit{Edges} \\
N & \in & \mathit{Nodes} & = \mathit{NodeLabel} \rightarrow \mathit{Value} \\
E & \in & \mathit{Edges} & = \mathcal{P}(\mathit{Edge}) \\
n_1 \rightsquigarrow n_2 & \in & \mathit{Edge} & = \mathit{NodeLabel} \times \mathit{NodeLabel} \\
n & \in & \mathit{NodeLabel} & = (\mathit{Label} \times \mathit{BindingEnv}) + (\mathit{Var} \times \mathit{Contour}) \\
v & \in & \mathit{Value} & = \mathit{Label} \times \mathit{BindingEnv} \\
b & \in & \mathit{BindingEnv} & = \mathit{Var} \rightarrow \mathit{Contour} \\
cn & \in & \mathit{Contour} & = \mathit{Label}^*
\end{array}
$$

Figure 2: Exact States

If $[\![k_l]\!] \in P$ and $N(l,b) = \perp$, then $\langle l, b, cn, N, E\rangle \longrightarrow \langle l, b, cn, N[\langle l, b\rangle \mapsto \langle l, b\rangle], E\rangle$ CONST

If $[\![x_l]\!] \in P$ and $N(l,b) = \perp$, VAR
then $\langle l, b, cn, N, E\rangle \longrightarrow \langle l, b, cn, N[\langle l, b\rangle \mapsto N(x, b(x))], E \cup \{\langle x, b(x)\rangle \rightsquigarrow \langle l, b\rangle\}\rangle$

If $[\![(\lambda x.e)_l]\!] \in P$ and $N(l,b) = \perp$, then $\langle l, b, cn, N, E\rangle \longrightarrow \langle l, b, cn, N[\langle l, b\rangle \mapsto \langle l, b\rangle], E\rangle$ LAMDBA

If $[\![(\mathit{rec}\, f\, (\lambda x.e)_{l'})_l]\!] \in P$ and $N(l,b) = \perp$, REC
then $\langle l, b, cn, N, E\rangle \longrightarrow \langle l, b, cn, N[\langle l, b\rangle, \langle f, cn\rangle \mapsto \langle l', b[f \mapsto cn]\rangle], E\rangle$

If $[\![([e_1]_l\, e_2)]\!] \in P$ and $N(l,b) = v$, then $\langle l, b, cn, N, E\rangle \longrightarrow \langle \mathit{first}(e_2), b, cn, N, E\rangle$ ARG

If $[\![([e_1]_{l_1}\, [e_2]_{l_2})_l]\!], [\![(\lambda x.e_{l''})_{l'}]\!] \in P$, $N(l_1, b) = \langle l', b'\rangle$, and $N(l_2, b) = v$, CALL
then $\langle l_2, b, cn, N, E\rangle \longrightarrow \langle \mathit{first}(e_{l''}), b'', l{:}cn, N[\langle x, l{:}cn\rangle \mapsto v], E \cup \{\langle l_2, b\rangle \rightsquigarrow \langle x, l{:}cn\rangle, \langle l'', b''\rangle \rightsquigarrow \langle l, b\rangle\}\rangle$
where $b'' = b'[x \mapsto l{:}cn]$

If $[\![\lambda x.e_l]\!] \in P$, $\langle l, b\rangle \rightsquigarrow \langle l', b'\rangle \in E$, and $N(l,b) = v$, RETURN
then $\langle l, b, l'{:}cn, N, E\rangle \longrightarrow \langle l', b', cn, N[\langle l', b'\rangle \mapsto v], E\rangle$

Figure 3: Exact Transition Rules for Functional Core

```
let f = λ (x) x
    g = λ (h y z)
          begin
            (h y)
            (h z)
          end
in begin
     (g f 1 2)
     (g f 3 4)
   end
```

Figure 4: A higher-order program.

$((\lambda fg.(\mathbf{begin}\ (g_4\, f_5\, 1_6\, 2_7)_3\ (g_9\, f_{10}\, 3_{11}\, 4_{12})_8\ \mathbf{end})_2)_1$
$(\lambda x.x_{14})_{13}$
$(\lambda hyz.(\mathbf{begin}\ (h_{18}\, y_{19})_{17}\ (h_{21}\, z_{22})_{20}\ \mathbf{end})_{16})_{15}$
$)_0$

Figure 5: Labeled version of sample program.

and by extending the function's environment to point to the new node.

The remaining rules define how the machine proceeds after finding the value of an expression. After evaluating a function, the machine uses ARG to start evaluating the argument. CALL describes how to proceed after the argument has been evaluated, assuming that the evaluation of the function yielded a "closure", *i.e.,* the label of a λ-expression paired with a binding environment. Under this rule, control transfers to the body of the function in an extension of the closure's environment. Also, the edge $\langle l'', b''\rangle \rightsquigarrow \langle l, b\rangle$ is added to allow control to return after the call is complete. The RETURN rule follows this edge from the body of the function back to the call site.

3.3 Example

To illustrate the semantics, consider the program shown in Figure 4[1] . The labeled version of the same program, after desugaring, is in Figure 5. The exact flow graph for the program is shown in Figure 6. Variable nodes are displayed as rectangles; expresssion nodes are displayed as ovals. The first line of a node contains the node label, the second line contains its value.

For the sake of brevity, variable nodes (*e.g.,* f0, x1) are *not* labeled by a *Var* $\times$ *Contour* pair. Instead, each instance of a variable is assigned a disambiguating integer. The one-to-one correspondence between node labels in the graph and node labels in the semantics is shown in Figure 7.

To simplify the presentation, a binding environment $[x_i \mapsto cn_i]$ is displayed as a list of the variable node labels corresponding to each pair $\langle x_i, cn_i\rangle$. For example, the environment $[h \mapsto \langle 3,0\rangle, y \mapsto \langle 3,0\rangle, z \mapsto \langle 3,0\rangle]$ is displayed as $[\mathrm{h1}, \mathrm{y1}, \mathrm{z1}]$.

[1] We use a trivial extension of the language with sequencing and multiple argument functions

Graph		Semantics
f0	=	$\langle f, \langle 0\rangle\rangle$
g0	=	$\langle g, \langle 0\rangle\rangle$
h0	=	$\langle h, \langle 8,0\rangle\rangle$
h1	=	$\langle h, \langle 3,0\rangle\rangle$
y0	=	$\langle y, \langle 8,0\rangle\rangle$
y1	=	$\langle y, \langle 3,0\rangle\rangle$
z0	=	$\langle z, \langle 8,0\rangle\rangle$
z1	=	$\langle z, \langle 3,0\rangle\rangle$
x0	=	$\langle x, \langle 20,8,0\rangle\rangle$
x1	=	$\langle x, \langle 17,8,0\rangle\rangle$
x2	=	$\langle x, \langle 20,3,0\rangle\rangle$
x3	=	$\langle x, \langle 17,3,0\rangle\rangle$

Figure 7: Correspondence between node labels

For this example, the value stored in a node corresponds to either an integer or a closure. For $[\![k_l]\!] \in P$, we display the value $\langle l, b\rangle$ as the integer k. We display closures as they are represented in the semantics. For example, the value $\langle 15, []\rangle$ displayed in node $\langle \mathbf{e9}, [\mathrm{f0,g0}]\rangle$ denotes the closure of

$$\lambda hyz.\mathbf{begin}\ (h\, y)\ (h\, z)\ \mathbf{end}$$

in an empty environment.

In this program, the outermost application creates the nodes f0 and g0, which receive their values from the anonymous procedures at $\langle \mathbf{e13}, []\rangle$ and $\langle \mathbf{e15}, []\rangle$ respectively. Function g is called twice in the outermost contour; thus, its value flows into $\langle \mathbf{e9}, [\mathrm{f0,g0}]\rangle$ and $\langle \mathbf{e4}, [\mathrm{f0,g0}]\rangle$. Each call to g receives f as its first argument. The first call to g creates the flow of $\langle \mathbf{e5}, [\mathrm{f0,g0}]\rangle$ into h1. Note that h1's contour is $\langle 3,0\rangle$; the 3 corresponds to the label of g's call site. The 0 corresponds to the label of the outermost application. The return value of this call (2) flows from $\langle \mathbf{e16}, [\mathrm{h1,y1,z1}]\rangle$ to $\langle \mathbf{e3}, [\mathrm{f0,g0}]\rangle$.

4 Abstract Semantics

The abstract semantics[5, 6] for a program P is given by a set of abstract states $\widehat{State}$ and an abstract transition function $\widehat{\mathcal{T}} : \widehat{State} \rightarrow \widehat{State}$, both specific to P. The definition of $\widehat{State}$ appears in Figure 8. The most notable difference from the exact semantics is that a node stores a *set* of abstract values, rather than a single value.

Like the exact semantics, the abstract semantics explicitly manipulates a flow graph. Intuitively, an abstract flow graph is constructed by (finitely) partitioning the nodes in the exact graph[22]. The values stored in the nodes of one partition are *represented* by the set of abstract values in the single abstract node corresponding to the partition. As in the exact semantics, a directed edge represents the flow of values from one node to another. In the abstract semantics, an edge can also be interpreted as a subset constraint.

A particular abstraction is chosen by specifying the following:

- $\widehat{Contour}$, a finite set of abstract contours.

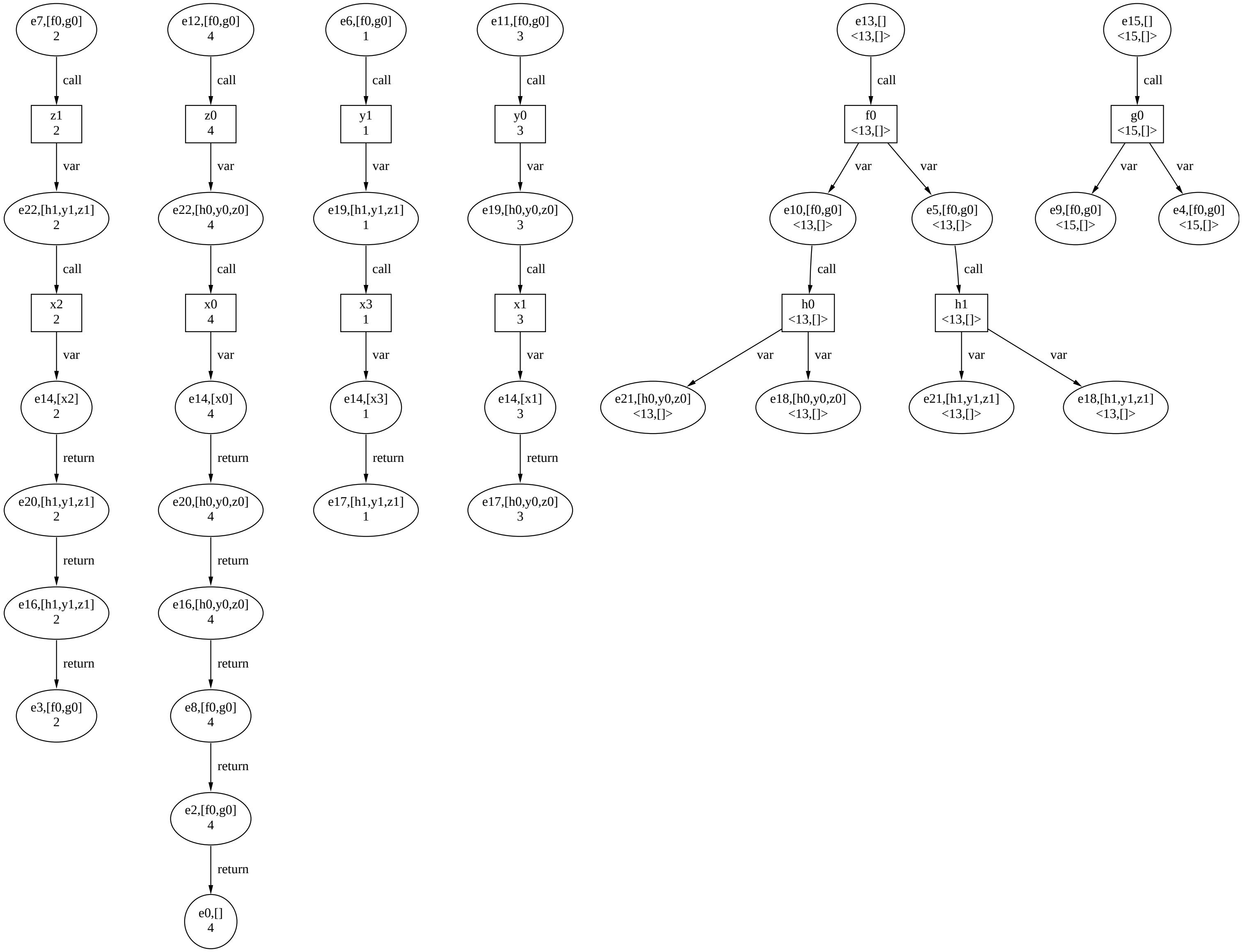

Figure 6: Exact flow graph for a simple higher-order program.

$$
\begin{array}{rcll}
\widehat{s} & \in & \widehat{State} & = \widehat{Nodes} \times \widehat{Edges} \\
\widehat{N} & \in & \widehat{Nodes} & = \widehat{NodeLabel} \rightarrow \mathcal{P}(\widehat{Value}) \\
\widehat{E} & \in & \widehat{Edges} & = \mathcal{P}(\widehat{Edge}) \\
\widehat{n}_1 \rightsquigarrow \widehat{n}_2 & \in & \widehat{Edge} & = \widehat{NodeLabel} \times \widehat{NodeLabel} \\
\widehat{n} & \in & \widehat{NodeLabel} & = (Label \times \widehat{BindingEnv}) + (Var \times \widehat{Contour}) \\
\langle l, \widehat{b} \rangle & \in & \widehat{Value} & = Label \times \widehat{BindingEnv} \\
\widehat{b} & \in & \widehat{BindingEnv} & = Var \rightarrow \widehat{Contour} \\
\widehat{cn} & \in & \widehat{Contour} &
\end{array}
$$

Figure 8: Abstract States

$$
\begin{array}{rcl}
\alpha(\langle l, b, cn, N, E \rangle) & = & \langle \alpha(N), \alpha(E) \rangle \\
\alpha(N) & = & \lambda \widehat{n}. \{\alpha(N(n)) \mid \alpha(n) = \widehat{n}\} \\
\alpha(E) & = & \{\alpha(n_1 \rightsquigarrow n_2) \mid n_1 \rightsquigarrow n_2 \in E\} \\
\alpha(n_1 \rightsquigarrow n_2) & = & \alpha(n_1) \rightsquigarrow \alpha(n_2) \\
\alpha(\langle l, b \rangle) & = & \langle l, \alpha_b(b) \rangle \\
\alpha(\langle x, cn \rangle) & = & \langle x, \alpha_{cn}(cn) \rangle
\end{array}
$$

Figure 9: Abstraction Functions

- α_{cn}, a contour abstraction function, mapping *Contour* to $\widehat{Contour}$.
- $\widehat{:}$ (written in infix), mapping $Label \times \widehat{Contour}$ to $\widehat{Contour}$. It abstracts the exact function ":" that appends a label onto a contour. It must satisfy $\alpha_{cn}(l{:}cn) = l \,\widehat{:}\, \alpha_{cn}(cn)$.
- α_b, a binding environment abstraction function, mapping *BindingEnv* to $\widehat{BindingEnv}$. It must satisfy $\alpha_{cn}(cur(b)) = \widehat{cur}(\alpha_b(b))$, where *cur* maps a binding environment to the contour of its lexically deepest variable and $\widehat{cur}$ maps an abstract binding environment to the abstract contour of its lexically deepest variable.
- $\widehat{extend}$, an abstract environment extension function, written $\widehat{extend}(\widehat{b}, x, \widehat{cn})$. It must satisfy $\alpha_b(b[x \mapsto cn]) = \widehat{extend}(\alpha_b(b), x, \alpha_{cn}(cn))$.

The precise relationship between exact states and abstract states is characterized by a collection of abstraction functions (α) which are homomorphic extensions of the given α_{cn} and α_b. See Figure 9 for details.

4.1 Abstract Transition Function

The abstract transition function builds a dataflow graph. Self-evaluating expressions (constants, lambdas and recursive definitions), create nodes with abstract values that serve as the source of flow. Edges are added to the graph in order to direct the flow to new nodes. Finally, abstract values are propagated across edges, possibly causing the creation of new sources of flow or new edges.

Formally, the abstract transition function $\widehat{\mathcal{T}}$ maps abstract graphs to abstract graphs. It is defined by:

$$\widehat{\mathcal{T}}(\widehat{N}, \widehat{E}) = \langle \tilde{N}, \tilde{E} \rangle$$

The definition of $\tilde{N}$ is given in Figure 10. The first two cases create abstract values in nodes corresponding to expressions that have become *reachable*. The formal definition of reachability is given below. Informally, an abstract node $\langle l, \widehat{b} \rangle$ is reachable if the corresponding exact state could be of the form $\langle l, b, cn, N, E \rangle$, where $\alpha_b(b) = \widehat{b}$. If $\langle l, \widehat{b} \rangle$ is *not* reachable, it implies that e_l is never evaluated in the exact semantics within any exact binding environment approximated by $\widehat{b}$; reachability is thus an important optimization to reduce the amount of flow generated within an abstract graph.

The third case in the definition of $\tilde{N}$ propagates values along edges in the graph. The set of values in a node $\widehat{n}$ is the union of the sets at all of the nodes that flow into $\widehat{n}$.

Definition (Reachability) The following rules define the predicate $reach \subseteq \widehat{NodeLabel}$.

1. $reach(first(P), \lambda x. \perp)$
2. For all $[\![([e_1]_{l_1}\ [e_2]_{l_2})_l]\!] \in P$,
   (a) If $\widehat{N}(l_1, \widehat{b}) \neq \{\}$, then $reach(first(e_2), \widehat{b})$.
   (b) If $\langle l', \widehat{b}' \rangle \in \widehat{N}(l_1, \widehat{b})$, $[\![(\lambda x.e)_{l'}]\!] \in P$, and $\widehat{N}(l_2, \widehat{b}) \neq \{\}$, then $reach(first(e), \widehat{extend}(\widehat{b}', x, l \,\widehat{:}\, \widehat{cur}(\widehat{b})))$.
3. For all $[\![p([e_1]_{l_1}, \ldots, [e_m]_{l_m})_l]\!] \in P$, if $\widehat{N}(l_m, \widehat{b}) \neq \{\}$, then $reach(l, \widehat{b})$.
4. For all $[\![c([e_1]_{l_1}, \ldots, [e_m]_{l_m})_l]\!] \in P$, if $\widehat{N}(l_m, \widehat{b}) \neq \{\}$, then $reach(l, \widehat{b})$.
5. For all $[\![case([e_1]_{l_1}, c(x_1, \ldots, x_m) \Rightarrow, e_2, y \Rightarrow e_3)]\!] \in P$
   (a) For $[\![c(e'_1, \ldots, e'_m)_{l'}]\!] \in P$, if there exists a $\widehat{b}'$ such that $\langle l', \widehat{b}' \rangle \in \widehat{N}(l_1, \widehat{b})$, then $reach(first(e_2), \widehat{extend}(\widehat{b}, x_i, \widehat{cur}(\widehat{b})))$.
   (b) For $[\![c'(e'_1, \ldots, e'_{m'})_{l'}]\!] \in P$, if $c \neq c'$ or $m \neq m'$, and there exists a $\widehat{b}'$ such that $\langle l', \widehat{b}' \rangle \in \widehat{N}(l_1, \widehat{b})$, then $reach(first(e_3), \widehat{extend}(\widehat{b}, y, \widehat{cur}(\widehat{b})))$.

$$\tilde{N}(\widehat{n}) = \begin{cases} \{\langle l, \widehat{b}\rangle\} & \text{if } \widehat{n} = \langle l, \widehat{b}\rangle \text{ and } \mathit{reach}\,(l, \widehat{b}) \\ \{\langle l', \widehat{b}\rangle\} & \text{if } \widehat{n} = \langle l, \widehat{b}\rangle,\ \mathit{reach}\,(l, \widehat{b}), \text{ and } [\![(\mathit{rec}\ f\ (\lambda x.e)_{l'})_l]\!] \in P \\ \bigcup\{\widehat{N}(\widehat{n}') \mid \widehat{n}' \rightsquigarrow \widehat{n} \in \widehat{E}\} & \text{otherwise} \end{cases}$$

Figure 10: Definition of $\tilde{N}$

■

The definition of $\tilde{E}$ for the functional core appears in Figure 11. The remaining cases are given in Appendix B. The definition is given as a union of sets of edges. Most of the edges are directly derived from the exact semantics. For example, edges of the form $\langle x, \widehat{b}(x)\rangle \rightsquigarrow \langle l, \widehat{b}\rangle$ correspond to variable lookup. As another example, for expressions of the form $[\![([e_1]_{l_1}\ [e_2]_{l_2})_l]\!]$, the edge $\langle l_2, \widehat{b}\rangle \rightsquigarrow \langle x, l \mathbin{\widehat{:}} \widehat{cur}(\widehat{b})\rangle$ corresponds to the flow of a value from the argument to the parameter of the function. The only kind of edges which are not directly derived from the exact semantics are of the form $\langle y, \widehat{b}'(y)\rangle \rightsquigarrow \langle y, \widehat{b}''(y)\rangle$. As we discuss in Section 7, these edges have the effect of copying bindings from one environment to another.

4.2 Correctness

We use $\widehat{T}^i$ to denote the i'th composition of $\widehat{T}$ with itself. We use $\xrightarrow{i}$ for the i-step exact transition relation and $\xrightarrow{*}$ for the reflexive transitive closure of $\longrightarrow$. The initial abstract state for a program is $\widehat{s_0} = \langle \lambda\widehat{n}.\{\}, \{\}\rangle$. Sets of abstract values are naturally ordered by $\subseteq$. This order can be extended pointwise and componentwise to construct a partial order $\sqsubseteq$ on $\widehat{State}$.

Lemma 4.1 (Monotonicity) *If $\widehat{s} \sqsubseteq \widehat{s}'$, then $\widehat{T}(\widehat{s}) \sqsubseteq \widehat{T}(\widehat{s}')$* ■

If $\widehat{Contour}$ is finite, then all sets in the definition of $\widehat{State}$ are finite. Also, all increasing chains in $\widehat{State}$ are finite. As a consequence, the following definition of least fixed point is well-defined.

Definition (Least fixed point) $\mathit{lfp}\,(\widehat{T}) = \widehat{s}$, such that $\widehat{T}(\widehat{s}) = \widehat{s}$ and $\exists i. \widehat{T}^i(\widehat{s_0}) = \widehat{s}$. ■

Finally, we prove a correctness theorem which relates the abstract and exact semantics. The theorem states that the abstract flow graph which is the least fixed point of the abstract transition function conservatively estimates all exact flow graphs that can arise as the result of the execution of the program.

Theorem 4.1 (Correctness) *For all s such that $s_0 \xrightarrow{*} s$, $\alpha(s) \sqsubseteq \mathit{lfp}\,(\widehat{T})$*

Proof Sketch: We show by induction on i that if $s_0 \xrightarrow{i} s_i = \langle l, b, cn, N, E\rangle$ and $\widehat{T}^{2*i}(\widehat{s_0}) = \widehat{s}_i = \langle \widehat{N}, \widehat{E}\rangle$, then

1. $\alpha(s_i) \sqsubseteq \widehat{s}_i$
2. If there exists an e such that $l = \mathit{first}(e)$ then $\mathit{reach}\,(l, \alpha(b))$.
3. For all $\langle l', b'\rangle$ such that $N(l', b') \neq \perp$, for all $x \in \mathit{dom}(b')$, $\alpha(N(x, b'(x))) \in \widehat{N}(x, \alpha_b(b')(x))$. ■

5 0CFA and Set Based Analysis

5.1 0CFA

In this section, we consider the following instantiation of the semantics:

$$\begin{aligned} &\widehat{Contour} = \{0\} \\ &\alpha_{cn}(cn) = 0 \\ &l \mathbin{\widehat{:}} \widehat{cn} = 0 \\ &\alpha_b(b) = \lambda x.0 \\ &\widehat{extend}\,(\widehat{b}, x, \widehat{cn}) = \lambda y.0 \end{aligned}$$

This version of the semantics is commonly called 0CFA[21]. With the above definitions, abstract contours and binding environments convey no information and can be removed from the abstract semantics. Figure 12 shows the simplified definition of $\widehat{State}$. Each subexpression and variable of the program corresponds to a single node in the graph. Similarly, each constant, constructor, primop, lambda and recursive function expression gives rise to a single abstract value. The definition of $\tilde{E}$ in the abstract transition function also becomes greatly simplified:

$$[\![x_l]\!]:\quad \tilde{E}_l = \{x \rightsquigarrow l \mid \mathit{reach}(l)\}$$

$$[\![(\mathit{rec}\ f\ \lambda x.e)_l]\!]:\quad \tilde{E}_l = \{l \rightsquigarrow f \mid \mathit{reach}(l)\}$$

$$[\![([e_1]_{l_1}\ [e_2]_{l_2})_l]\!]:$$
$$\tilde{E}_l = \{l_2 \rightsquigarrow x,\ l'' \rightsquigarrow l \mid l' \in \widehat{N}(l_1),\ [\![(\lambda x.e_{l''})_{l'}]\!] \in P\}$$

The effect of the abstraction is to establish an edge from a variable node to each expression node where the variable is referenced and, for each abstract closure stored in a function node, to establish an edge from the argument node to the parameter of the closure and an edge from the body of the lambda to the call site.

Figure 13 shows the flow graph for the program given in Figure 4 under the 0CFA instantiation. Because this analysis does not rely on binding environments or contours to disambiguate abstract values yielded by an expression evaluated in multiple contexts, the graph is simpler in structure than the graph produced by the exact semantics. As in the exact graph, integer values are displayed directly. Because there are no environments, closure values appear as the label of the lambda expression (*e.g.*, $\langle 13\rangle$ for $\lambda x.x$).

$$\tilde{E} = \bigcup_{l,\widehat{b}} \tilde{E}_{l\widehat{b}}$$

Each subexpression below induces a set of edges $\tilde{E}_{l\widehat{b}}$, for each binding environment $\widehat{b}$.

$$[\![x_l]\!] : \quad \tilde{E}_{l\widehat{b}} \quad = \quad \{\langle x, \widehat{b}(x)\rangle \rightsquigarrow \langle l, \widehat{b}\rangle \mid \mathit{reach}\,(l, \widehat{b})\}$$

$$[\![(\mathit{rec}\, f\ \lambda x.e)_l]\!] : \quad \tilde{E}_{l\widehat{b}} \quad = \quad \{\langle l, \widehat{b}\rangle \rightsquigarrow \langle f, \widehat{\mathit{cur}}(\widehat{b})\rangle \mid \mathit{reach}\,(l, \widehat{b})\}$$

$$[\![([e_1]_{l_1}\ [e_2]_{l_2})_l]\!] : \quad \tilde{E}_{l\widehat{b}} \quad = \quad \left\{ \begin{array}{l|l} \langle l_2, \widehat{b}\rangle \rightsquigarrow \langle x, l \,\widehat{:}\, \widehat{\mathit{cur}}(\widehat{b})\rangle, & \langle l', \widehat{b}'\rangle \in \widehat{N}(l_1, \widehat{b}), [\![(\lambda x.e_{l''})_{l'}]\!] \in P \\ \langle l'', \widehat{b}''\rangle \rightsquigarrow \langle l, \widehat{b}\rangle, & \widehat{b}'' = \widehat{\mathit{extend}}\,(\widehat{b}', x, l \,\widehat{:}\, \widehat{\mathit{cur}}(\widehat{b})), \\ \langle y, \widehat{b}'(y)\rangle \rightsquigarrow \langle y, \widehat{b}''(y)\rangle & y \in \mathit{dom}(\widehat{b}') \end{array} \right\}$$

Figure 11: Definition of $\tilde{E}$ for functional core.

$$\begin{array}{rcl} \widehat{\mathit{State}} & = & \widehat{\mathit{Nodes}} \times \widehat{\mathit{Edges}} \\ \widehat{\mathit{Nodes}} & = & \widehat{\mathit{NodeLabel}} \rightarrow \mathcal{P}(\widehat{\mathit{Value}}) \\ \widehat{\mathit{Edges}} & = & \mathcal{P}(\widehat{\mathit{Edge}}) \\ \widehat{\mathit{Edge}} & = & \widehat{\mathit{NodeLabel}} \times \widehat{\mathit{NodeLabel}} \\ \widehat{\mathit{NodeLabel}} & = & \mathit{Label} + \mathit{Var} \\ \widehat{\mathit{Value}} & = & \mathit{Label} \end{array}$$

Figure 12: OCFA Abstract States.

The two calls to g merge the values 2 and 4 into z and the values 1 and 3 into y. The calls to h further merge these values into x, which has the value $\{1, 2, 3, 4\}$. While the graph produced by a 0CFA-style analysis is thus relatively coarse compared to the graph generated by the exact semantics, the abstraction is nonetheless useful. Notice that the graph reveals all possible call sites for all functions; for example, it reveals that f occurs in a call position at **e21** and **e18** and that f is not applied at any other call-site.

5.2 Set-Based Analysis

Set-based analysis is a program analysis technique that associates a (possibly infinite) set of values with each program variable. It is based on ignoring inter-variable dependencies. We summarize the relevant technical parts of [8] in the remainder of this section.

The set-based analysis of a program P is defined via a judgement of the form $\mathcal{E} \vdash e \rightsquigarrow V$ where $\mathcal{E}$ is a *set environment* and V is a set of *constraint values*. The definition appears in Figure 14 [2]. Note that for a given $\mathcal{E}$ and e, there are potentially many V such that $\mathcal{E} \vdash e \rightsquigarrow V$.

Definition (Constraint value) A constraint value cv is either a constant, a lambda expression, or a constructor application of other constraint values.

$$cv \quad ::= \quad k \mid \lambda x.e \mid c(cv_1, \ldots, cv_m)$$

Definition (Set Environment) A set environment is a mapping from the variables of P to sets of constraint values. ■

The set-based operational semantics is similar to a standard operational semantics, except that constraint values (closures, in particular) omit an environment component. Instead, all environments are collapsed into the single set environment $\mathcal{E}$, which is used for variable lookup. *Safety* conditions are imposed on set environments in order to ensure that they contain the proper variable bindings.

Definition (Safety) A set environment $\mathcal{E}$ is safe with respect to program P if every derivation $\mathcal{E} \vdash P \rightsquigarrow V$ satisfies the following three conditions.

1. In every use of S-APP, $V_2 \subseteq \mathcal{E}(x)$.
2. In every use of S-MATCH, if $c(cv_1, \ldots, cv_m) \in V_1$, then $cv_i \in \mathcal{E}(x_i)$ for all $i = 1, \ldots, m$.
3. In every use of S-ELSE, if $cv \in V_1$ and $cv \neq c(cv_1, \ldots, cv_m)$, then $cv \in \mathcal{E}(y)$.

■

Set environments can be partially ordered by the pointwise extension of the subset ordering on their ranges: $\mathcal{E}_1 \sqsubseteq \mathcal{E}_2$ if for all x, $\mathcal{E}_1(x) \subseteq \mathcal{E}_2(x)$. We can also define the greatest lower bound of two set environments in the natural way: $(\mathcal{E}_1 \sqcap \mathcal{E}_2)(x) = \mathcal{E}_1(x) \cap \mathcal{E}_2(x)$. The property of safety is preserved under greatest lower bounds: if $\mathcal{E}_1$ and $\mathcal{E}_2$ are safe with respect to P, then $\mathcal{E}_1 \sqcap \mathcal{E}_2$ is safe with respect to P. Consequently, for every program P, there is a minimal set environment $\mathcal{E}_{Pmin}$ that is safe with respect to P. This environment is the *set-based analysis* of P.

The minimal set environment for the example program

[2] For simplicity, we do not treat primitive operations or recursive definitions.

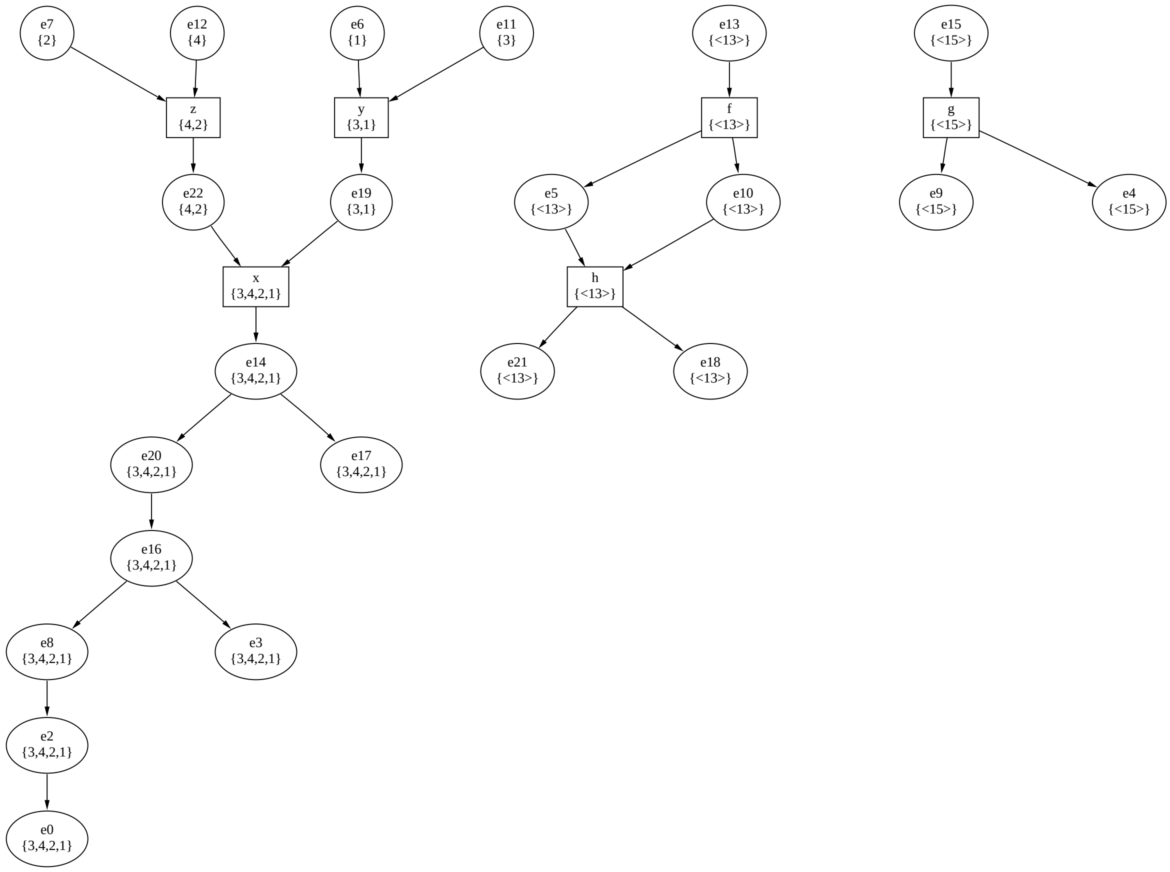

Figure 13: 0CFA abstraction of the exact flow graph shown in Figure 6.

$$\frac{}{\mathcal{E} \vdash x \rightsquigarrow \mathcal{E}(x)} \qquad \text{S-VAR}$$

$$\frac{}{\mathcal{E} \vdash k \rightsquigarrow \{k\}} \qquad \text{S-CONST}$$

$$\frac{}{\mathcal{E} \vdash \lambda x.e \rightsquigarrow \{\lambda x.e\}} \qquad \text{S-LAMDBA}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_1 \qquad \mathcal{E} \vdash e_2 \rightsquigarrow V_2 \qquad \mathcal{E} \vdash e \rightsquigarrow V}{\mathcal{E} \vdash (e_1\, e_2) \rightsquigarrow V} \quad \lambda x.e \in V_1 \qquad \text{S-APP}$$

$$\frac{\mathcal{E} \vdash e_i \rightsquigarrow V_i \quad i = 1, \ldots, m}{\mathcal{E} \vdash c(e_1, \ldots, e_m) \rightsquigarrow \{c(cv_1, \ldots, cv_m) \mid cv_i \in V_i\}} \qquad \text{S-CONS}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_1 \qquad \mathcal{E} \vdash e_2 \rightsquigarrow V_2}{\mathcal{E} \vdash case(e_1, c(x_1, \ldots, x_m) \Rightarrow e_2, y \Rightarrow e_3) \rightsquigarrow V_2} \quad \exists c(cv_1, \ldots, cv_m) \in V_1 \qquad \text{S-MATCH}$$

$$\frac{\mathcal{E} \vdash e_1 \rightsquigarrow V_1 \qquad \mathcal{E} \vdash e_3 \rightsquigarrow V_3}{\mathcal{E} \vdash case(e_1, c(x_1, \ldots, x_m) \Rightarrow e_2, y \Rightarrow e_3) \rightsquigarrow V_3} \quad \exists cv \in V_1\, s.t.\, cv \neq c(cv_1, \ldots, cv_m) \qquad \text{S-ELSE}$$

Figure 14: Set-Based Operational Semantics

in Figure 4 is given by:

$$\begin{array}{rcl}\mathcal{E}_{Pmin}(x) & = & \{1,2,3,4\}\\ \mathcal{E}_{Pmin}(y) & = & \{1,3\}\\ \mathcal{E}_{Pmin}(z) & = & \{2,4\}\\ \mathcal{E}_{Pmin}(f) & = & \{\lambda x.x\}\\ \mathcal{E}_{Pmin}(h) & = & \{\lambda x.x\}\\ \mathcal{E}_{Pmin}(g) & = & \{\lambda hyz.\mathbf{begin}\ (h\,y)\ (h\,z)\ \mathbf{end}\}\end{array}$$

As another example, consider the program P:

$$\begin{array}{l}\mathbf{let}\ f = \lambda x.c(x)\\ \mathbf{in}\ (f\ \ (f\ \ 1))\end{array}$$

The minimal set enironment for this program maps x to the infinite set $\{1, c(1), c(c(1)), c(c(c(1))), \ldots\}$.

5.3 Relating OCFA to Set-Based Analysis

As shown by the above example, the set-based analysis of a program can yield an infinite set. However, [8] proves that it is possible to use regular tree grammars to *represent* the sets defined by set-based analysis. In this section, we show that a similar theorem applies to the states computed by the 0CFA instantiation of the abstract semantics. In particular, we show that an abstract state implicitly contains a regular tree grammar and that this grammar exactly characterizes the set-based analysis of a program.

We first show how to construct a regular tree grammar from a collection of abstract nodes $\widehat{N}$. The nonterminals will be node labels and will derive constraint values.

Definition (Grammar) $Gr(\widehat{N})$ is a grammar with the following components:

$$\begin{array}{l}Nonterminals = \widehat{NodeLabel}\\ Productions = \{\widehat{n} \rightarrow \widehat{v} \mid \widehat{v} \in \widehat{N}(\widehat{n})\}\\ \quad\quad \cup \{l \rightarrow k \mid [\![k_l]\!] \in P\}\\ \quad\quad \cup \{l \rightarrow \lambda x.e \mid [\![(\lambda x.e)_l]\!] \in P\}\\ \quad\quad \cup \{l \rightarrow c(l_1, \ldots, l_m) \mid [\![(c([e_1]_{l_1}, \ldots, [e_m]_{l_m}))_l]\!] \in P\}\end{array}$$

■

We use the notation $\widehat{n} \Rightarrow^*_G cv$ to mean that $\widehat{n}$ derives cv in G. Because variables are nonterminals, a grammar can be used to define a set environment.

Definition (*Env*) $Env(G)$ is the set environment which maps a variable x to the set $\{cv \mid x \Rightarrow^*_G cv\}$. ■

For example, re-consider the program P shown earlier:

$$\begin{array}{l}\mathbf{let}\ f\ = \lambda x.(c(x_{l_3}))_{l_2}\\ \mathbf{in}\ (f\,(f\,1_{l_1}))\end{array}$$

We have labeled several relevant subexpressions of the program. Let $\langle \widehat{N}, \widehat{E}\rangle$ be the least fixed point of the abstract transition function for P. It is easy to show that $\widehat{N}(x) = \widehat{N}(l_3) = \{l_1, l_2\}$. Consequently, the grammar $Gr(\widehat{N})$ contains the following productions:

$$\begin{array}{ll}x \rightarrow l_1 & l_3 \rightarrow l_1\\ x \rightarrow l_2 & l_3 \rightarrow l_2\\ l_1 \rightarrow 1 & l_2 \rightarrow c(l_3)\end{array}$$

Hence,
$\{cv \mid x \Rightarrow^*_{Gr(\widehat{s})} cv\} = \{1, c(1), c(c(1)), c(c(c(1))), \ldots\}$, which is exactly $\mathcal{E}_{Pmin}(x)$.

From the perspective of set-based analysis, the 0CFA abstract semantics contains two kinds of edges. The first kind corresponds to safety constraints. For example, the edge $l_2 \rightsquigarrow x$ from an argument node to a variable node corresponds to the first safety constraint, $V_2 \subseteq \mathcal{E}(x)$. The second kind of edge corresponds to constraint value sets which are copied from the antecedent to the consequent in a set-based rule. For example, the edge $l'' \rightsquigarrow l$ from the body of a lambda to the call-site corresponds to the S-APP rule in Figure 14.

The following theorem formalizes the correspondence between 0CFA and set-based analysis.

Theorem 5.1 (Equivalence) *Let a program P be given. Let $\widehat{\mathcal{T}}$ be the abstract transition function for P. Let $lfp(\widehat{\mathcal{T}}) = \langle \widehat{N}, \widehat{E}\rangle$, $G = Gr(\widehat{N})$ and $\mathcal{E}_{Pabs} = Env(G)$. Then, $\mathcal{E}_{Pmin} = \mathcal{E}_{Pabs}$.* [3]

Proof Sketch: We establish $\mathcal{E}_{Pmin} \sqsubseteq \mathcal{E}_{Pabs}$ by showing that $\mathcal{E}_{Pabs}$ is safe with respect to P. The proof is by contradiction and requires Lemma 5.1. For the other direction, we establish by induction that $Env(Gr(\widehat{N}^i)) \sqsubseteq \mathcal{E}_{Pmin}$, where $\widehat{\mathcal{T}}^i(\widehat{s_0}) = \langle \widehat{N}^i, \widehat{E}^i\rangle$. The result follows because for some index i, $lfp(\widehat{\mathcal{T}}) = \widehat{\mathcal{T}}^i(\widehat{s_0})$. ■

Lemma 5.1 *If $reach(l)$ and $\mathcal{E}_{Pabs} \vdash e_l \rightsquigarrow V$ then $V \subseteq \{cv \mid l \Rightarrow^*_G cv\}$*

6 Implementation and Complexity

By defining the abstract semantics in terms of flow graphs, we enable direct and efficient implementations. First, because $\widehat{\mathcal{T}}$ is monotonic, an implementation can represent the abstract state with a single graph, and make destructive updates when necessary. Furthermore, the definition of $\widehat{N}$ (see Figure 10) makes it clear that is advantageous to represent the edges in the graph using adjacency lists. Given this representation, the abstract values at a node n can be immediately propagated in n's successors.

In our implementation, we maintain a graph in which each node n stores three things:

- R$[n]$, the set of abstract values which have reached n and have been propagated to n's successors.
- P$[n]$, the set of abstract values which have reached n and have not been propagated to n's successors.
- S$[n]$, a list of the successors of n.

[3]In order for this theorem to be true, we must make a slight modification to our semantics which restricts the flow of test values in case expressions.

We also maintain a set of nodes $N = \{n \mid \mathrm{P}[n] \neq \{\}\}$. The algorithm propagates values across edges until the fixed point is reached. For each node n, $\mathrm{R}[n]$ monotonically increases over the execution of the algorithm.

```
while N ≠ {}
  remove some n from N
  for each n' in S[n]
    for each v in P[n]
      if v ∉ R[n'] and v ∉ P[n']
      then P[n'] ← P[n'] ∪ {v}
    if P[n'] ≠ {}
    then N ← N ∪ {n'}
  R[n] ← R[n] ∪ P[n]
  P[n] ← {}
```

The above pseudocode omits the details describing the creation of new nodes and the addition of new edges. Both operations are performed when a value is added to a set $\mathrm{P}[n]$. The program is preprocessed to extract the necessary information from the definitions of reachability and $\tilde{E}$. Preprocessing removes much of the "interpretation overhead" of the analysis and makes the inner loop extremely tight.

The running time complexity of the algorithm is governed by the inner loop. For all nodes n and abstract values v, v is added to $\mathrm{P}[n]$ at most once. Consequently, the inner loop can be executed at most once for each edge and abstract value. Assuming the code in the inner loop is O(1), the worst-case running time of the algorithm is the product of the number of edges and the number of abstract values.

The current implementation is approximately 1500 lines of Scheme code. It is written in Scheme 48 [12], and makes extensive use of the Scheme 48 module system. The implementation is parameterized in exactly the same way as the abstract semantics. To construct a complete abstract interpreter, one must specify an "abstraction" module which defines operations for manipulating abstract contours and abstract binding environments. In order to simplify the construction of abstraction modules, we have created a general tool which characterizes a large class of call-string abstractions including all the ones described in this paper.

7 Instantiations

In the following, let n be the size of the program. We consider three specializations of the framework. The first is the 0CFA analysis given in Section 5. In 0CFA, the number of nodes in the graph is $O(n)$; no contour information is used to disambiguate multiple instantiations of a given procedure. Thus, the number of edges in the graph is $O(n^2)$. Since the number of values is $O(n)$, the worst-case time complexity is $O(n^3)$[8, 14].

We can construct a more sophisticated analysis without sacrificing polynomial-time complexity by disambiguating distinct calls to a function; we do this by preserving only the most recent call-site in the function's binding environment. We call this particular analysis polynomial-time 1CFA; the specifics appear below:

Graph		Semantics
f0	=	$\langle f, \langle 0\rangle\rangle$
g0	=	$\langle g, \langle 0\rangle\rangle$
h0	=	$\langle h, \langle 8\rangle\rangle$
h1	=	$\langle h, \langle 3\rangle\rangle$
y0	=	$\langle y, \langle 8\rangle\rangle$
y1	=	$\langle y, \langle 3\rangle\rangle$
z0	=	$\langle z, \langle 8\rangle\rangle$
z1	=	$\langle z, \langle 3\rangle\rangle$
x0	=	$\langle x, \langle 20\rangle\rangle$
x1	=	$\langle x, \langle 17\rangle\rangle$

Figure 15: Polynomial-time 1CFA node correspondence.

$$\widehat{Contour} = Label$$
$$\alpha_{cn}(\langle l_1, \ldots, l_m\rangle) = l_1$$
$$l' \,\widehat{:}\, l = l'$$
$$\alpha_b(b) = \lambda x.\alpha_{cn}(cur(b))$$
$$\widehat{extend}(\widehat{b}, x, l) = \lambda y.l$$

$\widehat{BindingEnv}$ is restricted to constant functions that map all variables in an abstract closure to a particular call-site, and hence its size is $O(n)$. Consequently, there are $O(n^2)$ nodes and hence $O(n^4)$ edges. There are $O(n^2)$ values that can flow along these edges corresponding to the $O(n^2)$ abstract closures that can be constructed. The worst-case complexity of this analysis is thus $O(n^6)$.

This analysis uses edges of the form $\langle y, \widehat{b}'(y)\rangle \rightsquigarrow \langle y, \widehat{b}''(y)\rangle$ (see Figure 11) to copy bindings from a closure's binding environment to the environment in which the application is evaluated. This is necessary because under this instantiation, function application requires *all* variables in the function's closure to be mapped to the application's call-site label. In other words, for any given application of a function f, every free variable in f's closure is mapped to the same contour, regardless of the call-site in which the variable was originally bound. The additional edges ensure that binding values are properly propagated from the closure to the "activation frame" associated with the application.

The graph produced by this analysis on the example shown in Figure 4 is given in Figure 7. This graph results in less merging than the 0CFA instantiation because call site information is used to disambiguate multiple calls to the same function. For example, variable node z0 contains z's binding value in the second call to g, and z1 contains z's value in the first call. However, the two calls to h in g merge values passed in the two calls to g. Consequently, the abstract value of this program is the set {2,4}. The correspondence between node labels in the graph and in the semantics is given in Figure 7.

Notice that a slightly more costly abstraction that preserves information about a function's *two* most recent call-sites would produce a flow graph with exact information. Under such an analysis, the graph for our sample program would associate four contours for x corresponding to all of f's possible call histories.

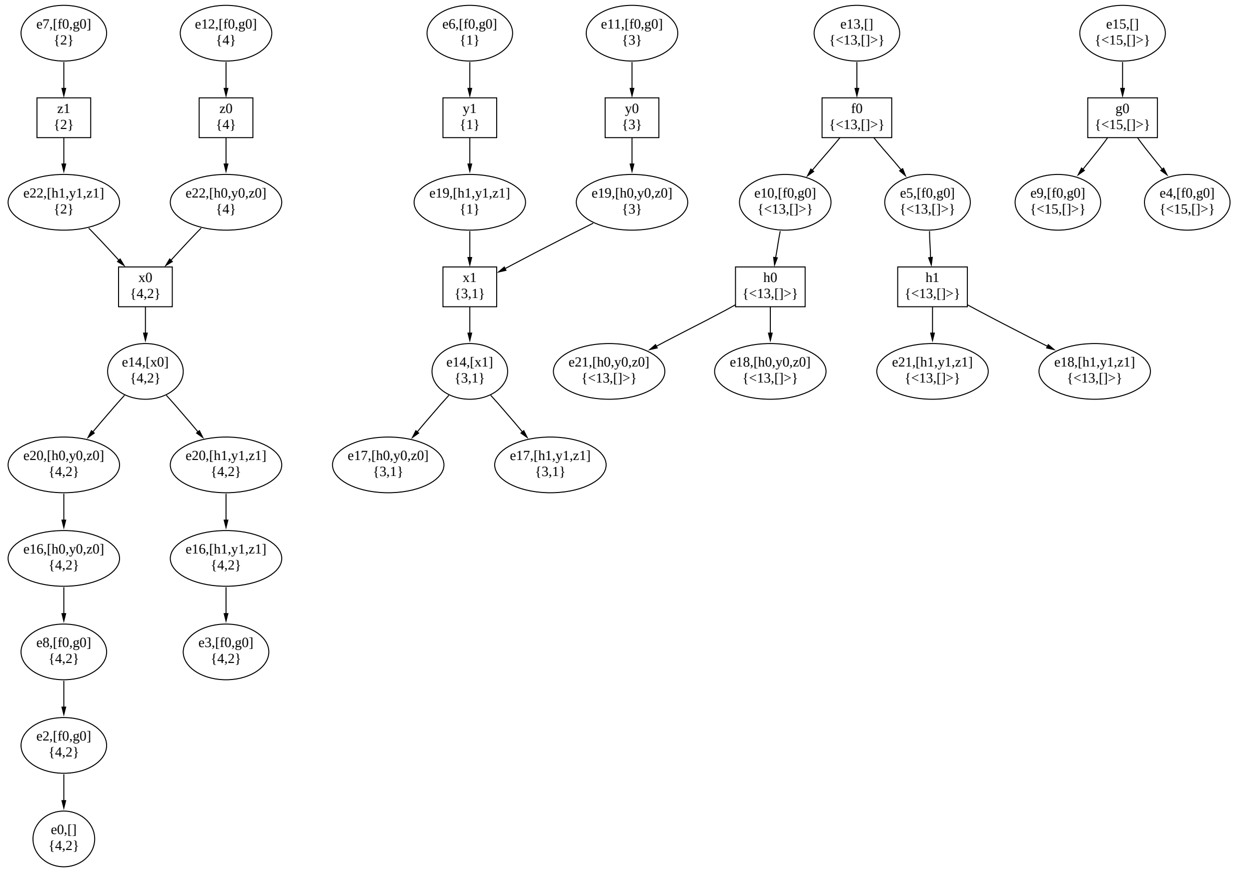

Figure 16: Polynomial-time 1CFA flow graph.

By allowing $\widehat{BindingEnv}$ to contain non-constant functions, we derive an exponential-time analysis that is related to the 1CFA analysis described in [21]. Whereas in polynomial-time 1CFA all free variables in an abstract closure are mapped to a single call-site, the exponential-time 1CFA relaxes this constraint and associates a variable in an abstract closure with the call-site at which it was bound. The only difference between this analysis and polynomial-time 1CFA is the treatment of binding environments:

$$\begin{array}{l} \alpha_b(b) = \lambda x.\alpha_{cn}(b(x)) \\ \widehat{extend}(\widehat{b}, x, l) = \widehat{b}[x \mapsto l] \end{array}$$

Since $\widehat{BindingEnv}$ now contains a full function space from variables to abstract contours, there can be an exponential number of closures constructed and an exponential number of expression nodes.

8 An Alternative Contour Representation

We have represented contours in terms of call-strings. While reasonably expressive, this particular representation has some drawbacks. Most notably, call-string information cannot be used to disambiguate distinct calls to a function based on the abstract values of the function's arguments. In other words, the accuracy of any instantiation of this framework is closely tied to the length of an abstract contour's call-string, not the abstract values generated by the interpretation. Thus, calls to a function made from the same call-site may not be disambiguated if the contour's call-string is not long enough, despite the fact that different calls supply different arguments.

The framework presented here is not dependent on representing contours as call-strings. It is straightforward to define alternative representations. For example, consider a modification in which exact contours contain call-strings augmented with information that records the argument value bound at each call-site.

$$\begin{array}{l} Contour = (Value \times Label)^* \\ \langle v', l' \rangle : \langle \langle v, l \rangle, \ldots \rangle = \langle \langle v', l' \rangle, \langle v, l \rangle, \ldots \rangle \end{array}$$

We can define a simple abstraction of this representation that disregards call-strings and preserves information about closures that are passed as arguments. Abstract binding environments can be treated as in the exponential variant of 1CFA.

$$\begin{array}{l} \widehat{Contour} = \{\lambda x.e \mid [\![\lambda x.e]\!] \in P\} + \{\text{NON-LAMBDA}\} \\ \alpha_{cn}(\langle \langle l, b \rangle, l' \rangle, \ldots) = \begin{cases} \lambda x.e & \text{if } [\![(\lambda x.e)_l]\!] \in P \\ \text{NON-LAMBDA} & \text{otherwise} \end{cases} \\ \langle \langle l, b \rangle, l' \rangle \,\widehat{:}\, \widehat{cn} = \begin{cases} \lambda x.e & \text{if } [\![(\lambda x.e)_l]\!] \in P \\ \text{NON-LAMBDA} & \text{otherwise} \end{cases} \\ \alpha_b(b) = \lambda x.\alpha_{cn}(b(x)) \\ \widehat{extend}(\widehat{b}, x, \widehat{cn}) = \widehat{b}[x \mapsto \widehat{cn}] \end{array}$$

Under this abstraction, two calls to a function f which supply closures created from different λ-expressions will be associated with different contours, even if those calls occur from the same call site. Distinct calls to f that supply non-functional values are merged.

For programs that use higher-order procedures, this choice of abstract contours can compute more precise control flow information than is possible with call-string abstractions. For example, consider a simple higher-order recursive function such as Scheme's *map* that takes a function f and a list L and applies f to each element of L. There are two ways in which a call-string based analysis of *map* will lose control flow information. First, if several functions are merged together before being passed as the functional argument to *map*, then call-strings will not separate the functions at the call. Second, even if all calls to *map* pass a different function at each call-site, the recursive call to *map* (or any other local helper function) inside the body of its definition will eventually cause the functional arguments to be merged.

The conditions necessary for the abstract semantics to be correct under this representation strategy are easily derived and are similar to those given in Section 4. We leave an investigation of the merits of these various representation choices for future research.

9 Conclusions and Related Work

Parameterized control-flow analysis via flow graphs not only provides an intuitive and customizable analysis framework, but also offers the potential of becoming usefully integrated within a realistic compiler. Optimizations such as unboxing, function specialization, type recovery, safety and liveness analysis, debugging[2], and global register analysis are all important components in an optimizing compiler toolbox and programming environment. Many of these analyses require efficient, sophisticated and tunable inter-procedural control-flow analyses. We believe that the ideas presented here provide exactly this capability.

There have been a number of previous efforts that rigorously address the control-flow analysis problem for high-level programming languages (*e.g.,* [3, 7, 10, 9, 11]); below, we compare three relevant approaches with the contributions presented here.

Shivers[20] presents a general model for control-flow analysis in Scheme via abstract interpretation of a denotational semantics. The analysis must first translate source expressions to CPS terms; this translation affects the results computed by the analysis[16]. Deriving an efficient implementation from the semantics is unintuitive and requires deviating from the semantic specification (e.g., the time-stamp approximation [21]). The choice of a denotational semantics makes it difficult in general to understand the complexity of implementations directly derived from the analysis. Finally, improving the accuracy of a 0CFA instantiation of the analysis without introducing exponential complexity is problematic; our framework offers a more general notion of abstraction that permits the expression of a hierarchy of polynomial-time abstractions which reflect progressively improved accuracy.

Set-based analysis[8] refers to an operational semantics which ignores all inter-variable dependencies that occur in a program. Because the semantics is tightly tied to this notion, it is not clear how to express a more precise analysis within the framework. We have shown that the 0CFA instantiation of our semantics is sufficient for computing a representation of the sets defined by set-based analysis. Set-based analysis has a simple and intuitive characterization; devising equally simply descriptions of other instantiations of our semantics is a topic for future work.

In order to develop a more precise analysis, [8] describes a polyvariant extension of the *algorithm* for computing set-based analysis. The extension uses information from a monovariant prepass in order to duplicate certain functions. The result of the extended algorithm can be interpreted as a monovariant analysis on a new program which is β-equivalent to the original program. Although β-substitution can be used to capture some aspects of duplication in contour-based analyses, it can not describe certain kinds of merging which occur in the abstract semantics. For example, it can not express the polynomial-1CFA analysis (Section 7) which copies bindings from one contour to another.

Stefanescu and Zhou[22] present an equational framework for the control-flow problem. Their equations have a strong correspondence with the subset constraints of [8] and with the edges of our abstract state. In their work, a program in the source language is closure converted[1] before being analyzed. In the context of our analysis, this transformation corresponds to the restriction of binding environments to constant functions since there are in effect no free variables found within a function after closure conversion. A natural attempt to formulate a 1CFA style analysis in their framework gives a polynomial time approximation[23] similar to the one we describe in Section 7. It appears impossible to express other analyses in their framework, *e.g.,* the exponential 1CFA analysis and the analysis of Section 8, which is not call string based.

References

[1] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Francois Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, 1993.

[3] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[4] William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[5] Patrick Cousot. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Foundation*, pages 303–342. Prentice-Hall, 1981.

[6] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. In *ACM 4^{th} Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[7] Alain Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *17^{th} ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[8] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 306–317, 1994.

[9] Paul Hudak and Jonathan Young. A Collecting Interpretation of Expressions. *ACM Transactions on Programming Languages and Systems*, pages 269–290, April 1991.

[10] Williams Ludwell Harrison III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[11] Neil Jones and Stephen Muchnick. Flow Analysis and Optimization of Lisp-like Structures. In *6^{th} ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[12] Richard Kelsey and Jonathan Rees. Scheme48 Progress Report. *Lisp and Symbolic Computation*, 1994.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[14] Jens Palsberg and Michael Schwartzbach. Safety Analysis versus Type Inference. *Information and Computation, to appear.*

[15] Young Gil Park and Benjamin Goldberg. Escape Analysis on Lists. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 116–127, June 1992.

[16] Amr Sabry and Matthias Felleisen. Is Continuation Passing Useful for Data Flow Analysis. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 1–12, June 1994.

[17] Zhong Shao and Andrew Appel. Space-Efficient Closure Representations. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 150–161, 1994.

[18] Micha Sharir and Amir Pneuli. *Two Approaches to Interprocedural Dataflow Analysis*, pages 189–235. Prentice-Hall, 1981.

[19] Olin Shivers. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation.* MIT Press, 1990.

[20] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[21] Olin Shivers. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, 1991.

[22] Dan Stefanescu and Yuli Zhou. An Equational Framework for the Flow Analysis of Higher-Order Functional Programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 318–327, 1994.

[23] Yuli Zhou, June 1994. Personal Communication.

A Exact Semantics for Language Extensions.

If $[\![p([e_1]_{l_1},\ldots,[e_m]_{l_m})]\!] \in P$ or $[\![c([e_1]_{l_1},\ldots,[e_m]_{l_m})]\!] \in P$, NEXT
then for $1 \le i < m$, if $N(l_i,b)=v$, then $\langle l_i,b,cn,N,E\rangle \longrightarrow \langle \mathit{first}([e_{i+1}]_{l_{i+1}}),b,cn,N,E\rangle$

If $[\![\mathit{box}([e_1]_{l_1})_l]\!] \in P$ and $N(l_1,b)=v$, BOX
then $\langle l_1,b,cn,N,E\rangle \longrightarrow \langle l,b,cn,N[\langle l,b\rangle \mapsto \langle l,b\rangle],E\rangle$

If $[\![\mathit{unbox}([e_1]_{l_1})_l]\!], [\![\mathit{box}(e_{l''})_{l'}]\!] \in P$ and $N(l_1,b)=\langle l',b'\rangle$, UNBOX
then $\langle l_1,b,cn,N,E\rangle \longrightarrow \langle l,b,cn,N[\langle l,b\rangle \mapsto N(l'',b')],E\cup\{\langle l'',b'\rangle \rightsquigarrow \langle l,b\rangle\}\rangle$

If $[\![\mathit{set\text{-}box!}([e_1]_{l_1},[e_2]_{l_2})_l]\!], [\![\mathit{box}(e_{l''})_{l'}]\!] \in P$, $N(l_1,b)=\langle l',b'\rangle$, and $N(l_2,b)=v$, SETBOX
then $\langle l_2,b,cn,N,E\rangle \longrightarrow \langle l,b,cn,N[\langle l,b\rangle \mapsto \mathit{Unspecified}, \langle l'',b'\rangle \mapsto v],E\cup\{\langle l_2,b\rangle \rightsquigarrow \langle l'',b'\rangle\}\rangle$

If $[\![c(e_1,\ldots,[e_m]_{lm})]\!] \in P$ and $N(l_m,b)=v$, CONS
then $\langle l_m,b,cn,N,E\rangle \longrightarrow \langle l,b,cn,N[\langle l,b\rangle \mapsto \langle l,b\rangle],E\rangle$

If $[\![\mathit{case}([e_1]_{l_1}, c(x_1,\ldots,x_m) \Rightarrow [e_2]_{l_2}, y \Rightarrow [e_3]_{l_3})_l]\!] \in P$

If $[\![c([e'_1]_{l'_1},\ldots[e'_m]_{l'_m})_{l'}]\!] \in P$ and $N(l_1,b)=\langle l',b'\rangle$, then MATCH
$\langle l_1,b,cn,N,E\rangle \longrightarrow$
$\langle \mathit{first}([e_2]_{l_2}),b'',cn,N[\langle x_i,cn\rangle \mapsto N(l'_i,b')],E\cup\{\langle l'_i,b'\rangle \rightsquigarrow \langle x_i,cn\rangle \mid 1\le i\le m\}\cup\{\langle l_2,b''\rangle \rightsquigarrow \langle l,b\rangle\}\rangle$
where $b''=b[x_1,\ldots,x_m \mapsto cn]$

If $[\![c'(e'_1,\ldots,e'_m)_{l'}]\!] \in P$, $N(l_1,b)=\langle l',b'\rangle$, and $c\ne c'$ or $m\ne m'$, then ELSE
$\langle l_1,b,cn,N,E\rangle \longrightarrow \langle \mathit{first}([e_3]_{l_3}),b'',cn,N[\langle y,cn\rangle \mapsto \langle l',b'\rangle],E\cup\{\langle l_1,b\rangle \rightsquigarrow \langle y,cn\rangle,\langle l_3,b''\rangle \rightsquigarrow \langle l,b\rangle\}\rangle$
where $b''=b[y\mapsto cn]$

For $i\in\{2,3\}$, if $N(l_i,b)=v$ and $\langle l_i,b\rangle \rightsquigarrow \langle l,b'\rangle \in E$, CONT
then $\langle l_i,b,cn,N,E\rangle \longrightarrow \langle l,b',cn,N[\langle l,b'\rangle \mapsto N(l_i,b)],E\rangle$

B Definition of $\tilde{E}$ for Language Extensions.

$[\![\mathit{unbox}([e_1]_{l_1})_l]\!]$:
$$\tilde{E}_{l\widehat{b}} = \{\langle l'',\widehat{b}'\rangle \rightsquigarrow \langle l,\widehat{b}\rangle \mid \langle l',\widehat{b}'\rangle \in \widehat{N}(l_1,\widehat{b}), [\![\mathit{box}(e_{l''})_{l'}]\!] \in P\}$$

$[\![\mathit{set\text{-}box!}([e_1]_{l_1},[e_2]_{l_2})]\!]$:
$$\tilde{E}_{l\widehat{b}} = \{\langle l_2,\widehat{b}\rangle \rightsquigarrow \langle l'',\widehat{b}'\rangle \mid \langle l',\widehat{b}'\rangle \in \widehat{N}(l_1,\widehat{b}), [\![\mathit{box}(e_{l''})_{l'}]\!] \in P\}$$

$[\![\mathit{case}([e_1]_{l_1}, c(x_1,\ldots,x_m) \Rightarrow [e_2]_{l_2}, y \Rightarrow [e_3]_{l_3})_l]\!]$:
$$\tilde{E}_{l\widehat{b}} = \{\langle l'_i,\widehat{b}'\rangle \rightsquigarrow \langle x_i,\mathit{cur}(\widehat{b})\rangle \mid \langle l',\widehat{b}'\rangle \in \widehat{N}(l_1,\widehat{b}),\ [\![c([e'_1]_{l'_1},\ldots[e'_m]_{l'_m})_{l'}]\!] \in P,\ 1\le i<m\}$$
$$\cup\{\langle l_1,\widehat{b}\rangle \rightsquigarrow \langle y,\mathit{cur}(\widehat{b})\rangle,\ \langle l_2,\widehat{\mathit{extend}}(\widehat{b},x_i,\widehat{\mathit{cur}}(\widehat{b}))\rangle \rightsquigarrow \langle l,\widehat{b}\rangle,\ \langle l_3,\widehat{\mathit{extend}}(\widehat{b},y,\widehat{\mathit{cur}}(\widehat{b}))\rangle \rightsquigarrow \langle l,\widehat{b}\rangle\}$$