An Equational Framework for the Flow Analysis of Higher Order Functional Programs

Dan Stefanescu*
Harvard University
Suffolk University

Yuli Zhou†
MIT Lab for Computer Science

Abstract

We present a simple method for the general flow analysis of high-order functional programs. The method computes an abstraction of the program's runtime environment via a system of monotonic equations. As the environment can grow unbounded, we exploit patterns in the program's control structure (i.e., the call-tree) to determine some static partition of the environment, and merge points in the environment belonging to the same equivalent-class. High order functions are handled by embedding control information into closures. The method is proven correct with respect to a rewriting system based operational semantics. Various implementation issues are also considered.

1 Introduction

This paper presents a simple technique for the static analysis of higher order functional programs. The main idea is to represent the runtime environment of a functional program as a mapping from unique dynamic labels, representing locations in the activation frames of functions, to values. The abstraction method consists in choosing a a relation of equivalence over the set of all dynamic labels, and using the partition to generate a system of monotonic flow equations whose solution provides the result of the analysis.

The result of the paper generalizes previous control-flow analysis techniques (0CFA, 1CFA, etc.) within an equational framework in the style of the original Cousot and Cousot approach [12]. It allows for approximate analysis that is adaptive both to individual programs and to compile-time resource constraints. The method is proven correct with respect to a rewriting system based operational semantics.

*Work supported by DARPA Contract Nr. F19628-92-C-0113 Author current address· 105 Aiken Computation Laboratory, Harvard University, 33 Oxford St, Cambridge, MA 02138 and 622 Fenton Building, Suffolk University, Boston MA 02114. E-mail: dan@das harvard edu, dan@hyper.clas suffolk edu

†Research performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310. Author's current address. NE43-253, MIT Lab for Computer Science, 545 Technology square, Cambridge, MA 02139 E-mail: zhou@abp.lcs.mit edu

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

1.1 Background

Traditional data flow analyses were developed for optimizing compilers of imperative languages [1]. Such an analysis is typically performed on flow-diagrams in which the nodes are basic blocks and the arcs are the program's control points. Properties on the exit arcs of each node are related to those associated with the entry arcs via data flow equations, and the solution of the system of equations provides the result of the analysis.

Cousot and Cousot ([12], see also [15, 16]) developed the theory of abstract interpretations, which unified data flow analysis under a general mathematical framework. Within this framework, an environment is a mapping of variables in the program to values, and a context is a mapping of control arcs to environments. Abstract interpretation is thus aimed at computing an abstract context, which approximates the set of environments obtainable at each control arc. Points in the abstract context are related via monotonic equations over lattices, and the system of equations is related to the static semantics of the program, which provides an exact summary of the program's contexts obtainable for a given set of inputs. In this way a general proof of the correctness of abstract interpretations was obtained.

Inter-procedural flow analysis, however, has the problem of unbounded runtime environment in the presence of recursive procedure calls. Two earlier solutions were given in [13, 28], employing techniques to bound the size of the abstract environment by merging points belonging to "similar" function calls. This is most obvious in the "call-strings" approach, in which the flow variables are indexed by call-strings that encode the invocation history of procedures. Bounding the number of (abstract) call-strings thus has the effect of bounding the size of the environment. Similar techniques were developed for analysis of programs with recursive data-structures. [22] introduced the use of "tokens" to reduce the environment in a general and flexible way. However, all these techniques were developed for first order languages in which function call-sites can be determined statically. More recently, Bourdoncle has generalized these ideas into a framework of dynamic partitioning, which also combines the technique of widening to deal with infinite abstract domains [5, 6].

Abstract interpretation took a new turn when Mycroft [26] adapted it to the strictness analysis of first order functional programs. His work has since been generalized to handle higher-order functions, data-structures and polymorphism [8, 20, 4]. A distinguishing characteristic of these methods is that abstract interpretation is generally viewed as "semantics in abstract domains", which assigns abstract values to expressions in the program. Naturally, functional values grow exponentially with the number of arguments and with levels of function abstraction, thus various techniques for dealing with this explosion have been developed, e.g., to compute only the "envelope" of the function graph [21], only the minimal function graphs needed by the program [23], or to represent function values compactly using type expressions [25]. Even with these optimizations, the practical utility of these methods appears to be limited due to their computational complexity.

In functional languages such as Scheme and ML, where functions are first class values, function call-sites are generally unknown. Shivers [29, 30] rightly identifies the problem with the lack of static control flow information, i.e., the control flow arcs for functional calls cannot be determined statically. His solution is therefore to perform control flow analysis (CFA) as a pre-requisite for other, more problem oriented, data flow analyses. Shivers' CFA employs a technique akin to the call-strings approach which handles higher order functions by embedding call-strings in closures.

Since closures are just a special kind of data-structures, this coincides somewhat with another line of research in abstract interpretation to compute properties of data-structures (e.g. [17]). A common characteristic of these methods is that the semantics of the program is first instrumented by adding control information (e.g., labels), according to which the abstraction is then defined.

1.2 Our Method

The flow analysis presented in this paper is based on a simple observation: given a functional program, its state during execution is the runtime environment, consisting of a tree of frames, with one frame per function invocation. Thus the abstract environment computed by the flow analysis can be bounded by first partitioning frames according to fixed patterns in the call tree, and then by merging frames according to their respective partitions. The static partition is just the first step towards utilizing control flow information embedded in the program, which also enables us to present the analysis as an equational system.

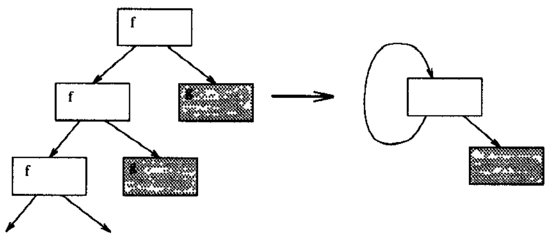

Figure 1: An example showing a simple 2-partition that merges frames according to its function, in this case f and g.

From the equational point of view, if we identify locations in frames by flow variables, then variables are related by a system of equations, which are monotonic since these locations are not updated. The only problem is that the system grows dynamically without bound as computation progresses. Fortunately, the fact that frames are merged also translates into a fixed number of flow variables, thus a finite, static system of equations.

We start with a functional kernel language, and define a rewriting system based semantics which makes the runtime environment explicit by using labels instead of lexical scoping and generates a flat representation of the history of the computation. This representation allows us to relate the runtime environment to the system of flow equations via a partition induced by a equivalence relation expressed on the labels, thus enabling a simple correctness proof.

The theoretical contribution of this paper is the presentation of a general framework for flow analysis in higher order functional settings together with a simple proof of correctness. Although the idea of partitioning program runtime environment has appeared in previous works, we feel that it has yet to be presented in a pure form together with the handling of higher order functions. Data-structures are not considered in this paper but we believe an extension to include them should not be difficult. On the other hand, we do not consider partitioning utilizing semantic information as in [5, 6], in this respect we are closer in spirit to control flow analysis. Most of the known works on control flow analysis are presented in terms of special cases corresponding to very simple partitions, e.g. 0CFA (mono-variant analysis) [2, 27, 30, 32], or 1CFA [30]. Recently [11] presented a polyvariant analysis which detailed a 1CFA-like approach.

The practical significance of our method lies in the fact that, unlike most previous works, it is presented in a simple equational framework. Since the equations distribute the flow constraints over the source program, the implementation of the method is more amenable to various optimizations that exploit data dependencies among the equations (see [14]), thus avoiding the repetitive work to compute unchanged quantities. Efficient implementations can be developed using an attribute system together with incremental computations (see [9]), which bridge the gap between theoretical methods and practical applications. The closest to our presentation style is a recent work ([32]) by Wand and Steckler which computes 0CFA by finding the minimum solution of a set of constraints generated from the parsed tree of the analyzed expression.

The rest of the paper is organized as follows: section 2 defines a simple functional language, whose operational semantics is detailed in section 3. In section 4, its static semantics is defined. Section 5 develops the equations for general flow analysis whose proof of soundness can be found in [31]. Section 6 discusses practical considerations for the application of our method in order to control the accuracy and the cost of the analysis. The paper closes with conclusions and directions for future work.

2 A Functional Kernel Language

We shall present the analysis on a small functional kernel language (FKL), which can be regarded as a textual representation of the graph of lambda expressions[1]. In FKL, all operators are labeled (a, b, c, ...); All lambda expressions are given a name ($f, g, h, \ldots$) and lifted to the top level. A program in FKL is a set of function definitions plus a main expression (figure 2). Within a function definition, x is the (only) formal variable, and $y_1, \ldots, y_l$ are the free variables of the body e. An expression is a set of bindings $\{s_1, \ldots, s_n\}$

[1] Alternatively, one can start from a typical minimal function language, e.g. $e ::= k \mid x \mid e_0\ e_1 \mid \lambda x\ e$, and then define a compilation function into FKL as in [31]

with a designated result label a. A binding s gives a label to each primitive operation in the program.

$$
\begin{array}{lcl}
P & ::= & e \text{ where } \{F_1, \ldots, F_r\} \\
F & ::= & f = \text{lambda}\,(x \mid y_1 \ldots y_l)\; e \\
e & ::= & {}^{a}\{\, s_1;\, \ldots;\, s_n \,\} \\
s & ::= & a = k \\
 & \mid & c = \text{apply } a\ b \\
 & \mid & a = \langle \text{closure } f,\ y_1 \ldots y_l \rangle
\end{array}
$$

Figure 2: Syntax of FKL

Example 1 In the following we show a FKL program (a) together with its equivalent in sugared lambda calculus (b). The λ-expression can be converted to the FKL program by creating a binding for each label, replacing λ's by closures and moving λ-definitions to the top level.

$$
\begin{array}{l}
{}^{a_1}\{a_1 = \text{apply } c_1\ c_3; \\
\quad c_1 = \langle \text{closure gf}, \rangle; \\
\quad c_3 = \langle \text{closure xf}, \rangle\} \\
\text{where } \{\text{gf} = \text{lambda}(g \mid)^{a_2}\{a_2 = \text{apply } a_3\ a_4; \\
\qquad\qquad\qquad\qquad\qquad a_3 = \text{apply } g\ c_2; \\
\qquad\qquad\qquad\qquad\qquad c_2 = \langle \text{closure yf}, \rangle; \\
\qquad\qquad\qquad\qquad\qquad a_4 = \text{apply } g\ v_1; \\
\qquad\qquad\qquad\qquad\qquad v_1 = 0\ \} \\
\qquad \text{xf} = \text{lambda}(x \mid)^{x}\{\}; \\
\qquad \text{yf} = \text{lambda}(y \mid)^{y}\{\}\ \}
\end{array}
$$

(a)

$$
\begin{array}{l}
\text{letrec } c_1 = \lambda^{\text{gf}} g.((g\ c_2)^{a_3}(g\ 0^{v_1})^{a_4})^{a_2}; \\
\qquad c_2 = \lambda^{\text{yf}} y.y; \\
\qquad c_3 = \lambda^{\text{xf}} x.x \\
\quad \text{in } (c_1\ c_3)^{a_1}
\end{array}
$$

(b)

Notation We define some operators that will become useful shortly. For each binding c = apply a b, let op$(c) = a$ and arg$(c) = b$. For each function $f = \text{lambda}(x \mid \ldots)^{a}\{\ldots\}$, let res$(f) = a$.

3 Dynamic Semantics

We shall model the runtime behavior of FKL programs as a rewriting system similar to that introduced in [3], except that we use labels instead of lexical scoping to deal with expansion of function calls. First, we need to extend the basic syntax introduced in figure 2 to handle *dynamic programs* as the intermediate states during the rewriting process. The new syntax is shown in figure 3.

$$
\begin{array}{lcl}
B & ::= & {}^{\alpha}\{\, s_1;\, \ldots;\, s_n \,\} \\
s & ::= & \alpha = k \\
 & \mid & \beta = \langle \text{closure } f,\ \alpha_1 \ldots \alpha_l \rangle \\
 & \mid & \gamma = \text{apply } \alpha\ \beta \\
 & \mid & \beta = \alpha
\end{array}
$$

Figure 3: Syntax of dynamic programs

A dynamic label has the form $\alpha.a$, where α is a unique scope identifier, and a is a static label within that scope. In other words, α is just the activation frame of some function invocation, and a a slot within that frame. α can be conveniently generated as the call-string of that function, i.e., a string of static labels $a_1 \ldots a_l$ representing the sequence of function calls leading to that particular frame. Dynamic labels are thus strings of static labels.

Let α, β, γ denote dynamic labels. Note that given a program $P ::= B$ where $\{F_1;\, \ldots;\, F_r\}$, only the top level expression B changes during rewriting, thus we shall take the dynamic program to be just the toplevel bindings B, leaving the program source code in the background. As shown in figure 3, a dynamic program is a flat set of *dynamic bindings*. The syntax for dynamic bindings is similar to that of static bindings in which static labels have been replaced by dynamic labels.

A binding of the form γ = apply α β is called an *apply binding*, where γ will be called a *call-site* of f if α is bound to a closure generated for function f. A binding of the form $\alpha = v$, where v is either a constant or a closure, will be called a *value binding*. Binding of the form $\beta = \alpha$ are absent from the source programs. These are called *copy bindings* and are needed just for sending arguments and receiving result in the expansion of function calls.

Given an initial program B_0, a dynamic execution of B_0 is any rewriting sequence

$$B_0 \longrightarrow \ldots \longrightarrow B_n \longrightarrow \ldots,$$

where the individual rewriting steps in the sequence are prescribed by the conditional rewrite rules shown in figure 4.

APPLY

$$
\frac{\begin{array}{c}\alpha = \langle \text{closure } f,\ \beta_1 \ldots \beta_l \rangle, \\ f = \text{lambda}\,(x \mid y_1 \ldots y_l)\,{}^{c}\{\, s_1;\, \ldots;\, s_r \,\}\end{array}}{\begin{array}{lcl}\gamma = \text{apply } \alpha\ \beta & \longrightarrow & \gamma.x = \beta; \\ & & \gamma.y_1 = \beta_1;\ \ldots;\ \gamma.y_l = \beta_l; \\ & & s'_1;\ \ldots;\ s'_r; \\ & & \gamma = \gamma.c\end{array}}
$$

PROPAGATE

$$
\frac{\alpha = v}{\beta = \alpha \longrightarrow \beta = v}
$$

Figure 4: Dynamic semantics of FKL.

In this rewriting system, a redex for a dynamic program B is either an apply or a copy binding. Before rewriting can occur, however, the conditions specified on the rule's numerator must be satisfied. These conditions are simply patterns for bindings and they are satisfied if they can be matched against some bindings in the program. Whenever the match is successful, the redex binding can be rewritten according to the denominator of the rewriting rule.

There are in fact just two rewriting rules. Rule APPLY describes what happens during a function application: a copy of the functions body is pasted in, along with the necessary copy bindings for sending argument, free variable values, and for returning result. In order for the scoping to be correct, static labels in the function's body are replaced by dynamic labels appropriate for the current invocation. This is defined by the following:

$$
s'_i \equiv \begin{cases} \gamma.a = k, & \text{if } s_i \equiv a = k \\ \gamma.c = \text{apply } \gamma.a\ \gamma.b, & \text{if } s_i \equiv c = \text{apply } a\ b \\ \gamma.a = \langle \text{closure } f,\ \gamma.y_1 \ldots \gamma.y_l \rangle, & \\ \qquad\qquad \text{if } s_i \equiv a = \langle \text{closure } f,\ y_1 \ldots y_l \rangle & \end{cases}
$$

Rule PROPAGATE, on the other hand, serves to propagate values along the copy bindings, simulating the effect of sending arguments and returning results.

Example 2 The following shows a possible rewriting sequence for the program in example 1. For clarity, the redexes and their matching conditions are shown underlined; the newly introduces bindings are marked.

$$
\begin{array}{ll}
{}^{a_1}\{\ \underline{a1 = \text{apply } c_1\ c_3;} & \longrightarrow \\
\quad \underline{c_1 = \langle\text{closure gf, }\rangle;} & \\
\quad \underline{c_3 = \langle\text{closure xf, }\rangle}\} & \\
{}^{a_1}\{\ \underline{{}^*a_1.g = c_3;} & \longrightarrow \\
\quad {}^*a_1.a_2 = \text{apply } a_1.a_3\ a_1.a_4; & \\
\quad {}^*a_1.a_3 = \text{apply } a_1.g\ a_1.c_2; & \\
\quad {}^*a_1.c_2 = \langle\text{closure yf, }\rangle; & \\
\quad {}^*a_1.a_4 = \text{apply } a_1.g\ a_1.v_1; & \\
\quad {}^*a_1.v_1 = 0; & \\
\quad {}^*a_1 = a_1.a_2; & \\
\quad c_1 = \langle\text{closure gf, }\rangle; & \\
\quad \underline{c_3 = \langle\text{closure xf, }\rangle}\} & \\
{}^{a_1}\{\ \underline{{}^*a_1.g = \langle\text{closure xf, }\rangle;} & \longrightarrow \ldots \\
\quad a_1.a_2 = \text{apply } a_1.a_3\ a_1.a_4; & \\
\quad \underline{a_1.a_3 = \text{apply } a_1.g\ a_1.c_2;} & \\
\quad a_1.c_2 = \langle\text{closure yf, }\rangle; & \\
\quad a_1.a_4 = \text{apply } a_1.g\ a_1.v_1; & \\
\quad a_1.v_1 = 0; & \\
\quad a_1 = a_1.a_2; & \\
\quad c_1 = \langle\text{closure gf, }\rangle; & \\
\quad c_3 = \langle\text{closure xf, }\rangle\} &
\end{array}
$$

In order to define the result of computations, we follow a commonly used technique to define a partial information content ordering on dynamic programs. Let D be the set of primitive constants. Given a program P, let LABELS be the set of static labels occurring in P. We shall assume static labels to be unique within a program. LABELS* then denote the set of dynamic labels.

Each dynamic program B can be seen as defining a mapping $\underline{B}$: LABELS* $\longrightarrow V$. The value domain V is defined by

$$V = D + \text{CLOSURES},$$
$$\text{CLOSURES} = \{\langle\text{closure } f,\ \alpha_1\ \ldots\ \alpha_l\rangle \mid \alpha_i \in \text{LABELS}^*\}$$

where f ranges over function symbols in the program. Intuitively, we can regard LABELS* $\longrightarrow V$ to be a store indexed by the dynamic labels, and each dynamic program B as a state of the store during computation. Moreover, the store is partitioned into frames, where each dynamic label $\alpha.a$ identifies a frame location. With this analogy in mind, we now define:

$$\underline{B}(\alpha) = \begin{cases} v & \text{if } \alpha = v \in B \text{ and } v \text{ is a value} \\ \perp & \text{otherwise} \end{cases}$$

and

$$\underline{B} \sqsubseteq \underline{B}' \quad \text{if} \quad \forall\alpha \in \text{LABELS}^*\ \underline{B}(\alpha) \sqsubseteq \underline{B}'(\alpha).$$

where $x \sqsubseteq y$, $x, y \in V$, if $x = \perp$ or $x = y$.

If $B \longrightarrow B'$, then clearly $\underline{B} \sqsubseteq \underline{B}'$. Thus the progress of computation only increases the information content in the store. It is also the case that all intermediate computation states form a directed set according to this partial ordering (this is essentially the confluence property of FKL, but we shall not be concerned with proving the correctness of this statement, which has little effect on the analysis presented later in this paper), thus a limit state can be meaningfully defined. Let

$$\text{Comp}(B) = \bigsqcup\{\underline{B}' \mid B \longrightarrow^* B'\}.$$

We shall call Comp(B) the *computation function* of B.

4 Static Semantics

Abstract analysis of a program will seldom be useful unless it is performed for all possible runs of the program w.r.t. a set of inputs. In other words, the source program P under analysis will usually have free variables which we assume to take non-closure values. The static semantics defines the collective result of running P on all possible bindings of the free variables.

Let $z_1, \ldots, z_m$ be the free variables of B, where each z_i is constrained to take inputs from $V_i \subseteq V$ (notation $z_i : V_i$). Let

$$2^V = 2^{D+\text{CLOSURES}} = 2^D \times 2^{\text{CLOSURES}}$$

be the power set of values. The static semantics is then defined by the static computation function $\widehat{\text{Comp}}(B)$ where, for all dynamic labels α we have:

$$\widehat{\text{Comp}}(B)(\alpha) = \bigcup\{\text{Comp}(B[v_1/z_1, \ldots, v_m/z_m])(\alpha) | v_i \in V_i\}$$

The static semantics provides an exact characterization of the runtime environments of B, but is almost always infinite, thus not computable in a practical sense. Using flow analysis, we wish to obtain a finite approximation to $\widehat{\text{Comp}}(B)$ that is computable at compile time.

5 General Flow Analysis

Let $\Pi = \{\sigma_1, \ldots, \sigma_s\}$ be a partition of LABELS*. Let $[\alpha]$ denote the equivalence class of α derived from the partition, i.e., the σ s.t. $\alpha \in \sigma$. We require the partition to generate a right congruence relation ([19]), i.e. to satisfy the following condition

$$\forall\, \alpha, \beta, c \quad [\alpha] = [\beta] \Longrightarrow [\alpha.c] = [\beta.c].$$

Define succ : $\Pi \times$ LABELS $\longrightarrow \Pi$ s.t. succ$([\alpha], c) = [\alpha.c]$. The previous condition guarantees succ to be well-defined.

Let $\sigma \in \Pi$. Intuitively, σ is an abstract frame. The entity $\sigma.c$, where c is a static label, is then a cell in that frame. Moreover, if c = apply $a\ b$ is a binding in the source program, then $\sigma.c$ is an abstract call-site for the functions that become bound at $\sigma.a$. Furthermore, succ(σ, c) represents the frame of the callee, which receives its input from, and sends its result to, the caller frames. Note that there could be multiple caller frames. This situation is different from the dynamic semantics, where each activation has a unique caller. In other words, the abstract frames form a graph rather than a tree, the graph may even be cyclic.

Example 3 In the following we show two practical partitions.

1. Let Π_0 = {LABELS*}. This partition corresponding to the 0CFA analysis as described in [30].

2. Let $\Pi_1 = \{\{\epsilon\}\} \cup \{[a] \mid a \in \textsc{Labels}\}$, where ϵ is the empty string, and $[a]$ is the set of labels in $\textsc{Labels}^*$ that end in a. Clearly, $\text{succ}(\sigma, c) = [c]$, thus the partition is well-formed. The corresponding analysis is equivalent to 1CFA as described in [30].

5.1 Abstract Domains

For the general flow analysis, abstract values are constructed from the partition of frames. Let Π be such a partition, the abstract domain for closures is define as:

$$\overline{\textsc{Closures}} = 2^{\{f_\sigma \mid \sigma \in \Pi\}}.$$

The corresponding abstract domain for values is

$$\overline{V} = \overline{D} \times \overline{\textsc{Closures}},$$

where $\overline{D}$ is some abstract domain of D, assumed to be a finite lattice. Clearly, $\overline{V}$ is also a finite lattice with the natural ordering induced from those of its two components.

We note that the above representation for abstract closures, namely f_σ, is really an abbreviation of the abstract closure $\langle \text{closure} f, \sigma.y_1 \ldots \sigma.y_n \rangle$. Such an abstraction for closures means that closures are only distinguished from where they occur in the program's call-tree, rather than by their semantic identity.

For the abstraction to be meaningful, we need to relate elements of $\overline{V}$ to those of 2^V, i.e., abstract values are meant to approximate sets of concrete values. This amounts to establishing a Galois connection [12] between the two domains, i.e., a pair of order preserving maps $\text{Abs} : 2^V \longrightarrow \overline{V}$ and $\text{Conc} : \overline{V} \longrightarrow 2^V$ s.t.

$$\text{Conc}(\text{Abs}(x)) \subseteq x, \quad \textit{and} \quad \text{Abs}(\text{Conc}(\overline{v})) = \overline{v} \qquad (1)$$

In our case, assume $\overline{D}$ is given with a predefined Galois connection (with 2^D), and let $\text{abs} : V \longrightarrow \overline{V}$ s.t.

$$\begin{aligned} \text{abs}(k) &= \langle \text{abs}_D(k), \emptyset \rangle = \langle \overline{k}, \emptyset \rangle, \\ \text{abs}(\langle \text{closure} f,\ \gamma.y_1 \ \ldots\ \gamma.y_l \rangle) &= \langle \bot_{\overline{D}}, \{f_{[\gamma]}\} \rangle. \end{aligned}$$

Then Abs and Conc can be defined as

$$\text{Abs}(x) = \bigsqcup_{v \in x} \text{abs}(v), \quad \text{Conc}(\overline{v}) = \{v \mid \text{abs}(v) \sqsubseteq \overline{v}\}.$$

It is straightforward to verify that condition (1) is satisfied.

5.2 General Flow Equations

Given source program P and a partition Π, Let Φ: $\Pi \times \textsc{Labels} \longrightarrow \overline{V}$ be a vector of flow variables (we shall write $\Phi_{\sigma.a}$ instead of $\Phi(\sigma, a)$ for the vector's elements). The system E of general flow equations is derived as follows:

- For each constant binding $a = k$, include the equations
$$\Phi_{\sigma\ a} = \langle \overline{k}, \emptyset \rangle; \qquad (2)$$
- For each closure binding $a = \langle \text{closure} f, \ldots \rangle$, include the equations
$$\Phi_{\sigma.a} = \langle \bot, \{f_\sigma\} \rangle; \qquad (3)$$
- For each free variable $z : U$, include the equations
$$\Phi_{\sigma.z} = \text{Abs}(U); \qquad (4)$$
- For each function definition $f = \text{lambda}(x \mid y'_1 \ldots y'_l)e$, include the equations
$$\begin{aligned} \Phi_{\sigma.x} = \bigsqcup\{\Phi_{\sigma'.\text{arg}(c)} \mid\ & \text{succ}(\sigma', c) = \sigma, \\ & \langle \bot, \{f_\delta\} \rangle \sqsubseteq \Phi_{\sigma'\ \text{op}(c)}\}, \qquad (5) \\ \Phi_{\sigma.y'_i} = \bigsqcup\{\Phi_{\delta.y_i} \mid\ & \text{succ}(\sigma', c) = \sigma, \\ & \langle \bot, \{f_\delta\} \rangle \sqsubseteq \Phi_{\sigma'\ \text{op}(c)}\}; \qquad (6) \end{aligned}$$
- For each application $c = \text{apply}\ a\ b$, include the equations
$$\Phi_{\sigma.c} = \bigsqcup\{\Phi_{\text{succ}(\sigma,c).\text{res}(f)} \mid \langle \bot, \{f_\delta\} \rangle \sqsubseteq \Phi_{\sigma.a}\}. \qquad (7)$$

Of the previous equations, (2) - (4) are relatively obvious. Equations (5) and (6) reflect how the flow of formal and free variables are merged. In particular, the formal variable $\sigma.x$ gets its flow from the arguments at all call-sites of the function (5), and a free variable $\sigma.y'_i$ gets its flow from inside the closures (6), the latter being due to the fact that free variables are bound where the closures are generated. Equation (7) means that the result of an application includes all the results of the functions called at that site.

Let us rewrite the system of equations E as $\Phi = F(\Phi)$, where, clearly, F is a monotonic operator. Define

$$\Phi^0 = (\bot, \ldots, \bot), \quad \text{and} \quad \Phi^n = F(\Phi^{n-1}), \quad n = 1, 2, \ldots,$$

then the system of equations has a least solution $\Phi^\infty = \bigsqcup_{n=0}^{\infty} \Phi^n$. Assuming the abstract domains to be finite[2], this limit can be reached at some finite number of iterations.

5.3 Correctness of Flow Analysis

The abstraction of $\widehat{\text{Comp}}$ is done in two dimensions. In one, the power set of the value domain 2^V is abstracted to a domain ($\overline{V}$), as described in section 5.1; In the other, dynamic labels are abstracted into a set of abstract frame locations, represented by $\Pi \times \textsc{Labels}$. Given some static semantics $\widehat{B} : \textsc{Labels}^* \longrightarrow 2^V$, its corresponding abstraction can be defined as $\text{ABS}(\widehat{B}) : \Pi \times \textsc{Labels} \longrightarrow \overline{V}$ s.t.

$$\text{ABS}(\widehat{B})(\sigma, a) = \bigsqcup_{[\alpha]=\sigma} \text{Abs}(\widehat{B}(\alpha.a)).$$

As we mentioned before, the correctness criteria for general flow analysis is that Φ^∞ should approximate the static semantics $\widehat{\text{Comp}}(B)$, which can be stated as the following:

Theorem 1 (Correctness of General Flow Analysis) *Let $P \equiv B$ where $\ldots$ be a program with free variables $z_1, \ldots, z_s$, $\Phi = F(\Phi)$ be the system of general flow analysis equations associated with P. The general flow analysis is sound, i.e.*

$$\text{ABS}(\widehat{\text{Comp}}(B))(\sigma, a) \sqsubseteq \Phi^\infty(\sigma, a)$$

[2]This restriction is not strictly necessary, as convergence can be guaranteed using techniques of widening [12, 7] even when the abstract domain is infinite.

$$\begin{tikzcd}[row sep=small]
B_{i-1} \arrow[d, "\text{APPLY}"'] & \equiv \{\ldots;\alpha.a = \langle\text{closure } f,\ \delta.y_1 \ \ldots\ \delta.y_l\rangle;\ldots;\underline{\alpha.c = \text{apply}\quad \alpha.a\ \alpha.b};\ldots\} \\
B_i \arrow[d] & \equiv \{\ldots;\gamma.d=\beta;\ldots\} \\
\vdots \arrow[d] & \\
B_{n-1} \arrow[d, "\text{PROPAGATE}"'] & \equiv \{\ldots;\beta = v;\ldots;\underline{\gamma.d=\beta};\ldots\} \\
B_n & \equiv \{\ldots;\gamma.d=v;\ldots\}
\end{tikzcd}$$

Figure 5: This diagram traces the history of the value binding to an original copy binding introduced via APPLY.

Proof Using the definitions of ABS, Abs and $\widehat{\text{Comp}}$ we obtain the following derivation:

$$\text{ABS}(\widehat{\text{Comp}}(B))(\sigma,a) = \bigcup_{[\alpha]=\sigma} \text{Abs}(\widehat{\text{Comp}}(B)(\alpha.a))$$

$$= \bigcup_{[\alpha]=\sigma}\ \bigcup_{v_i\in V_i} \text{abs}(\text{Comp}(B[v_1/z_1,\ldots,v_m/z_m])(\alpha.a))$$

Let $B_0 = B \cup \{z_1 = k_1;\ldots;z_s = k_s\}$ be any closed instance of B and let $B_0 \xrightarrow{n} B_n$. Then, by the definition of Comp, it is sufficient to show that $abs(\underline{B_n}(\alpha,a)) \subseteq \Phi^{n+1}_{[\alpha].a}$ for all n, α and a, as depicted in the following diagram:

$$\begin{tikzcd}
\Phi^1 \arrow[r, "F"] \arrow[d, dash] & \cdots \arrow[r, "F"] & \Phi^{n+1} \arrow[r, "F"] \arrow[d, dash] & \cdots \\
B_0 \arrow[r] & \cdots \arrow[r] & B_n \arrow[r] & \cdots
\end{tikzcd}$$

In other words, the soundness for the general flow equations follows if, for all γ and d:

$$\gamma.d = v \in B_n \implies \text{abs}(v) \subseteq \Phi^{n+1}_{[\gamma].d}. \qquad (8)$$

If condition (8) is satisfied, we say Φ^{n+1} is safe for B_n. To prove the condition we proceed by induction on n.

- (Base case) Φ^1 is safe for B_0: this is straightforward since the only value bindings in B_0 are those in the source program.

- (Induction step) Assuming that condition (8) holds for integers $< n$, we show that it also holds for n. To this end let

$$B_0 \longrightarrow \ldots \longrightarrow B_{n-1} \longrightarrow B_n.$$

We shall only be interested in value bindings created by the last rewriting step, since the other ones are already covered by the induction hypothesis. There are two cases to consider:

1. $\gamma.d = v$ is introduced by application of the APPLY rule. In this case, $d = v$ has to be in the original program, thus $\gamma.d = v$ is included in $\Phi^1 \subseteq \Phi^{n+1}$.

2. $\gamma.d = v$ is introduced by application of the PROPAGATE rule. In this case, $\gamma.d = v$ must be rewritten from redex $\gamma.d = \beta$ and there is some application of the APPLY rule that introduced the latter copy binding. Assume the copy binding is introduced in rewriting step i, where the function involved is defined as

$$f = \text{lambda}\ (x \mid y'_1\ \ldots\ y'_l)\ {}^e\{\ldots\}.$$

The situation is somewhat complicated, and we illustrate it in figure 5.

The copy binding $\gamma.d = \beta$ can be one of the following three kind, serving different roles. These are treated in turn.

(a) (sending argument) $\gamma \equiv \alpha.c$, $d \equiv x$ and $\beta \equiv \alpha.b$. By induction hypothesis, Φ^n is safe for B^{n-1}, thus for all B_i, $i < n$. We have

$$f_{[\delta]} \in \Phi^n_{[\alpha].a} \quad\text{and}\quad \text{abs}(v) \subseteq \Phi^n_{[\alpha].b}, \qquad (9)$$

i.e., both the closure and the value bindings are included in Φ^n.

Since $\text{succ}([\alpha],c) = [\gamma]$, $a = \text{op}(c)$ and $b = \text{arg}(c)$, from equation (G4.a) and (9) we derive

$$\Phi^{n+1}_{[\gamma].d} = \Phi^{n+1}_{[\alpha\ c]\ x} \supseteq \Phi^n_{[\alpha].b} \supseteq \text{abs}(v).$$

Therefore $\gamma.d = v$ is included in Φ^{n+1}.

(b) (passing free variable value) $\gamma \equiv \alpha.c$, $d \equiv y'_i$ and $\beta \equiv \delta.y_i$. By the induction hypothesis,

$$\text{abs}(v) \subseteq \Phi^n_{[\delta].y_i}. \qquad (10)$$

This case can be verified in the same way as for the previous one, from equations (G4.b) and (10).

(c) (returning result) $\gamma.d \equiv \alpha.c$ and $\beta = \alpha.c.e$. By the induction hypothesis,

$$\text{abs}(v) \subseteq \Phi^n_{[\alpha\ c].e}. \qquad (11)$$

Recall that $[\alpha.c] = \text{succ}([\alpha],c)$ and $e = \text{res}(f)$. In view of equation (G5), (11) becomes

$$\text{abs}(v) \subseteq \Phi^n_{\text{succ}([\alpha],c).\text{res}(f)} \subseteq \Phi^{n+1}_{[\alpha].c} = \Phi^{n+1}_{[\gamma].d}.$$

Thus $\gamma.d = v$ is included in Φ^{n+1}.

From (a), (b) and (c), we conclude that Φ^{n+1} is safe for B_n, for all integer n. □

Note, however, that the correctness condition is a "one-way" assertion: it says nothing about how precise the analysis is. In particular, the trivial analysis that assigns the largest possible value to every σ and a is correct, but is otherwise quite useless. Unfortunately, it is extremely hard, if ever possible, to characterize the precision of the analysis in any quantitative way. In practice, however, this does not seem to be a serious problem.

Example 4 Consider the expression in example 1 and its associated FKL program. The evaluation of this expression, i.e. the concrete flow Φ_{a_1}, is 0.

Suppose that we choose the 0CFA analysis in which integer constants are abstracted to token *int*, a typical approach in current systems. Then using the partition Π_0 in example 3.1 we generate the equations shown in figure 6.

$$\begin{array}{lcl}
\Phi_{v_1} & = & \langle \text{int}, \emptyset \rangle \\
\Phi_{c_1} & = & \langle \perp, \{gf\} \rangle \\
\Phi_{c_2} & = & \langle \perp, \{yf\} \rangle \\
\Phi_{c_3} & = & \langle \perp, \{xf\} \rangle \\
\Phi_{g} & = & \bigcup_{\imath} \{\Phi_{arg(a_\imath)} \mid \langle \perp, \{gf\} \rangle \subseteq \Phi_{op(a_\imath)}\} \\
\Phi_{y} & = & \bigcup_{\imath} \{\Phi_{arg(a_\imath)} \mid \langle \perp, \{yf\} \rangle \subseteq \Phi_{op(a_\imath)}\} \\
\Phi_{x} & = & \bigcup_{\imath} \{\Phi_{arg(a_\imath)} \mid \langle \perp, \{xf\} \rangle \subseteq \Phi_{op(a_\imath)}\} \\
\Phi_{a_\imath} & = & \bigcup \{\Phi_{res(f)}, \mid \langle \perp, \{f\} \rangle \subseteq \Phi_{op(a_\imath)}, f \in \{\text{gf}, \text{xf}, \text{yf}\}\}
\end{array}$$

Figure 6: 0CFA equations. In the equations above, $\imath = 1, 2, 3, 4$.

Other than $\Phi_g = \langle \perp, \{\text{xf}\} \rangle$, all the non-immediate flows in the least fixpoint solution of this system are equal to $\langle \{\text{int}, \{\text{yf}\} \rangle$. Thus, since $\Phi_{a_1} = \langle \{\text{int}, \{\text{yf}\} \rangle$, the 0CFA analysis produced flows which are strictly more conservative than the exact ones.

Choosing to use more resources, we try next the 1CFA analysis (see example 3.2) which distinguishes between calls of a function at different sites. We derive the equations shown in figure 7.

$$\begin{array}{lcl}
\Phi_{\sigma\ v_1} & = & \langle \text{int}, \emptyset \rangle \\
\Phi_{\sigma\ c_1} & = & \langle \perp, \{\text{gf}_\sigma\} \rangle \\
\Phi_{\sigma.c_2} & = & \langle \perp, \{\text{yf}_\sigma\} \rangle \\
\Phi_{\sigma.c_3} & = & \langle \perp, \{\text{xf}_\sigma\} \rangle \\
\Phi_{[a_\imath].g} & = & \bigcup_{\sigma} \{\Phi_{\sigma.arg(a_\imath)} \mid \langle \perp, \{\text{gf}_\delta\} \rangle \subseteq \Phi_{\sigma.op(a_\imath)}\} \\
\Phi_{[a_\imath].y} & = & \bigcup_{\sigma} \{\Phi_{\sigma.arg(a_\imath)} \mid \langle \perp, \{\text{yf}_\delta\} \rangle \subseteq \Phi_{\sigma.op(a_\imath)}\} \\
\Phi_{[a_\imath].x} & = & \bigcup_{\sigma} \{\Phi_{\sigma.arg(a_\imath)} \mid \langle \perp, \{\text{xf}_\delta\} \rangle \subseteq \Phi_{\sigma\ op(a_\imath)}\} \\
\Phi_{\sigma.a_\imath} & = & \bigcup \{\Phi_{[a_\imath].res(f)} \mid \langle \perp, \{f_\delta\} \rangle \subseteq \Phi_{\sigma.op(a_\imath)}, \\
 & & \qquad f \in \{\text{gf}, \text{xf}, \text{yf}\}\}
\end{array}$$

Figure 7: 1CFA equations where $i = 1, 2, 3, 4$ and σ ranges over all classes of equivalence of partition Π_1

The least fixpoint of this system of equations contains the following nontrivial flows:

$$\Phi_{\sigma\ a_1} = \Phi_{\sigma.a_2} = \Phi_{\sigma.a_4} = \Phi_{[a_2].y} = \Phi_{[a_4].x} = \langle \text{int}, \emptyset \rangle,$$
$$\Phi_{\sigma\ a_3} = \Phi_{[a_3]\ x} = \langle \perp, \{\text{yf}_{[a_1]}\} \rangle, \quad \text{and}$$
$$\Phi_{[a_1]\ g} = \bigcup_{\sigma} \Phi_{\sigma\ c_3}$$

Notice that $\Phi_{a_1} = \langle \text{int}, \emptyset \rangle = \text{abs}(0)$, thus 1CFA analysis performs better than 0CFA, in fact as well as possible.

6 Practical Considerations

In this section we briefly discuss some optimizations that can lead to significant performance increase in practical implementation of the flow analysis as described in the previous section.

6.1 Incremental Computation

Let $m = |\Pi|$ and $l = |\text{LABELS}|$. Recall the definition of the set of general flow equations from the previous section. We derived one equation for each abstract frame location $\sigma.a$, where $\sigma \in \Pi$ and $a \in$ LABELS. Thus the number of flow equations is $m \times l$.

It is possible that most of the equations need never be generated (e.g., equations for $\sigma.c$, where c occurs in function definition for f but σ never becomes a call-site of f), since they are not meaningful flows and do not contribute to the final solution.

However, it is generally not possible to determine which equations are actually needed before the analysis is run. Indeed, the equations corresponding to $\sigma.c$, where c occurs in the body of a function definition for f, are needed only if σ becomes an abstract call-site of f. Thus for the sake of efficiency in implementations of flow analysis, the set of flow equations should be generated incrementally on demand, i.e., when new call-sites are discovered.

Another place where incremental computation is likely to increase performance, is within the iteration of the vector of flow variables $\Phi^\imath$. Since the sequence is monotonic, at stage i we only need to compute the differential from the previous stage $\Phi^\imath - \Phi^{\imath-1}$. In addition, iteration should stop as soon as the differential becomes empty.

Going one step further, one should not naively compute $\Phi^1, \Phi^2, \ldots$ in that order, but should exploit the data-dependency among individual flow variables. In general, it is always better to follow the direction of dataflow, i.e., if $\Phi_{\sigma\ a}$ affects $\Phi_{\sigma'\ b}$, then one should delay the evaluation of the latter until the former has saturated. This is not always possible, since there may be cyclic dependencies. In case there are cyclic dependencies, one should treat a cyclic component of the dependency graph as a big flow variable, and try to delay the evaluation of any flow variable dependent on variables in that component until all variables in the component have reached saturation. Recently([7]), Bourdoncle introduced a data dependency based formal methodology for describing chaotic iteration strategies for systems of fixpoint equations as well as effective approaches for generating such strategies.

6.2 Partitioning of Call Sites

Flow analysis as presented in this paper is inherently syntactic. This is especially so in its handling of functions. We do not compute an abstract map for functions, which encodes all its input/output behavior; rather we distinguish function call sites, and compute a new set of flows corresponding to each abstract function call. The number of different flows we compute are fixed by the size of the partition, thus one may trade-off complexity vs. accuracy of analysis by choosing different partitions.

In general, more accurate flow analysis may be obtained by finer partitioning of call-sites. Consider a program containing a single function f, with potential call-sites illustrated by the following schema (we assume f can be called at any one of the four call-site):

```
ᵉ{ ...;
   a = apply ... ;
   b = apply ... ;
   c = apply ... ;
   ... }
where { f = lambda (x | ) ...; d = apply ... ;... }
```

If we perform the equivalent of 1CFA as defined in example 2, the following partition will be defined, providing 5 abstract frames:

$$\begin{aligned}
&\{\epsilon\},\\
&[a] = \{\text{dynamic labels ending in } a,\}\\
&[b] = \{\text{dynamic labels ending in } b,\}\\
&[c] = \{\text{dynamic labels ending in } c,\}\\
&[d] = \{\text{dynamic labels ending in } d.\}
\end{aligned}$$

It is easy to find out the flow of arguments and results, as illustrated in figure 8 (A) (note: call-site $\sigma.a$ sends argument to frame σ' if $\text{succ}(\sigma, a) = \sigma'$; Likewise $\sigma.a$ receives results from σ').

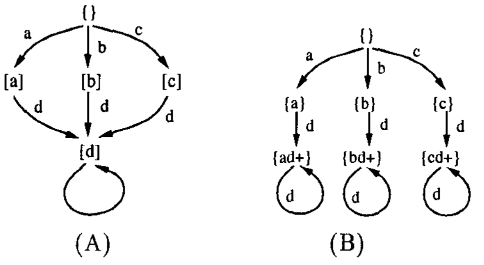

Figure 8: Two sample partitions, where arcs indicate the flow of arguments.

The problem with partition (A) is that too many frames are sending arguments to frame $[d]$. Clearly, a good partition should try to minimize the number of senders for each abstract frame σ, thus less flows get merged. In order to examine the situation in more detail, let us first introduce some notation.

Given a program P, we define its *call-graph* to be a labeled graph. The nodes of the graph are the function symbols in P. There is an arc $f \xrightarrow{a} g$ if f may call g at call-site a. For example, the program we are considering has the call graph shown in figure 9.

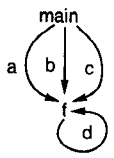

Figure 9: A sample call graph, showing one function f that can be called at call-sites a, b and c within the main expression, with another call-site d within its body.

Clearly, the set of dynamic frames is only a subset of the paths generated from the call-graph. The call-graph provides us with clues as to how to obtain a better partition the set of dynamic frames into a finite set of abstract frames. For example, partition (B) shown in figure 8 has the following abstract frames (we represent each equivalent class in the partition as a regular expression, where others contains the rest of the call-strings):

$$\{\epsilon, a, b, c, ad^+, bd^+, cd^+, others\}$$

which has the potential of providing a finer analysis.

As we pointed out earlier, it is generally not possible to obtain an accurate call-graph before any analysis is performed. Thus the initial call-graph has to be conservative and assume the worst: essentially any call-site where the operator is unknown can be the call-site of any function. After the analysis, however, we will generally have a much better approximation of the call-graph. In view of this, flow analysis can be iterated to achieve better precision. Taking efficiency into consideration, it may be a good idea to perform simple flow analysis first just to obtain a better call-graph, then to perform the general flow analysis tailored for the particular values that one wish to capture.

7 Conclusions and Further Work

We have presented a framework for doing flow analysis for higher-order functions by solving a system of monotonic flow equations. The nature of the abstraction process is to define a partition of the program's runtime environment according to fixed patterns in the program's call-tree, and then to merge points belonging to the same class in the partition. This enabled us to present the analysis in the equational framework of [12], while still being able to handle functions as first class values.

The practical motivation behind this work is the need to control the computational resources required for performing flow analysis on functional values, which is known to explode with higher order functions([24]). As shown in the previous section, by choosing appropriate partitions, one can control the accuracy and the cost of the analysis according to the problem to be solved.

The contribution of our paper lies in clarifying the abstraction of program control points, thereby setting the foundation of a general framework for flow analysis. Furthermore, this framework is presented in a simple equational setting, which exposes optimization opportunities for implementation.

Future work can be carried out in three directions: 1) Replacing FKL to a kernel of a true functional language. This primarily means adding data structures. Since functional closure is a special kind of data structure, we expect that they can be handled similarly. 2) Handle imperative language constructs. We believe this is possible by adding more control points, i.e., by indexing flow variables by labels in the program, thus making it possible to define the program state at every control point. 3) Considering the practical aspect of the method. Clearly, the usefulness of any static analysis method can only be validated by efficient implementations for real compiler optimization problems. Implementations of our method are amenable to use of incremental computation, which optimizes the fixpoint calculation in three aspects: generating equations on demand, computing only differentials during each iteration step, following data dependencies among the flow variables([7]) etc. Preliminary work in this area is reported in [9].

References

[1] Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1987.

[2] Anders Bondorf *Automatic Autoprojection of Higher Order Recursive Equations* Proc. ESOP'90, Springer Verlag, 1990

[3] Zena M. Ariola and Arvind *A Syntactic Approach to Program Transformations.* Proc. of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation PEPM'91. SIGPLAN Notices, Vol 26 No. 9, 116–129.

[4] Gebreselassie Baraki. *A Note on Abstract Interpretation of Polymorphic Functions.* 5th ACM Conference on Functional Languages and Computer Architecture. Lecture Notes in Computer Science Vol 523, August 1991.

[5] Francois Bourdoncle *Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity.* Proc. International Workshop PLILP'90. Lecture Notes in Computer Science Vol. 456, Springer-Verlag.

[6] Francois Bourdoncle *Abstract Interpretation by Dynamic Partitioning.* J. Functional Programming 2 (4): 407–435, October 1992.

[7] Francois Bourdoncle *Efficient chaotic iteration strategies with widenings* Proceedings of the International Conference on Formal Methods in Programming and their Applications, Lecture Notes in Computer Science 735, Springer Verlag, 1993

[8] G. Burns and C. Hankin and S. Abramsky. *Strictness Analysis for Higher-Order Functions.* In Science of Computer Programming, 7:249–278, 1986.

[9] T. Cheatham, H. Gao, and D. Stefanescu *A Suite of Analysis Tools Based on a General Purpose Abstract Interpreter,* Proceedings of the International Conference on Compiler Construction, Edinburgh, April 1994

[10] Thomas E. Cheatham, Jr. and Dan Stefanescu *A Suite of Optimizers Based on Abstract Interpretation,* PEPM'92, San Francisco, June 1992

[11] Charles Consel *Polyvariant Binding-Time Analysis For Applicative Languages,* PEPM'93, Copenhagen, June 1993

[12] Patrick Cousot and Radhia Cousot *Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximate fixpoints,* Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pages 238-252, 1977.

[13] Patrick Cousot and Radhia Cousot Static determination of dynamic properties of recursive procedures IFIP Conference on Formal Description of Programming Concepts, vol 1, pages 237-277, Saint Andrews, 1977

[14] Patrick Cousot *Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice,* Rapport de Recherche No. 88, Laboratoire IMAG, Grenoble, Sept. 1977.

[15] Patrick Cousot and Radhia Cousot Systematic design of program analysis frameworks Conference Record of the Sixth ACM Symposium on Principles of Programming Languages, San Antonio, 1979

[16] Patrick Cousot *Semantic foundations of program analysis* In S.S.Muchnick and N.D.Jones, editors, Program Flow Analysis: Theory and Applications, chapter 10, pg. 303-342, Prentice-Hall, 1981

[17] Alain Deutsch. *On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Applications.* Proc. 17th Annual ACM Symp. on Principles of Programming Languages, 157–168, San Francisco, Jan. 1990.

[18] Alain Deutsch. *A Storeless Model of Aliasing and Its Abstractions Using Finite Representations of Right-Regular Equivalence Relations* Proceedings of the 1992 International Conference on Computer Languages, 2–13, Oakland, California, April, 1992.

[19] Samuel Eilenberg *Automata, Languages and Machines,* volume A, Academic Press, 1974

[20] John Hughes. *Abstract Interpretation of First-order Polymorphic Functions.* Technical Report 89/R4, University of Glasgow, Dept. of Computing Science, 1988.

[21] S. Hunt and C. Hankin. *Fixed Points and Frontiers: A New Perspective.* J. of Functional Programming, 1:91–120, 1991.

[22] Neil Jones and Steven Muchnick *A Flexible Approach to Intraprocedural Data Flow Analysis and Programs with Recursive Data Structures,* Conference Record of the Ninth ACM Symposium on Principles of Programming Languages, pages 66-74, 1982.

[23] Neil Jones and Alan Mycroft *Data Flow Analysis of Applicative Programs Using Minimal Functions Graphs: Abridged Version,* Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages, pages 296-306, 1986.

[24] Atty Kanamori and Daniel Weise *An Empirical study of an Abstract Interpretation of Scheme Programs,*Technical Report, Stanford University, April 1992.

[25] T. Koo and P. Mishra. *Strictness Analysis: A New Perspective Based on Type Inference.* Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture, 260–272, 1989.

[26] Alan Mycroft *Abstract Interpretations and Optimizing Transformations for Applicative Programs,* Ph.D. Thesis, University of Edinburgh, Scotland, 1981.

[27] Jens Palsberg and Michael I. Schwartzbach *Binding Time Analysis: Abstract Interpretation versus Type Inference,* Technical Report, Aarhus University, 1992.

[28] M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis.* In: Muchnick and Jones (eds), Program Flow Analysis, Theory and Applications. Prentice-Hall, 189–233, 1981.

[29] Olin Shivers *Control Flow Analysis in Scheme* ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta GA, June 22-24, 1988.

[30] Olin Shivers *Control Flow Analysis of Higher Order Languages or Taming Lambda*, Technical Report CMU-CS-91-145, Carnegie Mellon University, May 1991.

[31] Dan Stefanescu and Yuli Zhou *An Equational Framework for the Abstract Flow Analysis of Functional Programs*, Technical Report, Harvard University, January 1993.

[32] Mitchell Wand and Paul Steckler *Selective and Lightweight Closure Conversion*,Conference Record of the 21st ACM Symposium on Principles of Programming Languages, pages 435-445, 1994.