Equality-Based Flow Analysis versus Recursive Types

JENS PALSBERG
Purdue University

Equality-based control-flow analysis has been studied by Henglein, Bondorf and Jørgensen, DeFouw, Grove, and Chambers, and others. It is faster than the subset-based 0-CFA, but also more approximate. Heintze asserted in 1995 that a program can be safety checked with an equality-based control-flow analysis if and only if it can be typed with recursive types. In this article we falsify Heintze's assertion, and we present a type system equivalent to equality-based control-flow analysis. The new type system contains both recursive types and an unusual notion of subtyping. We have $s \leq t$ if s and t unfold to the same regular tree, and we have $\bot \leq t \leq \top$ where t is a function type. In particular, there is no nontrivial subtyping between function types.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Flow analysis, type systems

1. INTRODUCTION

Control-flow analysis is done to determine approximate sets of functions that may be called from the call sites in a program. In this article we address an instance of the question

> How does flow analysis relate to type systems?

Our focus is on

(1) equality-based control-flow analysis which has been studied by Henglein [1992], Bondorf and Jørgensen [1993], DeFouw et al. [1998], and others, and
(2) recursive types which, for example, are present in a restricted form in Java [Gosling et al. 1996], in the form of recursive interfaces where equality and subtyping are based on names rather than structure.

Equality-based control-flow analysis is a simplification of subset-based control-flow analysis [Heintze and McAllester 1997; Palsberg 1995; Shivers 1991]. We will use the following abbreviations:

—0-CFA$_{\subseteq}$: subset-based control-flow analysis and

Author's address: Department of Computer Science, Purdue University, West Lafayette, IN 47907; email: palsberg@cs.purdue.edu.

—0-CFA$_=$: equality-based control-flow analysis.

0-CFA$_\subseteq$ is also known as, simply, 0-CFA. We can illustrate the difference between 0-CFA$_\subseteq$ and 0-CFA$_=$ by considering how they analyze a call site e_1e_2 in a functional program. Suppose $\lambda x.e$ is a function in that program. We want a flow analysis to express that

> if $\lambda x.e$ becomes the result of evaluating e_1, then flow relations are established (1) between the actual argument e_2 and the formal argument x and (2) between the body e and the call site e_1e_2.

With a subset-based analysis, the flow relations are subset inclusions. This models that values flow from the actual argument to the formal argument, and from the body of the function back to the call site. With an equality-based analysis, the flow relations are equations. Thus, the flow information for the actual and formal argument is forced to be the same, and the flow information for the body and the call site is also forced to be the same. Intuitively, the equations establish a bidirectional flow of information.

0-CFA$_=$ is more approximate than 0-CFA$_\subseteq$. Both have been implemented many times for various purposes. In general, for functional and object-oriented languages, 0-CFA$_\subseteq$ can be executed in cubic time. For programs with finite types, 0-CFA$_\subseteq$ can be executed in quadratic time [Heintze and McAllester 1997], and particular flow-oriented questions such as "identify all functions called from one specific call site" can be answered in linear time [Heintze and McAllester 1997]. For comparison, 0-CFA$_=$ can always be executed in almost-linear time [Henglein 1992]. Which one of 0-CFA$_\subseteq$ and 0-CFA$_=$ is the better choice in practice? For a language like ML [Milner et al. 1990] where functions have finite polymorphic types and where data may have recursive types, experiments by Heintze and McAllester [1997] indicate that it is a good choice to use 0-CFA$_\subseteq$. They implemented a variant of the quadratic-time algorithm for 0-CFA$_\subseteq$ which treated data in a much simplified way. For the problem of pointer analysis, there are algorithms which are close cousins of 0-CFA$_\subseteq$ and 0-CFA$_=$ [Steensgaard 1996]. For this problem, the condition of finite types does not hold in general. Shapiro and Horwitz [1997] presented an experimental comparison of the two algorithms, and they confirm the theoretical conclusion that 0-CFA$_=$ is faster and more approximate than 0-CFA$_\subseteq$. For an object-oriented language like Java, the condition of finite types is seldomly satisfied because of, for example, binary methods [Bruce et al. 1995]. DeFouw et al. [1998] experimentally compared a family of flow-analysis algorithms whose time complexities are at most cubic time. Both 0-CFA$_=$ and some of its variants do well in that comparison. Ashley [1996] has also presented a flow analysis with time complexity less than cubic time. It remains open how it relates to 0-CFA$_=$. Bondorf and Jørgensen [1993] implemented both 0-CFA$_\subseteq$ and 0-CFA$_=$ for Scheme as part of the partial evaluator Similix. For Scheme, the condition of finite types does not hold in general. They concluded that the two analyses have comparable precision for their application and that 0-CFA$_=$ is much faster. In summary, 0-CFA$_=$ has in experiments proved to be a preferable alternative to 0-CFA$_\subseteq$ for many applications.

Flow analyses such as 0-CFA can be formulated using constraints (see for example Palsberg [1995] and Palsberg and Schwartzbach [1994]). This approach proceeds in two steps: (1) derive flow constraints from the program text and (2) compute the

least solution of the constraints. The least solution is the desired flow information. The precision of the analysis stems from the choice of constraints. For example, one choice leads to 0-CFA$_\subseteq$, and another choice leads to 0-CFA$_=$. The kind of flow constraints used, for example, by Palsberg [1995] always admits a least solution.

We can turn a flow analysis into a predicate which accepts and rejects programs, by extending it with safety constraints. For example, for a call site e_1e_2 in a functional program, a safety constraint might express: "does the flow information for e_1 denote only functions?" Safety constraints do not always have a solution. They can be derived from the program text, just like flow constraints. This means that we can do a flow-based safety analysis of a program in two steps: (1) derive flow and safety constraints from the program text and (2) decide if the constraints are satisfiable. Such a safety analysis performs a task akin to type inference, in the sense that "safe" is like "typable."

Palsberg and O'Keefe [1995] showed that a program can be safety checked with 0-CFA$_\subseteq$ if and only if it can be typed in Amadio and Cardelli's type system with subtyping and recursive types [Amadio and Cardelli 1993]. The proof of this connection makes explicit the close relationship between flow and subtyping.

Heintze [1995] asserted that a program can be safety checked with 0-CFA$_=$ if and only if it can be typed with recursive types. This assertion is reasonable because it says that, intuitively, if we replace subset inclusions by equalities, then the need for subtyping disappears. Heintze's assertion is also consistent with the observation that both 0-CFA$_=$ and type inference with recursive types can be executed in almost-linear time. Perhaps surprisingly, Heintze's assertion is false. For example, consider the λ-term:

$$E_1 \;=\; \lambda f.\lambda g.g(f\mathsf{0})(f(\lambda x.x)).$$

The variable f is applied to both the number $\mathsf{0}$ and the function $\lambda x.x$. Thus, the λ-term E_1 does not have a type in a type system with recursive types but no subtyping. Still, a 0-CFA$_=$-based safety analysis accepts this program, by assigning both f and g the empty flow set (see Section 2 for details).

For another example, consider the λ-term

$$E_2 \;=\; (\lambda f.\lambda g.g(f(\lambda a.\mathsf{0}))(f(\lambda b.\lambda x.x)))(\lambda y.\mathsf{0}).$$

It reminds a bit of the previous example, but now f is applied to $(\lambda a.\mathsf{0})$ and $(\lambda b.\lambda x.x)$. Again, the λ-term e_2 does not have a type in a type system with recursive types but no subtyping. For E_2, a conservative flow analysis cannot assign the empty flow set to f because that flow set should at least contain $(\lambda y.\mathsf{0})$. Still, a 0-CFA$_=$-based safety analysis accepts this program, by assigning y a flow set which contains both $(\lambda a.\mathsf{0})$ and $(\lambda b.\lambda x.x)$.

Given that Heintze's assertion is false, we are left with two questions:

(1) which type system corresponds to 0-CFA$_=$ and

(2) which control-flow analysis corresponds to recursive types?

Palsberg and O'Keefe's result [1995] implies that E_1 and E_2 can be typed if we have both recursive types and Amadio/Cardelli subtyping. Their result also seems to indicate that adding both recursive types and all of the Amadio/Cardelli subtyping to match 0-CFA$_=$ would be overkill. Thus, to answer the first question, it makes

sense to ask "how much subtyping is necessary and sufficient to match 0-CFA$_=$?" To answer the second question we must ask "what restrictions on 0-CFA$_=$ must we impose to match recursive types?"

In this article we answer the first question, and we give a partial answer to the second question. We show that a program can be safety checked with 0-CFA$_=$ if and only if it can be typed with recursive types and an unusual restriction of Amadio/Cardelli subtyping. We have $s \leq t$ if s and t unfold to the same regular tree, and we have $\bot \leq t \leq \top$ where t is a function type. In particular, there is no nontrivial subtyping between function types. To see why nontrivial subtyping between function types is not required to match 0-CFA$_=$, consider the program $(\lambda x.e)e'$. Let $\langle x \rangle$ be a flow variable for the binding occurrence of x, and let $[\![(\lambda x.e)e']\!]$, $[\![\lambda x.e]\!]$, $[\![e]\!]$, $[\![e']\!]$ be flow variables for the occurrences $(\lambda x.e)e'$, $\lambda x.e$, e, e', respectively. If φ is a map from flow variables to flow sets, which satisfies the 0-CFA$_=$ constraints, then in particular it satisfies

$$\begin{aligned} \varphi([\![e']\!]) &= \varphi(\langle x \rangle) \\ \varphi([\![e]\!]) &= \varphi([\![(\lambda x.e)e']\!]). \end{aligned}$$

We can also use $\langle x \rangle$, $[\![(\lambda x.e)e']\!]$, $[\![\lambda x.e]\!]$, $[\![e]\!]$, $[\![e']\!]$ as type variables, and for a type system such as simple types where there is no nontrivial subtyping between function types, we get, among others, the following constraints on type correctness:

$$\begin{aligned} [\![\lambda x.e]\!] &= \langle x \rangle \to [\![e]\!] \\ [\![\lambda x.e]\!] &= [\![e']\!] \to [\![(\lambda x.e)e']\!]. \end{aligned}$$

Unification gives that a typing must satisfy the constraints

$$\begin{aligned} [\![e']\!] &= \langle x \rangle \\ [\![e]\!] &= [\![(\lambda x.e)e']\!]. \end{aligned}$$

Thus, we get the same form of relationships between the types as there are between the flow sets. If we allow nontrivial subtyping between function types, then the constraints on type correctness become [Palsberg and O'Keefe 1995]

$$\begin{aligned} [\![\lambda x.e]\!] &\geq \langle x \rangle \to [\![e]\!] \\ [\![\lambda x.e]\!] &\leq [\![e']\!] \to [\![(\lambda x.e)e']\!]. \end{aligned}$$

In particular, this opens the possibility for a nontrivial relationship

$$\langle x \rangle \to [\![e]\!] \leq [\![e']\!] \to [\![(\lambda x.e)e']\!]$$

and hence

$$\begin{aligned} [\![e']\!] &\leq \langle x \rangle \\ [\![e]\!] &\leq [\![(\lambda x.e)e']\!]. \end{aligned}$$

These constraints are closely related to the flow constraints used in 0-CFA$_\subseteq$ [Palsberg and O'Keefe 1995].

We also show that if a program can be safety checked with a certain restriction of 0-CFA$_=$, then it can be typed with recursive types. Our restriction of 0-CFA$_=$ is that all flow sets must be nonempty and consistent. Consistency means that (1)

if two functions $\lambda x.e$ and $\lambda y.e'$ occur in the same flow set then the flow sets for x and y are equal and (2) the flow sets for e and e' are equal.

In slogan-form, our results read

$$\begin{aligned} \textit{0-CFA}_{=} &= \textit{Recursive types} + \textit{A tiny drop of subtyping.} \\ \textit{Recursive types} &\supseteq \textit{0-CFA}_{=} - \textit{Inconsistency} - \textit{Emptiness.} \end{aligned}$$

The key to understanding the second result is that both empty flow sets and flow sets with two or more inconsistent functions have no counterparts in a type system with just recursive types. The restricted version of 0-CFA$_=$ does not fully match recursive types, because a program may have a type for which no flow set exists.

In the next section, we present Heintze's definition of 0-CFA$_=$. In Section 3 we present the new type system, and in Sections 4 and 5 we prove our results. Our example language is a λ-calculus, defined by the grammar

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid 0 \mid \mathsf{succ}\ e,$$

where succ denotes the successor function on integers.

2. EQUALITY-BASED CONTROL-FLOW ANALYSIS

Given a λ-term P, assume that P has been α-converted such that all bound variables are distinct and different from the free variables. Let $\mathsf{Var}(P)$ be the set of λ-bound variables in P. Let X_P be the set of variables consisting of one variable $\langle x \rangle$ for each $x \in \mathsf{Var}(P)$. Let Y_P be a set of variables disjoint from X_P consisting of one variable $[\![e]\!]$ for each occurrence of a subterm e of P. (The notation $[\![e]\!]$ is ambiguous because there may be more than one occurrence of e in P. However, it will always be clear from context which occurrence is meant.) The set $\mathsf{Abs}(P)$ is the set of occurrences of subterms $\lambda x.e$ of P. The set $\mathsf{CL}(P)$ is $\mathit{Powerset}(\mathsf{Abs}(P)) \cup \{\{\mathsf{Int}\}\}$. Flow-based safety analysis of a λ-term P can be phrased in terms of a constraint system over $X_P \cup Y_P$ where the variables range over $\mathsf{CL}(P)$:

—For every occurrence in P of a subterm of the form 0, the constraint

$$[\![0]\!] = \{\mathsf{Int}\};$$

—for every occurrence in P of a subterm of the form succ e, the two constraints

$$\begin{aligned} [\![e]\!] &= \{\mathsf{Int}\} \\ [\![\mathsf{succ}\ e]\!] &= \{\mathsf{Int}\}; \end{aligned}$$

—for every occurrence in P of a subterm of the form $\lambda x.e$, the constraint

$$\{\lambda x.e\} \subseteq [\![\lambda x.e]\!];$$

—for every occurrence in P of a subterm of the form $e_1 e_2$, the constraint

$$[\![e_1]\!] \subseteq \mathsf{Abs}(P);$$

—for every occurrence in P of a λ-variable x, the constraint

$$\langle x \rangle = [\![x]\!];$$

—for every occurrence in P of a subterm of the form $\lambda x.e$, and for every occurrence in P of a subterm of the form $e_1 e_2$, the constraints

$$\{\lambda x.e\} \subseteq [\![e_1]\!] \;\Rightarrow\; [\![e_2]\!] = \langle x \rangle$$
$$\{\lambda x.e\} \subseteq [\![e_1]\!] \;\Rightarrow\; [\![e]\!] = [\![e_1 e_2]\!].$$

The last two constraints create a connection between a call site $e_1 e_2$ and a potential callee $\lambda x.e$. Notice that two of the constraints are *not* equalities, but subset inclusions. This is the key reason why subtyping is needed to match this safety analysis.

This constraint system mixes flow constraints and safety constraints. The safety constraints are

—for $\mathsf{succ}\ e$: $\quad [\![\mathsf{succ}\ e]\!] \subseteq \{\mathsf{Int}\}$ and
—for $e_1 e_2$: $\quad [\![e_1]\!] \subseteq \mathsf{Abs}(P)$,

and the rest are flow constraints. Notice that because Int and functions cannot occur in the same flow set we have that a constraint such as $[\![\mathsf{0}]\!] = \{\mathsf{Int}\}$ has the same effect as $[\![\mathsf{0}]\!] \supseteq \{\mathsf{Int}\}$.

Denote by $C(P)$ the system of constraints generated from P in this fashion. Let $\mathsf{Cmap}(P)$ be the set of total functions from $X_P \cup Y_P$ to $\mathsf{CL}(P)$. A function $\varphi \in \mathsf{Cmap}(P)$ satisfies $C(P)$ if it satisfies all constraints in $C(P)$. We say that P is 0-CFA$_=$ safe if $C(P)$ is satisfiable.

For example, consider again

$$E_1 \;=\; \lambda f.\lambda g.g(f_1\mathsf{0})(f_2(\lambda x.x)),$$

where we have labeled the two occurrences of f as f_1 and f_2, for notational convenience. We have

$$\begin{aligned}
\mathsf{Var}(E_1) &= \{f, g, x\} \\
X_{E_1} &= \{\langle f \rangle, \langle g \rangle, \langle x \rangle\} \\
Y_{E_1} &= \{[\![E_1]\!], [\![\lambda g.g(f_1\mathsf{0})(f_2(\lambda x.x))]\!], [\![g(f_1\mathsf{0})(f_2(\lambda x.x))]\!], [\![g(f_1\mathsf{0})]\!], \\
&\quad [\![f_2(\lambda x.x)]\!], [\![g]\!], [\![f_1\mathsf{0}]\!], [\![f_1]\!], [\![\mathsf{0}]\!], [\![f_2]\!], [\![\lambda x.x]\!], [\![x]\!]\}
\end{aligned}$$

The constraint system $C(E_1)$ has the pointwise $\subseteq$-least solution φ_1:

$$\begin{aligned}
\varphi_1([\![E_1]\!]) &= \{E_1\} \\
\varphi_1([\![\lambda g.g(f_1\mathsf{0})(f_2(\lambda x.x))]\!]) &= \{\lambda g.g(f_1\mathsf{0})(f_2(\lambda x.x))\} \\
\varphi_1([\![g(f_1\mathsf{0})(f_2(\lambda x.x))]\!]) &= \varphi_1([\![g(f_1\mathsf{0})]\!]) = \varphi_1([\![f_2(\lambda x.x)]\!]) = \varphi_1(\langle g \rangle) \\
&= \varphi_1([\![g]\!]) = \varphi_1([\![f_1\mathsf{0}]\!]) = \varphi_1(\langle f \rangle) \\
&= \varphi_1([\![f_1]\!]) = \varphi_1[\![f_2]\!]) = \varphi_1(\langle x \rangle) = \varphi_1([\![x]\!]) = \emptyset \\
\varphi_1[\![\mathsf{0}]\!]) &= \{\mathsf{0}\} \\
\varphi_1[\![\lambda x.x]\!]) &= \{\lambda x.x\}
\end{aligned}$$

Next consider again

$$E_2 \;=\; (\lambda f.\lambda g.g(f_1(\lambda a.\mathsf{0}))(f_2(\lambda b.\lambda x.x)))(\lambda y.\mathsf{0}),$$

where we have labeled the occurrences of f as f_1 and f_2, for notational convenience. The constraint system $C(E_2)$ has the pointwise $\subseteq$-least solution φ_2:

$$\begin{aligned}
\varphi_2(\langle y\rangle) &= \varphi_2([\![\lambda a.\mathsf{0}]\!]) = \varphi_2([\![\lambda b.\lambda x.x]\!]) = \{\lambda a.\mathsf{0}, \lambda b.\lambda x.x\}\\
\varphi_2([\![\mathsf{0}]\!]) &= \{\mathsf{Int}\}\\
\varphi_2(\langle f\rangle) &= \varphi_2([\![f_1]\!]) = \varphi_2([\![f_2]\!]) = \{\lambda y.\mathsf{0}\}\\
\varphi_2(\langle g\rangle) &= \varphi_2([\![g]\!]) = \varphi_2(\langle a\rangle) = \varphi_2(\langle b\rangle) = \varphi_2(\langle x\rangle) = \varphi_2([\![x]\!]) = \emptyset\\
&\text{etc.}
\end{aligned}$$

3. THE TYPE SYSTEM

We use v to range over type variables drawn from a countably infinite set $\mathbf{Tv}$. Types are defined by the grammar

$$t ::= t_1 \to t_2 \mid \mathsf{Int} \mid v \mid \mu v.t \mid \top \mid \bot,$$

with the restriction that a type is not allowed to contain anything of the form

$$\mu v_1. \ldots \mu v_n.v_1.$$

We identify types with their infinite unfoldings under the rule

$$\mu v.t = t[v := \mu v.t].$$

Such infinite unfolding eliminates all uses of μ in types. It follows that types are a class of regular trees over the alphabet

$$\Sigma = \{\to, \mathsf{Int}, \top, \bot\} \cup \mathbf{Tv}.$$

There is a subtype relation $\leq$ on types:

$$\begin{array}{lcll}
t & \leq & t & \text{for all types } t,\\
\bot & \leq & t & \text{for all types } t \text{ except } \mathsf{Int}, \text{ and}\\
t & \leq & \top & \text{for all types } t \text{ except } \mathsf{Int}.
\end{array}$$

It is straightforward to show that $\leq$ is a partial order. Notice that $\bot$ is a lower bound, and $\top$ is an upper bound for only the function types but not Int. A more suggestive notation might be $\bot_{\to}$ for $\bot$ and $\top_{\to}$ for $\top$.

A type environment is a partial function with finite domain which maps λ-variables to types. We use A to range over type environments. We use the notation $A[x : t]$ to denote an environment which maps x to t, and maps y, where $y \neq x$, to $A(y)$. A type judgment has the form $A \vdash e : t$, and it means that in the type environment A, the expression e has type t. Formally, this holds when it is derivable using the rules below.

$$A[x : t] \vdash x : t \tag{1}$$

$$\frac{A[x : s] \vdash e : t}{A \vdash \lambda x.e : u} \quad (s \to t \leq u) \tag{2}$$

$$\frac{A \vdash e_1 : u \quad A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \quad (u \leq s \to t) \tag{3}$$

$$A \vdash 0 : \mathsf{Int} \tag{4}$$

$$\frac{A \vdash e : \mathsf{Int}}{A \vdash \mathsf{succ}\ e : \mathsf{Int}} \tag{5}$$

Notice that there is no subsumption rule; instead subtyping can only be used in a restricted way in rules (2) and (3). We say that e is RS-typable if $A \vdash e : t$ is derivable for some A, t. (RS stands for "restricted subtyping.") The type system has the subject reduction property; that is, if $A \vdash e : t$ is derivable, and e beta-reduces to e', then $A \vdash e' : t$ is derivable. This can be proved by straightforward induction on the structure of the derivation of $A \vdash e : t$.

Here follow type derivations for the two λ-terms E_1, E_2 from Section 1. The first type derivation uses the abbreviation $A = \emptyset[f : \bot][g : \bot]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A \vdash g : \bot \quad \dfrac{A \vdash f : \bot \quad A \vdash 0 : \mathsf{Int}}{A \vdash f0 : \bot}}{A \vdash g(f0) : \bot} \quad \dfrac{A \vdash f : \bot \quad \dfrac{A[x : \bot] \vdash x : \bot}{A \vdash \lambda x.x : \bot \to \bot}}{A \vdash f(\lambda x.x) : \bot}}{A \vdash g(f0)(f(\lambda x.x)) : \bot}}{\emptyset[f : \bot] \vdash \lambda g.g(f0)(f(\lambda x.x)) : \bot \to \bot}}{\emptyset \vdash \lambda f.\lambda g.g(f0)(f(\lambda x.x)) : \bot \to (\bot \to \bot)}$$

Notice the four uses of subtyping. Notice also that the only possible type for f is $\bot$.

The second derivation uses the abbreviation $A' = \emptyset[f : (\top \to \mathsf{Int})][g : \bot]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A' \vdash g : \bot \quad \dfrac{\cdots}{A' \vdash f(\lambda a.0) : \mathsf{Int}}}{A' \vdash g(f(\lambda a.0)) : \bot} \quad \dfrac{\cdots}{A' \vdash f(\lambda b.\lambda x.x) : \mathsf{Int}}}{A' \vdash g(f(\lambda a.0))(f(\lambda b.\lambda x.x)) : \bot}}{\emptyset[f : (\top \to \mathsf{Int})] \vdash \lambda g.g(f(\lambda a.0))(f(\lambda b.\lambda x.x)) : \bot \to \bot} \quad \dfrac{\emptyset[y : \top] \vdash 0 : \mathsf{Int}}{\emptyset \vdash \lambda y.0 : \top \to \mathsf{Int}}}{\emptyset \vdash \lambda f.\lambda g.g(f(\lambda a.0))(f(\lambda b.\lambda x.x)) : (\top \to \mathsf{Int}) \to (\bot \to \bot)}}{\emptyset \vdash (\lambda f.\lambda g.g(f(\lambda a.0))(f(\lambda b.\lambda x.x)))(\lambda y.0) : \bot \to \bot}$$

Notice that the only possible common type for both $(\lambda a.0)$ and $(\lambda b.\lambda x.x)$ is $\top$.

The reason why there is no subsumption rule of the form

$$\frac{A \vdash e : s}{A \vdash e : t} \quad (s \le t)$$

is that we want to disallow the use of subsumption immediately after a use of the rule for variables. If we add a subsumption rule, then more λ-terms become typable. For example, consider

$$E_3 \ = \ (\lambda f.\lambda g.g(f(\lambda x.0))(ff))(\lambda y.y).$$

If we have a subsumption rule, then we can give $\lambda y.y$ the type $\top \to \top$; we can give both $\lambda x.0$ and the last occurrence of f the type $\top$; and it is then straightforward to complete a type derivation for E_3. Notice that the fragment of the type derivation for the last occurrence of f is of the form

$$\frac{A \vdash f : \top \to \top}{A \vdash f : \top} \quad (\top \to \top \le \top).$$

Without a subsumption rule, this type derivation is not possible. Indeed, no type derivation using rules (1)–(5) is possible. To see that, let s_1 be the type of $\lambda y.y$, and let s_2 be the type of f. From $\lambda y.y$ we have $t \to t \leq s_1$, where t is the type of x. Moreover, from (ff) we have $s_2 \leq s_2 \to u$, where u is the type of (ff). We have $s_1 = s_2$, so $t \to t \leq s_1 = s_2 \leq s_2 \to u$; hence $t \to t = s_1 = s_2 = s_2 \to u$; hence $s_1 = s_2 = t = u = \mu\alpha.(\alpha \to \alpha)$. Consider now $(f(\lambda x.0))$. The type of $\lambda x.0$ is of the form $s' \to \mathsf{Int}$ or $\top$. In both cases, it cannot be an argument of a function of type $\mu\alpha.(\alpha \to \alpha)$. We conclude that E_3 is not RS-typable.

4. THE EQUIVALENCE RESULT

THEOREM 4.1. *A λ-term P is 0-CFA$_=$ safe if and only if P is RS-typable.*

We prove this theorem in two steps. Lemma 4.3 shows that if P is 0-CFA$_=$ safe, then P is RS-typable. To prove that lemma we use the technique from Palsberg and Pavlopoulou [1998]. Lemma 4.4 shows that if P is RS-typable, then P is 0-CFA$_=$ safe. To prove that lemma we use a technique which is more direct than the one used to show a similar result, for 0-CFA$_\subseteq$, in Palsberg and O'Keefe [1995].

4.1 From Flows to Types

First we consider the mapping of flows to types. Given a program P, a map $\varphi \in \mathsf{Cmap}(P)$, and $S \subseteq \mathsf{Abs}(P)$, we say that S is φ-consistent if for all $\lambda x_1.e_1, \lambda x_2.e_2 \in S$ we have $\varphi(\langle x_1 \rangle) = \varphi(\langle x_2 \rangle)$ and $\varphi([\![e_1]\!]) = \varphi([\![e_2]\!])$. Given a program P and $\varphi \in \mathsf{Cmap}(P)$, define the equation system $\Gamma(P, \varphi)$:

—For each $S \in \mathit{range}(\varphi)$, let v_S be a type variable, and
  —if $S = \emptyset$, then $\Gamma(P, \varphi)$ contains the equation

$$v_S \;=\; \bot;$$

  —if $S = \{\mathsf{Int}\}$, then $\Gamma(P, \varphi)$ contains the equation

$$v_S \;=\; \mathsf{Int};$$

  —if $S = \{\lambda x_1.e_1, \ldots, \lambda x_n.e_n\}, n > 0$, then there are two cases: either S is φ-consistent and then $\Gamma(P, \varphi)$ contains the equation

$$v_S \;=\; v_{\varphi(\langle x_1 \rangle)} \to v_{\varphi([\![e_1]\!])};$$

  otherwise $\Gamma(P, \varphi)$ contains the equation

$$v_S = \top.$$

Every equation system $\Gamma(P, \varphi)$ has a unique solution. To see this, notice that for every type variable, there is exactly one equation with that variable as the left-hand side. Thus, intuitively, we obtain the solution by using each equation as an unfolding rule, possibly infinitely often.

LEMMA 4.2. *If $\varphi \in \mathsf{Cmap}(P)$, $\varphi(w_1) \subseteq \varphi(w_2) \subseteq \mathsf{Abs}(P)$, and ψ is the unique solution of $\Gamma(P, \varphi)$, then $\psi(v_{\varphi(w_1)}) \leq \psi(v_{\varphi(w_2)})$.*

PROOF. Support first that $\varphi(w_1) = \emptyset$. We then have $\psi(v_{\varphi(w_1)}) = \bot$. Since $\varphi(w_2) \subseteq \mathsf{Abs}(P)$, we have $\bot \leq \psi(v_{\varphi(w_2)})$; hence $\psi(v_{\varphi(w_1)}) \leq \psi(v_{\varphi(w_2)})$.

Suppose then that $\varphi(w_1)$ is φ-inconsistent. From $\varphi(w_1) \subseteq \varphi(w_2)$ we then have also that $\varphi(w_2)$ is φ-inconsistent, so $\psi(v_{\varphi(w_1)}) = \top = \psi(v_{\varphi(w_2)})$.

Suppose finally that $\varphi(w_1) = \{\lambda x_1.e_1, \ldots, \lambda x_n.e_n\}$, $n > 0$, and that $\varphi(w_1)$ is φ-consistent. There are two cases. If $\varphi(w_2)$ is φ-inconsistent, then $\psi(v_{\varphi(w_1)}) = \psi(v_{\varphi(\langle x_1 \rangle)}) \to \psi(v_{\varphi(\langle e_1 \rangle)}) \leq \top = \psi(v_{\varphi(w_2)})$. If $\varphi(w_2)$ is φ-consistent, then $\psi(v_{\varphi(w_1)}) = \psi(v_{\varphi(\langle x_1 \rangle)}) \to \psi(v_{\varphi(\langle e_1 \rangle)}) = \psi(v_{\varphi(w_2)})$. □

LEMMA 4.3. *If φ satisfies $C(P)$, $A = \lambda(x \in \mathsf{Var}(P)).\psi(v_{\varphi(\langle x \rangle)})$, ψ is the unique solution of $\Gamma(P, \varphi)$, and e is a subterm of P, then we can derive $A \vdash e : \psi(v_{\varphi([\![e]\!])})$.*

PROOF. We proceed by induction on the structure of e. In the base case, consider first $e \equiv x$. We have $A(x) = \psi(v_{\varphi(\langle x \rangle)})$, so we can derive $A \vdash x : \psi(v_{\varphi(\langle x \rangle)})$. This is the desired derivation because $\varphi(\langle x \rangle) = \varphi([\![x]\!])$.

Consider then $e \equiv \mathsf{0}$. We have $\varphi([\![\mathsf{0}]\!]) = \{\mathsf{Int}\}$, so $\psi(v_{\varphi([\![\mathsf{0}]\!])}) = \mathsf{Int}$; and we can derive $A \vdash \mathsf{0} : \psi(v_{\varphi([\![\mathsf{0}]\!])})$.

In the induction step, consider first $e = \mathsf{succ}\ e'$. We have $\varphi([\![e]\!]) = \varphi([\![\mathsf{succ}\ e']\!]) = \{\mathsf{Int}\}$, so $\psi(v_{\varphi([\![e]\!])}) = \psi(v_{\varphi([\![\mathsf{succ}\ e']\!])}) = \mathsf{Int}$. From the induction hypothesis we have that we can derive $A \vdash e' : \mathsf{Int}$, and we can then also derive $A \vdash \mathsf{succ}\ e' : \mathsf{Int}$.

Consider next $e \equiv \lambda x.e'$. We have $\{\lambda x.e'\} \subseteq \varphi([\![\lambda x.e']\!])$, and from Lemma 4.2 we get $\psi(v_{\{\lambda x.e'\}}) \leq \psi(v_{\varphi([\![\lambda x.e']\!])})$. From the induction hypothesis, we have that we can derive $A \vdash e' : \psi(v_{\varphi([\![e']\!])})$, and we have $A = A[x : \psi(v_{\varphi(\langle x \rangle)})]$. Thus, we can also derive $A \vdash \lambda x.e' : \psi(v_{\varphi([\![\lambda x.e']\!])})$ because $\psi(v_{\varphi([\![\lambda x.e']\!])}) \geq \psi(v_{\{\lambda x.e'\}}) = \psi(v_{\varphi(\langle x \rangle)}) \to \psi(v_{\varphi([\![e']\!])})$.

Finally, consider $e \equiv e_1 e_2$. We have $\varphi([\![e_1]\!]) \subseteq \mathsf{Abs}(P)$, and for every $\lambda x.e' \in \varphi([\![e_1]\!])$ we have $\varphi([\![e_2]\!]) = \varphi(\langle x \rangle)$ and $\varphi([\![e']\!]) = \varphi([\![e_1 e_2]\!])$. From the induction hypothesis, we have that we can derive $A \vdash e_1 : \psi(v_{\varphi([\![e_1]\!])})$ and $A \vdash e_2 : \psi(v_{\varphi([\![e_2]\!])})$. There are two cases. If $\varphi([\![e_1]\!]) = \emptyset$, then $\psi(v_{\varphi([\![e_1]\!])}) = \bot \leq \psi(v_{\varphi([\![e_2]\!])}) \to \psi(v_{\varphi([\![e_1 e_2]\!])})$, and we can derive $A \vdash e_1 e_2 : \psi(v_{\varphi([\![e_1 e_2]\!])})$. If $\varphi([\![e_1]\!]) \neq \emptyset$, then we use $\varphi([\![e_1]\!]) \subseteq \mathsf{Abs}(P)$ to conclude that $\varphi([\![e_1]\!]) = \{\lambda x_1.e'_1, \ldots, \lambda x_n.e'_n\}, n > 0$. Moreover, $\varphi(\langle x_i \rangle) = \varphi([\![e_2]\!]) = \varphi(\langle x_j \rangle)$ for all $i \in 1..n, j \in 1..n$, and $\varphi([\![e'_i]\!]) = \varphi([\![e_1 e_2]\!]) = \varphi([\![e'_j]\!])$ for all $i \in 1..n, j \in 1..n$. Hence, $\varphi([\![e_1]\!])$ is φ-consistent. Thus, $\psi(v_{\varphi([\![e_1]\!])}) = \psi(v_{\varphi(\langle x_1 \rangle)}) \to \psi(v_{\varphi([\![e'_1]\!])}) = \psi(v_{\varphi([\![e_2]\!])}) \to \psi(v_{\varphi([\![e_1 e_2]\!])})$, so we can derive $A \vdash e_1 e_2 : \psi(v_{\varphi([\![e_1 e_2]\!])})$. □

For example, consider again the λ-term

$$E_1 \ = \ \lambda f.\lambda g.g(f\mathsf{0})(f(\lambda x.x)),$$

and recall the function φ_1 from Section 2 which satisfies $C(E_1)$. The constraint system $\Gamma(E_1, \varphi_1)$ is

$$\begin{aligned}
v_{\emptyset} &= \bot \\
v_{\{\mathsf{Int}\}} &= \mathsf{Int} \\
v_{\{\lambda x.x\}} &= v_{\emptyset} \to v_{\emptyset} \\
v_{\{\lambda g.g(f\mathsf{0})(f(\lambda x.x))\}} &= v_{\emptyset} \to v_{\emptyset} \\
v_{\{E_1\}} &= v_{\emptyset} \to v_{\{\lambda g.g(f\mathsf{0})(f(\lambda x.x))\}}.
\end{aligned}$$

When we plug this into the construction in the proof of Lemma 4.3, we get the type derivation for E_1 shown in Section 3. We leave it to the reader to carry out

the construction for E_2 and φ_2. It will lead to the type derivation for E_2 shown in Section 3.

4.2 From Types to Flows

Next we consider the mapping of types to flows. If Δ is the type derivation $A \vdash P : t$, then define f_Δ to map types to elements of $\mathsf{CL}(P)$:

$$\begin{aligned} f_\Delta(\bot) &= \emptyset \\ f_\Delta(\mathsf{Int}) &= \{\mathsf{Int}\} \\ f_\Delta(\top) &= \mathsf{Abs}(P) \\ f_\Delta(s \to s') &= \text{the set of occurrences } \lambda x.e \text{ of } P \text{ where } \Delta \text{ contains} \\ &\quad \text{a judgment of the form } A'[x : s] \vdash e : s' \text{ for some } A'. \end{aligned}$$

Define also $\varphi_\Delta \in \mathsf{Cmap}(P)$ such that

$$\begin{aligned} \varphi_\Delta(\langle x \rangle) = f_\Delta(s) \quad & \text{for an occurrence } \lambda x.e \text{ of } P \text{ where } \Delta \text{ contains} \\ & \text{a judgment of the form } A'[x : s] \vdash e : s' \text{ for some } A', \\ \varphi_\Delta([\![e]\!]) = f_\Delta(s) \quad & \text{for an occurrence } e \text{ of } P \text{ where } \Delta \text{ contains} \\ & \text{a judgment of the form } A' \vdash e : s \text{ for some } A'. \end{aligned}$$

LEMMA 4.4. *If Δ is the type derivation $A \vdash P : t$, then φ_Δ satisfies $C(P)$.*

PROOF. We consider in turn each of the constraints in $C(P)$. For an occurrence of $\mathsf{0}$ and the constraint $[\![\mathsf{0}]\!] = \{\mathsf{Int}\}$, we have that Δ contains a judgment of the form $A' \vdash \mathsf{0} : \mathsf{Int}$, and $\varphi_\Delta([\![\mathsf{0}]\!]) = f_\Delta(\mathsf{Int}) = \{\mathsf{Int}\}$.

For an occurrence of $\mathsf{succ}\ e$ and the constraints $[\![e]\!] = \{\mathsf{Int}\}$ and $[\![\mathsf{succ}\ e]\!] = \{\mathsf{Int}\}$, we have that Δ contains judgments of the forms $A \vdash e : \mathsf{Int}$ and $A \vdash \mathsf{succ}\ e : \mathsf{Int}$, and $\varphi_\Delta([\![e]\!]) = f_\Delta(\mathsf{Int}) = \{\mathsf{Int}\}$ and $\varphi_\Delta([\![\mathsf{succ}\ e]\!]) = f_\Delta(\mathsf{Int}) = \{\mathsf{Int}\}$.

For an occurrence x and the constraint $\langle x \rangle = [\![x]\!]$, we have that Δ contains a judgment of the form $A[x : s] \vdash x : s$ for some s, and $\varphi_\Delta(\langle x \rangle) = f_\Delta(s) = \varphi_\Delta([\![x]\!])$.

For an occurrence $\lambda x.e$ and the constraint $\{\lambda x.e\} \subseteq [\![\lambda x.e]\!]$, we have that Δ contains judgments of the forms $A'[x : s] \vdash e : s'$ and $A' \vdash \lambda x.e : u$ where $s \to s' \leq u$. There are two cases. If $u = \top$, then $\varphi_\Delta([\![\lambda x.e]\!]) = f_\Delta(\top) = \mathsf{Abs}(P) \supseteq \{\lambda x.e\}$. If $u = s \to s'$, then $\varphi_\Delta([\![\lambda x.e]\!]) = f_\Delta(s \to s') \supseteq \{\lambda x.e\}$.

For an occurrence $e_1 e_2$ and the constraint $[\![e_1]\!] \subseteq \mathsf{Abs}(P)$, and the constraints, for every occurrence $\lambda x.e$ in $\mathsf{Abs}(P)$,

$$\begin{aligned} \{\lambda x.e\} \subseteq [\![e_1]\!] &\Rightarrow [\![e_2]\!] = \langle x \rangle \\ \{\lambda x.e\} \subseteq [\![e_1]\!] &\Rightarrow [\![e]\!] = [\![e_1 e_2]\!], \end{aligned}$$

we have that Δ contains judgments of the forms $A' \vdash e_1 : u$, $A' \vdash e_2 : s$, and $A' \vdash e_1 e_2 : s'$, where $u \leq s \to s'$. There are two cases. If $u = \bot$, then $\varphi_\Delta([\![e_1]\!]) = f_\Delta(\bot) = \emptyset \subseteq \mathsf{Abs}(P)$, and the other constraints are vacuously satisfied. If $u = s \to s'$, then $\varphi_\Delta([\![e_1]\!]) = f_\Delta(s \to s')$. From the definition of $f_\Delta(s \to s')$ we have $f_\Delta(s \to s') \subseteq \mathsf{Abs}(P)$. Suppose $\lambda x.e \in \varphi_\Delta([\![e_1]\!])$. We have $\varphi_\Delta(\langle x \rangle) = f_\Delta(s) = \varphi_\Delta([\![e_2]\!])$ and $\varphi_\Delta([\![e]\!]) = f_\Delta(s') = \varphi_\Delta([\![e_1 e_2]\!])$. □

5. CONCLUDING REMARKS

If we remove from Section 3 the types $\top$, $\bot$, and the notion of subtyping, then we get a traditional system of recursive types. Given a program P and a map

$\varphi \in \mathsf{Cmap}(P)$, we say that φ is consistent if for all $S \in \mathit{range}(\varphi)$ we have that S is φ-consistent. If we add to Section 2 the conditions

—$\mathsf{CL}(P)$ does not contain $\emptyset$ and

—$\mathsf{Cmap}(P)$ does not contain inconsistent maps

then we get a notion of flow-based safety analysis which we here will refer to as restricted-0-CFA$_=$ safety. It is easy to modify the proof of Lemma 4.3 to show the following result.

THEOREM 5.1. *If a λ-term P is restricted-0-CFA$_=$ safe, then P is typable with recursive types.*

Intuitively, the theorem says that if we want a flow analysis weaker than recursive types, then we can start with 0-CFA$_=$, outlaw $\emptyset$, and insist on internal consistency in all flow sets. The converse of Theorem 5.1 is false. For example, if we attempt to modify the proof of Lemma 4.4, then we run into trouble in the case $e_1 e_2$, because there is no guarantee that $f_\Delta(s \to s') \neq \emptyset$, where $s \to s'$ is the type of e_1. Such a situation arises with the program

$$E_4 \;=\; \lambda x.\mathsf{succ}(x\mathsf{0}).$$

With recursive types but not subtyping, there is just one type derivation for E_4, using the abbreviation $A = \emptyset[x : (\mathsf{Int} \to \mathsf{Int})]$:

$$\dfrac{\dfrac{\dfrac{A \vdash x : \mathsf{Int} \to \mathsf{Int} \qquad A \vdash \mathsf{0} : \mathsf{Int}}{A \vdash x\mathsf{0} : \mathsf{Int}}}{A \vdash \mathsf{succ}(x\mathsf{0}) : \mathsf{Int}}}{\emptyset \vdash \lambda x.\mathsf{succ}(x\mathsf{0}) : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}}$$

We have

$$\mathsf{CL}(E_4) \;=\; \{\; \emptyset, \{\lambda x.\mathsf{succ}(x\mathsf{0})\}, \{\mathsf{Int}\} \;\}.$$

Suppose $\varphi \in \mathsf{Cmap}(E_4)$ satisfies $C(E_4)$. It it straightforward to show that $\varphi([\![x]\!]) \neq \{\mathsf{Int}\}$ and $\varphi([\![x]\!]) \neq \{\lambda x.\mathsf{succ}(x\mathsf{0})\}$, so $\varphi([\![x]\!]) = \emptyset$. Thus, E_4 is not restricted-0-CFA$_=$ safe, and E_4 is therefore a counterexample to the converse of Theorem 5.1.

We leave it as an open problem to find a flow analysis equivalent to recursive types.

An unusual aspect of Heintze's definition of 0-CFA$_=$ is that Int and functions cannot occur in the same flow set. To allow that we might define

$$\mathsf{CL}(P) \;=\; \mathit{Powerset}(\mathsf{Abs}(P) \cup \{\mathsf{Int}\}),$$

and change the constraints from Section 2 such that the constraints for $\mathsf{0}$ and $\mathsf{succ}\ e$ become

$$\begin{array}{rcll} \{\mathsf{Int}\} & \subseteq & [\![\mathsf{0}]\!] & \\ [\![e]\!] & \subseteq & \{\mathsf{Int}\} & \text{(a safety constraint)} \\ \{\mathsf{Int}\} & \subseteq & [\![\mathsf{succ}\ e]\!]. & \end{array}$$

There is a systematic way of obtaining this modified flow analysis: begin with the constraints for 0-CFA$_\subseteq$ [Palsberg and O'Keefe 1995] and

—change $\langle x \rangle \subseteq [\![x]\!]$ to $\langle x \rangle = [\![x]\!]$ and
—change

$$\{\lambda x.e\} \subseteq [\![e_1]\!] \Rightarrow [\![e_2]\!] \subseteq \langle x \rangle$$
$$\{\lambda x.e\} \subseteq [\![e_1]\!] \Rightarrow [\![e]\!] \subseteq [\![e_1 e_2]\!]$$

to

$$\{\lambda x.e\} \subseteq [\![e_1]\!] \Rightarrow [\![e_2]\!] = \langle x \rangle$$
$$\{\lambda x.e\} \subseteq [\![e_1]\!] \Rightarrow [\![e]\!] = [\![e_1 e_2]\!].$$

All other constraints remain the same.

The type system that matches the modified flow analysis can be obtained by changing the type system from Section 3 such that $\leq$ is the smallest reflexive and transitive relation on types where $\bot \leq t \leq \top$ for all types t, and such that the type rules for $\mathsf{0}$ and $\mathsf{succ}\ e$ become

$$A \vdash \mathsf{0} : t \quad (\mathsf{Int} \leq t)$$

$$\frac{A \vdash e : s}{A \vdash \mathsf{succ}\ e : t} \quad (s \leq \mathsf{Int},\ \mathsf{Int} \leq t).$$

Notice that in this modified type system, $\bot$ is the least type, and $\top$ is the greatest type.

REFERENCES

Amadio, R. M. and Cardelli, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15,* 4, 575–631.

Ashley, J. M. 1996. A practical and flexible flow analysis for higher-order languages. In *Proceedings of POPL'96, 23rd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 184–194.

Bondorf, A. and Jørgensen, J. 1993. Efficient analyses for realistic off-line partial evaluation. *J. of Func. Program. 3,* 3, 315–346.

Bruce, K. B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S. F., Trifonov, V., Leavens, G. T., and Pierce, B. C. 1995. On binary methods. *Theory Pract. Obj. Syst. 1,* 3, 221–242.

DeFouw, G., Grove, D., and Chambers, C. 1998. Fast interprocedural class analysis. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 222–236.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, Mass.

Heintze, N. 1995. Control-flow analysis and type systems. In *Proceedings of SAS'95, International Static Analysis Symposium*. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Berlin, 189–206.

Heintze, N. and McAllester, D. 1997. Linear-time subtransitive control flow analysis. In *Proceedings of ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM, New York, 261–272.

Henglein, F. 1992. Dynamic typing. In *Proceedings of ESOP'92, European Symposium on Programming*. Lecture Notes in Computer Science, vol. 582. Springer-Verlag, Berlin, 233–253.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Mass.

Palsberg, J. 1995. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst. 17,* 1 (Jan.), 47–62.

Palsberg, J. and O'Keefe, P. M. 1995. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst. 17,* 4 (July), 576–599.

Palsberg, J. and Pavlopoulou, C. 1998. From polyvariant flow information to intersection and union types. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 197–208.

Palsberg, J. and Schwartzbach, M. I. 1994. *Object-Oriented Type Systems*. John Wiley & Sons, New York.

Shapiro, M. and Horwitz, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 1–14.

Shivers, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, CMU–CS–91–145, Carnegie Mellon Univ., Pittsburgh, Pa.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 32–41.

Received December 1997; revised June 1998; accepted August 1998