A Type System Equivalent to Flow Analysis*

Jens Palsberg† Patrick O'Keefe‡

Abstract

Flow-based safety analysis of higher-order languages has been studied by Shivers, and Palsberg and Schwartzbach. Open until now is the problem of finding a type system that accepts exactly the same programs as safety analysis.

In this paper we prove that Amadio and Cardelli's type system with subtyping and recursive types accepts the same programs as a certain safety analysis. The proof involves mappings from types to flow information and back. As a result, we obtain an inference algorithm for the type system, thereby solving an open problem.

1 Introduction

1.1 Background

Many program analyses for higher-order languages are based on *flow analysis*, also known as *closure analysis*. Examples include various analyses in the Standard ML of New Jersey compiler [3], and the binding-time analyses for Scheme in the partial evaluators Schism [5] and Similix [4]. Such analyses have the advantage that they can be applied to *untyped* languages. This is in contrast to more traditional abstract interpretations which use types when defining the abstract domains.

Recently, it has become popular to define program analyses for typed languages by annotating the types with information about program behavior [8, 2]. This has lead to clear specifications of a range of analyses, and often such an analysis can be efficiently computed by a straightforward extension of a known type inference algorithm.

*Extended abstract of a paper in Proc. POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages.

†Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark. E-mail: palsberg@daimi.aau.dk.

‡151 Coolidge Avenue #211, Watertown, MA 02172, USA. E-mail: pmo@world.std.com.

The precision of a type-based analysis depends on the expressiveness of the underlying type system. Similarly, the precision of a flow-based analysis depends on the expressiveness of the underlying flow analysis. In this paper we address an instance of the following fundamental question:

> **Fundamental question.** What type-based analysis computes the same information as a given flow-based analysis?

We consider the case of flow-based *safety* analysis, that is, an analysis which collects type information from for example constants and applications of primitive operations. Such an analysis was first presented in 1991 by Shivers [13] who called it type *recovery*. Later, Palsberg and Schwartzbach [11, 10] proved that on the basis of the collected information, one can define a predicate which accepts only programs which cannot go wrong. They called this *safety* analysis. They also proved that their safety analysis accepts more programs than simple type inference.

In this paper, we consider the following instance of the above question:

> Which type system accepts the same programs as safety analysis?

The particular safety analysis we consider is defined in Section 3.

Our thesis is that the type system that answers this specific question will in many cases also be the answer to the fundamental question.

1.2 Our result

We prove that a natural type system with subtyping and recursive types accepts the same programs as safety analysis. The proof involves mappings from types to flow information and back.

The type system has been studied by Amadio and Cardelli [1], and an $O(n^2)$ algorithm for deciding the subtyping relation has been presented by Kozen, Palsberg, Schwartzbach, [6]. Open until now is the question of type inference. As a corollary of our result we get a type inference algorithm which works by first doing safety analysis and then mapping the flow information to types.

The set of types can be presented by the following grammar:

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

The type system contains the following components: the binary function type constructor $\to$, the constant type Int, the possibility for creating recursive types, and two more constant types $\top$, and $\bot$. Moreover, there is a subtype relation, written $\leq$. In contrast, safety analysis uses an abstract domain containing sets of syntactic occurrences of abstractions and the constant Int.

In slogan-form, our result reads:

Flow analysis + Safety checks =
Simple types + Recursive types + $\top$ + $\bot$ + Subtyping

Each component of the type system captures a facet of flow analysis:

- The function type constructor $\rightarrow$ corresponds to a set of abstractions. Intuitively, a function type is less concrete than a set of abstractions. Indeed, the other components of the type system are essential to make it accept the same programs as the safety analysis.
- The constant Int is used for the same purpose in both systems. For simplicity, we do not consider other base types, or product and sum constructors, etc. Such constructs can be handled by techniques that are similar to the ones we will present.
- Recursive types are needed in order that safety analysis accepts all programs that do not contain constants.
- The constant $\top$ corresponds to the largest possible set of flow information. This type is needed for variables which can hold both a function and a base value. Intuitively, a program with such a variable should be type incorrect. However, the flow-based analysis may detect that this variable is only passed around but never actually used. For the type system to have that capability, $\top$ is required.
- The constant $\bot$ corresponds to the empty set of flow information. This type is needed for variables which are used both as a function and as a base value. Intuitively, a program that uses a variable in both these ways should be type incorrect. However, the flow-based analysis may detect that this part of the program will never be executed. For the type system to have that capability, $\bot$ is required.
- Subtyping is needed to capture flow of information. Intuitively, if information flows from A to B, then the type of A will be a subtype of the type of B.

Palsberg and Schwartzbach [10, 11] proved that the system *without* $\bot$ accepts at most as many programs as safety analysis. In this paper we present the type system which accepts exactly the same programs as safety analysis. This may be seen as a natural culmination of the previous results.

1.3 Examples

Our example language is a λ-calculus, generated by the following grammar:

$$E ::= x \mid \lambda x.E \mid E_1 E_2 \mid 0 \mid \mathsf{succ}\ E$$

Programs that yield a run-time error include $(\mathsf{0}\ x)$, $\mathsf{succ}(\lambda x.x)$, and $(\mathsf{succ}\ \mathsf{0})(x)$, because $\mathsf{0}$ is not a function, succ cannot be applied to functions, and $(\mathsf{succ}\ \mathsf{0})$ is not a function. These programs are not typable and they are rejected by safety analysis. Some programs can be typed in the type system without the use of $\bot$ and $\top$, for example

$$\lambda x.xx\ :\ \mu\alpha.\alpha \to \alpha\ ,$$

where $E : t$ means "E has type t". Some programs require the use of $\top$, for example

$$(\lambda f.(\lambda x.fI)(f\mathsf{0}))I\ :\ \top\ ,$$

where $I = \lambda x.x$. Note that $\top$ is the only type of $(\lambda f.(\lambda x.fI)(f\mathsf{0}))I$. Some programs require the use of $\bot$, for example

$$\lambda x.x(\mathsf{succ}\ x)\ :\ \bot \to t \quad \text{for any } t.$$

Both type inference and safety analysis can be phrased as solving a system of constraints, derived from the program text. We will now present the constraint systems for the last of the above examples. For notational convenience, we give each of the two occurrences of x a label so that the λ-term reads $\lambda x.x_1(\mathsf{succ}\ x_2)$. For brevity, let $E = \lambda x.x_1(\mathsf{succ}\ x_2)$. The constraint system for type inference looks as follows:

$$\begin{array}{rcl}
x \to [\![x_1(\mathsf{succ}\ x_2)]\!] & \leq & [\![E]\!] \\
[\![x_1]\!] & \leq & [\![\mathsf{succ}\ x_2]\!] \to [\![x_1(\mathsf{succ}\ x_2)]\!] \\
x & \leq & [\![x_1]\!] \\
x & \leq & [\![x_2]\!] \\
\mathsf{Int} & \leq & [\![\mathsf{succ}\ x_2]\!] \\
[\![x_2]\!] & \leq & \mathsf{Int}
\end{array}$$

Here, the symbols x, $[\![x_1]\!]$, $[\![x_2]\!]$, $[\![\mathsf{succ}\ x_2]\!]$, $[\![x_1(\mathsf{succ}\ x_2)]\!]$, $[\![E]\!]$ are type variables. Solving this constraint system yields that the possible types for the λ-term $\lambda x.x(\mathsf{succ}\ x)$ are $\top$ and $\bot \to t$ for any type t. Among these, $\bot \to \bot$ is a least type. In general, however, such a constraint system need not have a least solution.

The constraint system for safety analysis looks as follows:

$$\begin{array}{rcl}
\{E\} & \subseteq & [\![E]\!] \\
[\![x_1]\!] & \subseteq & \{E\} \\
x & \subseteq & [\![x_1]\!] \\
x & \subseteq & [\![x_2]\!] \\
\{E\} \subseteq [\![x_1]\!] & \Rightarrow & [\![\mathsf{succ}\ x_2]\!] \subseteq x \\
\{E\} \subseteq [\![x_1]\!] & \Rightarrow & [\![x_1(\mathsf{succ}\ x_2)]\!] \subseteq [\![x_1(\mathsf{succ}\ x_2)]\!] \\
\{\mathsf{Int}\} & \subseteq & [\![\mathsf{succ}\ x_2]\!] \\
[\![x_2]\!] & \subseteq & \{\mathsf{Int}\}
\end{array}$$

If such a constraint system is solvable, then it has a least solution. This particular constraint system is indeed solvable, and the least solution is the mapping φ, where

$$\varphi([\![E]\!]) = \{E\}$$
$$\varphi([\![\mathsf{succ}\ x_2]\!]) = \{\mathsf{Int}\}$$
$$\varphi([\![x_1(\mathsf{succ}\ x_2)]\!]) = \varphi(x) = \varphi([\![x_1]\!]) = \varphi([\![x_2]\!]) = \emptyset$$

In the following two sections we present the type system and the safety analysis, and in Section 4 we prove that they accept the same programs.

2 The type system

2.1 Types

Definition 1 Let $\Sigma = \{\rightarrow, \mathsf{Int}, \bot, \top\}$ be the ranked alphabet where $\rightarrow$ is binary and $\mathsf{Int}, \bot, \top$ are nullary. A *type* is a regular tree over Σ. A *path* from the root of such a tree is a string over $\{0, 1\}$, where 0 indicates "left subtree" and 1 indicates "right subtree". □

Definition 2 We represent a type by a *term*, that is, a partial function

$$t : \{0,1\}^* \rightarrow \Sigma$$

with domain $\mathcal{D}(t)$ where t maps each path from the root of the type to the symbol at the end of the path. The set of all such terms is denoted T_Σ. □

Following [6], we finitely represent a term by a so-called *term automaton*, as follows.

Definition 3 A *term automaton* over Σ is a tuple

$$\mathcal{M} \quad = \quad (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

- Q is a finite set of *states*,
- $q_0 \in Q$ is the *start state*,
- $\delta : Q \times \{0,1\} \rightarrow Q$ is a partial function called the *transition function*, and
- $\ell : Q \rightarrow \Sigma$ is a (total) *labeling function*,

such that for any state $q \in Q$, if $\ell(q) \in \{\rightarrow\}$ then

$$\{i \mid \delta(q,i) \text{ is defined}\} \;=\; \{0,1\}$$

and if $\ell(q) \in \{\mathsf{Int}, \bot, \top\}$ then

$$\{i \mid \delta(q,i) \text{ is defined}\} \;=\; \emptyset\ .$$

The partial function δ extends naturally to a partial function

$$\widehat{\delta} : Q \times \{0,1\}^* \rightarrow Q$$

inductively as follows:

$$\begin{array}{rcl} \widehat{\delta}(q,\epsilon) & = & q \\ \widehat{\delta}(q,\alpha i) & = & \delta(\widehat{\delta}(q,\alpha),i)\ ,\ \text{for } i \in \{0,1\}. \end{array}$$

The term represented by $\mathcal{M}$ is the term

$$t_{\mathcal{M}} \;=\; \lambda\alpha.\ell(\widehat{\delta}(q_0,\alpha))\ .$$

□

Intuitively, $t_{\mathcal{M}}(\alpha)$ is determined by starting in the start state q_0 and scanning the input α, following transitions of $\mathcal{M}$ as far as possible. If it is not possible to scan all of α because some i-transition along the way does not exist, then $t_{\mathcal{M}}(\alpha)$ is undefined. If on the other hand $\mathcal{M}$ scans the entire input α and ends up in state q, then $t_{\mathcal{M}}(\alpha) = \ell(q)$.

Types are ordered by the subtype relation $\leq$, as follows.

Definition 4 The *parity* of $\alpha \in \{0,1\}^*$ is the number mod 2 of 0's in α. The parity of α is denoted $\pi\alpha$. A string α is said to be *even* if $\pi\alpha = 0$ and *odd* if $\pi\alpha = 1$. Let $\leq_0$ be the partial order on Σ given by

$$\begin{array}{llll} \bot \leq_0 \rightarrow & \text{and} & \rightarrow \leq_0 \top & \text{and} \\ \bot \leq_0 \mathsf{Int} & \text{and} & \mathsf{Int} \leq_0 \top & \end{array}$$

and let $\leq_1$ be its reverse

$$\begin{array}{llll} \top \leq_1 \rightarrow & \text{and} & \rightarrow \leq_1 \bot & \text{and} \\ \top \leq_1 \mathsf{Int} & \text{and} & \mathsf{Int} \leq_1 \bot & \end{array}$$

For $s,t \in T_\Sigma$, define $s \leq t$ if $s(\alpha) \leq_{\pi\alpha} t(\alpha)$ for all $\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t)$. □

Kozen, Palsberg, and Schwartzbach [6] showed that the relation $\leq$ is equivalent to the order defined by Amadio and Cardelli [1]. The relation $\leq$ is a partial order, and if $s \rightarrow t \leq s' \rightarrow t'$, then $s' \leq s$ and $t \leq t'$ [1, 6].

2.2 Type rules

If E is a λ-term, t is a type, and A is a type environment, *i.e.* a partial function assigning types to variables, then the judgement

$$A \vdash E : t$$

means that E has the type t in the environment A. Formally, this holds when the judgement is derivable using the following six rules:

$$A \vdash \mathsf{0} : \mathsf{Int} \tag{1}$$

$$\frac{A \vdash E : \mathsf{Int}}{A \vdash \mathsf{succ}\ E : \mathsf{Int}} \tag{2}$$

$$A \vdash x : t \quad (\text{provided } A(x) = t) \tag{3}$$

$$\frac{A[x \leftarrow s] \vdash E : t}{A \vdash \lambda x.E : s \to t} \tag{4}$$

$$\frac{A \vdash E : s \to t \quad A \vdash F : s}{A \vdash EF : t} \tag{5}$$

$$\frac{A \vdash E : s \quad s \leq t}{A \vdash E : t} \tag{6}$$

The first five rules are the usual rules for simple types and the last rule is the rule of *subsumption*.

The type system has the subject reduction property, that is, if $A \vdash E : t$ is derivable and E β-reduces to E', then $A \vdash E' : t$ is derivable. This is proved by straightforward induction on the structure of the derivation of $A \vdash E : t$.

2.3 Constraints

Given a λ-term E, the type inference problem can be rephrased in terms of solving a system of type constraints. Assume that E has been α-converted so that all bound variables are distinct. Let X_E be the set of λ-variables x occurring in E, and let Y_E be a set of variables disjoint from X_E consisting of one variable $[\![F]\!]$ for each occurrence of a subterm F of E. (The notation $[\![F]\!]$ is ambiguous because there may be more than one occurrence of F in E. However, it will always be clear from context which occurrence is meant.) We generate the following system of inequalities over $X_E \cup Y_E$:

- for every occurrence in E of a subterm of the form $\mathsf{0}$, the inequality

$$\mathsf{Int} \quad \leq \quad [\![\mathsf{0}]\!] \ ;$$

- for every occurrence in E of a subterm of the form $\mathsf{succ}\ F$, the two inequalities

$$\begin{aligned} \mathsf{Int} &\leq [\![\mathsf{succ}\ F]\!] \\ [\![F]\!] &\leq \mathsf{Int}\ ; \end{aligned}$$

- for every occurrence in E of a subterm of the form $\lambda x.F$, the inequality

$$(x \to [\![F]\!])_{\lambda x.F} \leq [\![\lambda x.F]\!]\ ;$$

- for every occurrence in E of a subterm of the form GH, the inequality

$$[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}\ ;$$

- for every occurrence in E of a λ-variable x, the inequality

$$x \leq [\![x]\!]\ .$$

The subscripts are present to ease notation in Section 4.1; they have no semantic impact and will be explicitly written only in Section 4.1.

Denote by $T(E)$ the system of constraints generated from E in this fashion. The solutions of $T(E)$ over T_Σ correspond to the possible type annotations of E in a sense made precise by Theorem 5.

Let A be a type environment assigning a type to each λ-variable occurring freely in E. If ψ is a function assigning a type to each variable in $X_E \cup Y_E$, we say that ψ *extends* A if A and ψ agree on the domain of A.

Theorem 5 *The judgement $A \vdash E : t$ is derivable if and only if there exists a solution ψ of $T(E)$ extending A such that $\psi([\![E]\!]) = t$. In particular, if E is closed, then E is typable with type t if and only if there exists a solution ψ of $T(E)$ such that $\psi([\![E]\!]) = t$.*

Proof. Similar to the proof of Theorem 2.1 in the journal version of [7], in outline as follows. Given a solution of the constraint system, it is straightforward to construct a derivation of $A \vdash M{:}t$. Conversely, observe that if $A \vdash M{:}t$ is derivable, then there exists a derivation of $A \vdash M{:}t$ such that each use of one of the ordinary rules is followed by exactly one use of the subsumption rule. The approach in for example [14, 11] then gives a set of inequalities of the desired form. □

3 The safety analysis

Following [10, 11], we will use a flow analysis as a basis for a safety analysis. Given a λ-term E, assume that E has been α-converted so that all bound variables are distinct. The set $\mathsf{Abs}(E)$ is the set of subterms of E of the form $\lambda x.F$. The set $\mathsf{Cl}(E)$ is the powerset of $\mathsf{Abs}(E) \cup \{\mathsf{Int}\}$. Safety analysis of a λ-term E can be phrased as solving the following system of constraints over $X_E \cup Y_E$ where type variables range over $\mathsf{Cl}(E)$.

- For every occurrence in E of a subterm of the form $\mathsf{0}$, the constraint

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

- for every occurrence in E of a subterm of the form $\mathsf{succ}\ F$, the two constraints

  $$\{\mathsf{Int}\} \subseteq [\![\mathsf{succ}\ F]\!]$$
  $$[\![F]\!] \subseteq \{\mathsf{Int}\}$$

  where the latter provides a safety check;

- for every occurrence in E of a subterm of the form $\lambda x.F$, the constraint

  $$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![\lambda x.F]\!] ;$$

- for every occurrence in E of a subterm of the form GH, the constraint

  $$[\![G]\!] \subseteq (\mathsf{Abs}(E))_{GH} ;$$

  which provides a safety check;

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

  $$x \subseteq [\![x]\!] ;$$

- for every occurrence in E of a subterm of the form $\lambda x.F$, and for every occurrence in E of a subterm of the form GH, the constraints

  $$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \Rightarrow [\![H]\!] \subseteq x$$
  $$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \Rightarrow [\![F]\!] \subseteq [\![GH]\!] .$$

Again, the subscripts are present to ease notation in Section 4.1; they have no semantic impact and will be explicitly written only in Section 4.1.

Denote by $C(E)$ the system of constraints generated from E in this fashion. A solution of $C(E)$ assigns an element of $\mathsf{Cl}(E)$ to each type variable such that all constraints are satisfied. Solutions are ordered by variable-wise set inclusion. See [10, 12] for a cubic time algorithm that given E computes the least solution of $C(E)$ or decides that none exists. See [9] for a proof of the following subject reduction property. If E β-reduces to E', and $C(E)$ has solution φ, then $C(E')$ also has solution φ.

4 Equivalence

4.1 Deductive Closures

We now introduce two auxiliary constraint systems called $\overline{C}(E)$ and $\overline{T}(E)$. They may be thought of as "deductive closures" of $C(E)$ and $T(E)$. We then show that they are isomorphic (Theorem 9).

Definition 6 For every λ-term E, define $\overline{C}(E)$ to be the smallest set such that:

- The non-conditional constraints of $C(E)$ are members of $\overline{C}(E)$.
- If a constraint $c \Rightarrow K$ is in $C(E)$ and c is in $\overline{C}(E)$, then K is in $\overline{C}(E)$.
- For $s \in X_E \cup Y_E$, if $r \subseteq s$ and $s \subseteq t$ both are in $\overline{C}(E)$, then $r \subseteq t$ is in $\overline{C}(E)$.

Notice that every constraint in $\overline{C}(E)$ is of the form $W \subseteq W'$, where W is of the forms V, $\{\mathsf{Int}\}$, or $(\{\lambda x.F\})_{\lambda x.F}$, and where W' is of the forms V, $\{\mathsf{Int}\}$, or $(\mathsf{Abs}(E))_{GH}$, for $V \in X_E \cup Y_E$. □

Definition 7 For every λ-term E, define $\overline{T}(E)$ to be the smallest set such that:

- $T(E) \subseteq \overline{T}(E)$.
- If $(s \to t)_{\lambda x.F} \leq (s' \to t')_{GH}$ is in $\overline{T}(E)$, then $s' \leq s$ and $t \leq t'$ are in $\overline{T}(E)$.
- For $s \in X_E \cup Y_E$, if $r \leq s$ and $s \leq t$ both are in $\overline{T}(E)$, then $r \leq t$ is in $\overline{T}(E)$.

Notice that every constraint in $\overline{T}(E)$ is of the form $W \leq W'$ where W is of the forms V, Int, or $(V \to V')_{\lambda x.F}$, and where W' is of the form V, Int, or $(V \to V')_{GH}$, for $V, V' \in X_E \cup Y_E$. □

We will now present the definition of two functions $\mathcal{I}$ and $\mathcal{J}$, one from $\overline{C}(E)$ to $\overline{T}(E)$ and one from $\overline{T}(E)$ to $\overline{C}(E)$. After the definition we prove that they are well-defined and each others inverses.

Definition 8 The functions

$$\mathcal{I} : \overline{C}(E) \to \overline{T}(E)$$
$$\mathcal{J} : \overline{T}(E) \to \overline{C}(E)$$

are defined as follows.

$$\begin{aligned} \mathcal{I}(W \subseteq W') &= (\mathcal{L}_{\mathcal{I}}(W) \leq \mathcal{L}_{\mathcal{I}}(W')) \\ \mathcal{J}(W \leq W') &= (\mathcal{L}_{\mathcal{J}}(W) \subseteq \mathcal{L}_{\mathcal{J}}(W')) \end{aligned}$$

where the functions $\mathcal{L}_{\mathcal{I}}$ and $\mathcal{L}_{\mathcal{J}}$ are:

$$\mathcal{L}_{\mathcal{I}}(W) = \begin{cases} W & \text{if } W \in X_E \cup Y_E \\ \mathsf{Int} & \text{if } W = \{\mathsf{Int}\} \\ (x \to [\![F]\!])_{\lambda x.F} & \text{if } W = (\{\lambda x.F\})_{\lambda x.F} \\ ([\![H]\!] \to [\![GH]\!])_{GH} & \text{if } W = (\mathsf{Abs}(E))_{GH} \end{cases}$$

$$\mathcal{L}_{\mathcal{J}}(W) = \begin{cases} W & \text{if } W \in X_E \cup Y_E \\ \{\mathsf{Int}\} & \text{if } W = \mathsf{Int} \\ (\{\lambda x.F\})_{\lambda x.F} & \text{if } W = (x \to [\![F]\!])_{\lambda x.F} \\ (\mathsf{Abs}(E))_{GH} & \text{if } W = ([\![H]\!] \to [\![GH]\!])_{GH} \end{cases}$$

□

Theorem 9 *The sets $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic, and $\mathcal{I}$ and $\mathcal{J}$ are bijections and each others inverses.*

Proof. If $\mathcal{I}$ and $\mathcal{J}$ are well-defined, then clearly they are inverses of each other and thus bijections, so $\overline{C}(E)$ and $\overline{T}(E)$ are isomorphic.

First we show that $\mathcal{I}$ is well-defined. We proceed by induction on the construction of $\overline{C}(E)$. In the base case, consider the non-conditional constraints of $C(E)$ and observe that for those we have:

$\overline{C}(E)$	$\overline{T}(E)$
$\{\mathsf{Int}\} \subseteq [\![\mathsf{0}]\!]$	$\mathsf{Int} \leq [\![\mathsf{0}]\!]$
$\{\mathsf{Int}\} \subseteq [\![\mathsf{succ}\ F]\!]$	$\mathsf{Int} \leq [\![\mathsf{succ}\ F]\!]$
$[\![F]\!] \subseteq \{\mathsf{Int}\}$	$[\![F]\!] \leq \mathsf{Int}$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![\lambda x.F]\!]$	$(x \to [\![F]\!])_{\lambda x.F} \leq [\![\lambda x.F]\!]$
$[\![G]\!] \subseteq (\mathsf{Abs}(E))_{GH}$	$[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$
$x \subseteq [\![x]\!]$	$x \leq [\![x]\!]$

It follows that the lemma holds in the base case.

In the induction step, consider first the constraints

$$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \;\Rightarrow\; [\![H]\!] \subseteq x$$
$$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \;\Rightarrow\; [\![F]\!] \subseteq [\![GH]\!]$$

in $C(E)$ and suppose $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\overline{C}(E)$. By the induction hypothesis, $(x \to [\![F]\!])_{\lambda x.F} \leq [\![G]\!]$ is in $\overline{T}(E)$. Moreover, $[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $T(E)$ and thus also in $\overline{T}(E)$. Hence, $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $\overline{T}(E)$, so also $[\![H]\!] \leq x$ and $[\![F]\!] \leq [\![GH]\!]$ are in $\overline{T}(E)$.

Consider then $r \subseteq s$ and $s \subseteq t$ in $\overline{C}(E)$, and suppose $s \in X_E \cup Y_E$. By the induction hypothesis, $\mathcal{L}_{\mathcal{I}}(r) \leq \mathcal{L}_{\mathcal{I}}(s)$ and $\mathcal{L}_{\mathcal{I}}(s) \leq \mathcal{L}_{\mathcal{I}}(t)$ are in $\overline{T}(E)$. From $s \in X_E \cup Y_E$ we get $\mathcal{L}_{\mathcal{I}}(s) = s$, so $\mathcal{L}_{\mathcal{I}}(r) \leq \mathcal{L}_{\mathcal{I}}(t)$ is in $\overline{T}(E)$.

Then we show that $\mathcal{J}$ is well-defined. We proceed by induction on the construction of $\overline{T}(E)$. In the base case, consider the constraints of $T(E)$. Using the same table as above we observe that $\mathcal{J}$ is well-defined on all these constraints.

In the induction step, consider first $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ in $\overline{T}(E)$. It is sufficient to prove that $\mathcal{L}_{\mathcal{J}}([\![H]\!]) \subseteq \mathcal{L}_{\mathcal{J}}(x)$ and $\mathcal{L}_{\mathcal{J}}([\![F]\!]) \subseteq \mathcal{L}_{\mathcal{J}}([\![GH]\!])$ are in $\overline{C}(E)$, or equivalently, that $[\![H]\!] \subseteq x$ and $[\![F]\!] \subseteq [\![GH]\!]$ are in $\overline{C}(E)$. In $C(E)$ we have

$$\begin{aligned} (\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \;\; &\Rightarrow \;\; [\![H]\!] \subseteq x \\ (\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!] \;\; &\Rightarrow \;\; [\![F]\!] \subseteq [\![GH]\!] \; . \end{aligned}$$

Moreover, $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\overline{C}(E)$. Hence, $[\![H]\!] \subseteq x$ and $[\![F]\!] \subseteq [\![GH]\!]$ are in $\overline{C}(E)$.

Consider then $r \leq s$ and $s \leq t$ in $\overline{T}(E)$, and suppose $s \in X_E \cup Y_E$. By the induction hypothesis, $\mathcal{L}_{\mathcal{J}}(r) \subseteq \mathcal{L}_{\mathcal{J}}(s)$ and $\mathcal{L}_{\mathcal{J}}(s) \subseteq \mathcal{L}_{\mathcal{J}}(t)$ are in $\overline{C}(E)$. From $s \in X_E \cup Y_E$ we get $\mathcal{L}_{\mathcal{J}}(s) = s$, so $\mathcal{L}_{\mathcal{J}}(r) \subseteq \mathcal{L}_{\mathcal{J}}(t)$ is in $\overline{C}(E)$. □

4.2 The equivalence proof

Definition 10 For every λ-term E, let $\mathsf{Tmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to T_Σ and let $\mathsf{Cmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to $\mathsf{Cl}(E)$. □

The following construction is the key to mapping flow information to types.

Definition 11 For every λ-term E, $\varphi \in \mathsf{Cmap}(E)$, and $q_0 \in \mathsf{Cl}(E)$, define the term automaton $\mathcal{A}(E, \varphi, q_0)$ as follows:

$$\mathcal{A}(E, \varphi, q_0) \;\; = \;\; (\mathsf{Cl}(E),\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

- $\delta(\{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}, 0) = \bigcap_{i=1}^{n} \varphi(x_i)$ for $n > 0$
- $\delta(\{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}, 1) = \bigcup_{i=1}^{n} \varphi[\![E_i]\!]$ for $n > 0$
- $$\ell(q) = \begin{cases} \bot & \text{if } q = \emptyset \\ \mathsf{Int} & \text{if } q = \{\mathsf{Int}\} \\ \to & \text{if } q \subseteq \mathsf{Abs}(E) \wedge q \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$

□

Lemma 12 *Suppose* $\varphi \in \mathsf{Cmap}(E)$ *and* $S_1, S_2 \in \mathsf{Cl}(E)$. *If* $S_1 \subseteq S_2$, *then* $t_{\mathcal{A}(E,\varphi,S_1)} \leq t_{\mathcal{A}(E,\varphi,S_2)}$.

Proof. Define the orderings $\subseteq_0$, $\subseteq_1$ on $\mathsf{Cl}(E)$ such that $\subseteq_0$ equals $\subseteq$ and $\subseteq_1$ equals $\supseteq$. The desired conclusion follows immediately from the property that if $\alpha \in \mathcal{D}(t_{\mathcal{A}(E,\varphi,S_1)}) \cap \mathcal{D}(t_{\mathcal{A}(E,\varphi,S_2)})$, then $\widehat{\delta}(S_1,\alpha) \subseteq_{\pi\alpha} \widehat{\delta}(S_2,\alpha)$. This property is proved by straightforward induction on the length of α. □

We can now prove that the type system and the safety analysis accept the same programs.

Theorem 13 *For every λ-term E, the following seven conditions are equivalent:*

1. *$C(E)$ is solvable.*
2. *$T(E)$ is solvable.*
3. *$\overline{T}(E)$ is solvable.*
4. *$\overline{C}(E)$ is solvable.*
5. *$\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$.*
6. *$\overline{T}(E)$ does not contain constraints of the forms $\mathsf{Int} \leq V \to V'$ or $V \to V' \leq \mathsf{Int}$, where $V, V' \in X_E \cup Y_E$.*
7. *The function*

$$\lambda V.\{\ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E)\ \}$$

is the least solution of $C(E)$.

Proof. Given a λ-term E, notice that by the isomorphism of Theorem 9, (5) $\Leftrightarrow$ (6). To show the remaining equivalences, we proceed by proving the implications:

$$(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (4) \Rightarrow (5) \Rightarrow (7) \Rightarrow (1)$$

To prove (1) $\Rightarrow$ (2), suppose $C(E)$ has solution $\varphi \in \mathsf{Cmap}(E)$. Let f be the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and define $\psi \in \mathsf{Tmap}(E)$ by $\psi = f \circ \varphi$. We will show that $T(E)$ has solution ψ. We consider each of the constraints in turn. The cases of the constraints generated from subterms of the forms $\mathsf{0}$, $\mathsf{succ}\ E$, x are immediate, by using Lemma 12. Consider then $\lambda x.F$ and the constraint $x \to [\![F]\!] \leq [\![\lambda x.F]\!]$. By Lemma 12 we get

$$\psi(x) \to \psi([\![F]\!]) = f(\{\lambda x.F\}) \leq \psi([\![\lambda x.F]\!])\ .$$

Consider then GH and the constraint $[\![G]\!] \leq [\![H]\!] \to [\![GH]\!]$. We know that $\varphi([\![G]\!]) \subseteq \mathsf{Abs}(E)$ so there are two cases. Suppose first that $\varphi([\![G]\!]) = \emptyset$. We

then have $\psi([\![G]\!]) = \bot \leq \psi([\![H]\!]) \to \psi([\![GH]\!])$. Consider then the case where $\varphi([\![G]\!]) = \{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}$, for $n > 0$. We then have that $\varphi([\![H]\!]) \subseteq \varphi([\![x_i]\!])$ and $\varphi([\![E_i]\!]) \subseteq \varphi([\![GH]\!])$ for $i \in \{1, \ldots, n\}$. Thus, $\varphi([\![H]\!]) \subseteq \bigcap_{i=1}^{n} \varphi([\![x_i]\!])$ and $\bigcup_{i=1}^{n} \varphi([\![E_i]\!]) \subseteq \varphi([\![GH]\!])$. So, by Lemma 12,

$$\begin{aligned}
\psi([\![G]\!]) &= f(\varphi([\![G]\!])) \\
&= f(\bigcap_{i=1}^{n} \varphi([\![x_i]\!])) \to f(\bigcup_{i=1}^{n} \varphi([\![E_i]\!])) \\
&\leq f(\varphi([\![H]\!])) \to f(\varphi([\![GH]\!])) \\
&= \psi([\![H]\!]) \to \psi([\![GH]\!])
\end{aligned}$$

To prove (2) $\Rightarrow$ (3), suppose $T(E)$ has solution $\psi \in \mathsf{Tmap}(E)$. It is sufficient to show that $\overline{T}(E)$ has solution ψ, and this can be proved by straightforward induction on the construction of $\overline{T}(E)$.

To prove (3) $\Rightarrow$ (4), suppose $\overline{T}(E)$ has solution $\psi \in \mathsf{Tmap}(E)$. Define $\overline{\varphi} \in \mathsf{Cmap}(E)$ as follows:

$$\overline{\varphi}(V) = \begin{cases} \emptyset & \text{if } (\psi(V))(\epsilon) = \bot \\ \{\mathsf{Int}\} & \text{if } (\psi(V))(\epsilon) = \mathsf{Int} \\ \mathsf{Abs}(E) & \text{if } (\psi(V))(\epsilon) = \to \\ \mathsf{Abs}(E) \cup \{\mathsf{Int}\} & \text{if } (\psi(V))(\epsilon) = \top \end{cases}$$

We will show that $\overline{C}(E)$ has solution $\overline{\varphi}$. To see this, let $W \subseteq Z$ be a constraint in $\overline{C}(E)$. If it is of the forms $\{\mathsf{Int}\} \subseteq \{\mathsf{Int}\}$ or $\{\lambda x.F\} \subseteq \mathsf{Abs}(E)$, then it is solvable by all functions, including $\overline{\varphi}$. For the remaining cases, notice that by Theorem 9, $\mathcal{L}_{\mathcal{I}}(W) \leq \mathcal{L}_{\mathcal{I}}(Z)$ is in $\overline{T}(E)$ and thus it has solution ψ. This means that $W \subseteq Z$ cannot be of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$. Suppose then that $W \subseteq Z$ is of one of the remaining forms, that is, $\{\mathsf{Int}\} \subseteq V$, $V \subseteq V'$, $V \subseteq \{\mathsf{Int}\}$, $\{\lambda x.F\} \subseteq V$, $V \subseteq \mathsf{Abs}(E)$, where $V, V' \in X_E \cup Y_E$. We will treat just the first of them, the others are similar. For a constraint of the form $\{\mathsf{Int}\} \subseteq V$, it follows that $\mathsf{Int} \leq V$ is in $\overline{T}(E)$. Since $\overline{T}(E)$ has solution ψ we get that $(\psi(V))(\epsilon) \in \{\mathsf{Int}, \top\}$. Thus, $\overline{\varphi}(V)$ is either $\{\mathsf{Int}\}$ or $\mathsf{Abs}(E) \cup \{\mathsf{Int}\}$, and hence $\{\mathsf{Int}\} \subseteq V$ has solution $\overline{\varphi}$.

To prove (4) $\Rightarrow$ (5), observe that constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$ are not solvable.

To prove (5) $\Rightarrow$ (7), suppose $\overline{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$. Define

$$\varphi' = \lambda V.\{\ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E)\ \}$$

We proceed in four steps, as follows.

- First we show that φ' is a solution of $\overline{C}(E)$. We consider in turn each of the seven possible forms of constraints in $\overline{C}(E)$. Constraints of the forms

$\{\mathsf{Int}\} \subseteq \{\mathsf{Int}\}$ and $\{\lambda x.F\} \subseteq \mathsf{Abs}(E)$ have any solution, including φ'. We are thus left with constraints of the forms $\{\mathsf{Int}\} \subseteq V$, $V \subseteq V'$, $V \subseteq \{\mathsf{Int}\}$, $\{\lambda x.F\} \subseteq V$, $V \subseteq \mathsf{Abs}(E)$, where $V, V' \in X_E \cup Y_E$. We will treat just the first three, since case four is similar to case one and since case five is similar to case three. For a constraint of the form $\{\mathsf{Int}\} \subseteq V$, notice that $\mathsf{Int} \in \varphi'(V)$, so the constraint has solution φ'. For a constraint of the form $V \subseteq V'$, suppose $k \in \varphi'(V)$. Then the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$, and hence the constraint $\{k\} \subseteq V'$ is also in $\overline{C}(E)$. It follows that $k \in \varphi'(V')$. For a constraint of the form $V \subseteq \{\mathsf{Int}\}$, suppose it does not have solution φ'. Hence, there exist $k \in \varphi'(V)$ such that $k \neq \mathsf{Int}$. It follows that the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$, and hence the constraint $\{k\} \subseteq \{\mathsf{Int}\}$ is also in $\overline{C}(E)$, a contradiction.

- Next we show that φ' is the least solution of $\overline{C}(E)$. To do this, let φ be any solution of $\overline{C}(E)$ and suppose $V \in X_E \cup Y_E$. It is sufficient to prove that $\varphi'(V) \subseteq \varphi(V)$. Suppose $k \in \varphi'(V)$. Then the constraint $\{k\} \subseteq V$ is in $\overline{C}(E)$. Since φ is a solution of $\overline{C}(E)$, $k \in \varphi(V)$.

- Next we show that φ' is a solution of $C(E)$. Consider first the non-conditional constraints of $C(E)$. Since these constraints are also members of $\overline{C}(E)$, they have solution φ'. Consider then $\{\lambda x.F\} \subseteq V \Rightarrow K$ in $C(E)$ and suppose $\{\lambda x.F\} \subseteq V$ has solution φ'. Then by the definition of φ', we have that $\{\lambda x.F\} \subseteq V$ is in $\overline{C}(E)$, so also K is in $\overline{C}(E)$, and hence K has solution φ'.

- Finally we show that φ' is the least solution of $C(E)$. To do this, let φ be any solution of $C(E)$. Then φ is also a solution of $\overline{C}(E)$, as can be proved by straightforward induction on the construction of $\overline{C}(E)$. Since φ' is the least solution of $\overline{C}(E)$, φ' is smaller than φ.

To prove (7) $\Rightarrow$ (1), simply notice that since $C(E)$ has a solution, it is solvable. □

Corollary 14 *The type system accepts the same programs as the safety analysis.*

4.3 Algorithms

As corollaries of Theorem 13 we get two cubic time algorithms. Given a λ-term E, first observe that both $\overline{C}(E)$ and $\overline{T}(E)$ can be computed in time $O(n^3)$ where n is the size of E. We can then easily answer the following two questions:

- Question **(safety)**: Is E accepted by safety analysis?
  Algorithm: Check that $\overline{C}(E)$ does not contain constraint of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$.

- Question **(type inference)**: Is E typable? If so, what is an annotation of it?
  Algorithm: Use the safety checking algorithm. If E turns out to be typable, we get an annotation by first calculating the two functions

  $$\varphi' = \lambda V.\{\ k \mid \text{ the constraint } \{k\} \subseteq V \text{ is in } \overline{C}(E)\ \}$$

  and

  $$f = \lambda S.t_{\mathcal{A}(E,\varphi,S)}$$

  and then forming the composition

  $$\psi = f \circ \varphi' \ .$$

  This function ψ is a solution of $T(E)$.

The question of type inference has been open until now. In contrast, it is well-known that flow analysis in the style discussed in this paper can be computed in time $O(n^3)$.

Acknowledgements

The authors thank Mitchell Wand for encouragement and helpful discussions. The results of this paper were obtained while the first author was at Northeastern University, Boston; he is currently hosted by BRICS, Basic Research in Computer Science, Centre of the Danish National Research Foundation.

References

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL'91.

[2] Torben Amtoft. Minimal thunkification. In *Proc. WSA '93*, pages 218–229, 1993.

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

[4] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, December 1991.

[5] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proc. PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, 1993.

[6] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science.* To appear. Also in Proc. POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 419–428, Charleston, South Carolina, January 1993.

[7] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[8] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, 1989.

[9] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems.* To appear. Also in Proc. CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (*LNCS* 787), pages 276–290, Edinburgh, Scotland, April 1994.

[10] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation.* To appear.

[11] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.

[12] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[13] Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 47–87. MIT Press, 1991.

[14] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.