A Type System Equivalent to Flow Analysis

JENS PALSBERG and PATRICK O'KEEFE
Massachusetts Institute of Technology

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 article 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.

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: Constraints, flow analysis

1. INTRODUCTION

1.1 Background

Many program analyses for higher-order languages are based on *flow analysis*, also known as *closure analysis*. Examples include the binding-time analyses for Scheme in the partial evaluators Schism [Consel 1993] and Similix [Bondorf 1991]. 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 [Amtoft 1993; Kuo and Mishra 1989]. 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.

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 article we address an instance of the following fundamental question:

ACM Transactions on Programming Languages and Systems, 17(4):576–599, July 1995. A preliminary version of this article was presented at POPL'95.
Authors' addresses: J. Palsberg, Laboratory for Computer Science, Massachusetts Institute of Technology, Room NE43-340, 545 Technology Square, Cambridge, MA 02139; email: palsberg@theory.lcs.mit.edu; P. O'Keefe, 151 Coolidge Avenue #211, Watertown, MA 02172; email: pmo@world.std.com.

> **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 [1991b] who called it type *recovery*. Later, Palsberg and Schwartzbach [1992; 1995] 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 article, 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. It is based on a flow analysis which in the terminology of Shivers [1991a] is a 0CFA, that is, a 0-level control-flow analysis. Intuitively, it is a flow analysis which for each function merges all environment information.

Many program analyses are based on 0CFA-style analyses; for example, see Sestoft [1991], Wand and Steckler [1994], and Heintze [1994]. Our thesis is that the type system that answers the specific question will in many cases also be the answer to the fundamental question.

Flow-based analyses have the reputation of fitting poorly together with separate compilation because they deal with program points. In contrast, traditional type systems such as that of ML fit well together with separate compilation because one can compute a principal type for each subterm. Our hope is that the type system that answers the specific question above will lead to a better understanding of how to create program analyses that are both modular and have the power of flow-based analyses.

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 [1993], and an $O(n^2)$ algorithm for deciding the subtyping relation has been presented by Kozen et al. [1995]. 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:

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

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 [1992; 1995] proved that the system *without* $\bot$ accepts at most as many programs as safety analysis. In this article 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 $(0\ x)$, $\mathsf{succ}(\lambda x.x)$, and $(\mathsf{succ}\ 0)(x)$; because 0 is not a function, succ cannot be applied to functions, and $(\mathsf{succ}\ 0)$ 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,

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

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

$$(\lambda f.(\lambda x.fI)(f0))I \ : \ \top \ ,$$

where $I = \lambda x.x$. Note that $\top$ is the only type of $(\lambda f.(\lambda x.fI)(f0))I$ because f has to be assigned the type $\top \to \top$. 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. The definitions of such constraint systems will be given in Sections 2.3 and 3. We 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 of E looks as follows:

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

Here, the symbols x, $[\![x_1]\!]$, $[\![x_2]\!]$, $[\![\mathsf{succ}\ x_2]\!]$, $[\![x_1(\mathsf{succ}\ x_2)]\!]$, and $[\![E]\!]$ are type variables. Intuitively, the type variable x is associated with the bound variable x, and the other type variables of the form $[\![\ldots]\!]$ are associated with particular occurrences of subterms. 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. This reflects that, in Amadio and Cardelli's type system, a typable term need not have a least type. For example, the term $\lambda x.x$ has both of the types $\bot \to \bot$ and $\top \to \top$, and these types are incomparable minimal types of $\lambda x.x$. Thus, before type checking and separately compiling a module, we may want to annotate the module boundary explicitly with the types we are interested in. It remains open, however, if modular program analyses can based on Amadio and Cardelli's type system.

The constraint system for safety analysis of E looks as follows:

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

This constraint system uses the same variables as the one above, but now the type variables range over finite sets of occurrences of abstractions and the constant Int.

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

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

This example will be treated in much further detail in Section 5.1.

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. In Section 5 we present two examples. In Section 6 we discuss various extensions, and in Section 7 we outline directions for further work. The reader is encouraged to refer to the examples while reading the other sections.

2. THE TYPE SYSTEM

2.1 Types

We now define the notions of type, term, and term automaton. The idea is that a type is represented by a term which in turn is represented by a term automaton.

Definition 2.1.1. 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 2.1.2. 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 Kozen et al. [1995], we finitely represent a term by a so-called *term automaton*, as follows.

Definition 2.1.3. A *term automaton* over Σ is a tuple

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

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:

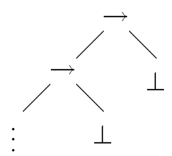
$$\begin{aligned}\widehat{\delta}(q,\epsilon) &= q \\ \widehat{\delta}(q,\alpha i) &= \delta(\widehat{\delta}(q,\alpha),i) \text{ , for } i \in \{0,1\}.\end{aligned}$$

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

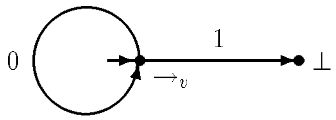
$$t_{\mathcal{M}} = \lambda\alpha.\ell(\widehat{\delta}(q_0,\alpha)) \ .$$

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 $t_{\mathcal{M}}(\alpha) = \ell(q)$.

For example, consider the type



which can be understood as a representation of $\mu v.(v \rightarrow \bot)$. We represent this type by the term t where the domain of t is the infinite regular set $0^* + 0^*1$ and where $t(0^n) = \rightarrow$ and $t(0^n 1) = \bot$ for all $n \geq 0$. The corresponding term automaton is



Thus, infinite paths in a type yield cycles in the corresponding term automaton.

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

Definition 2.1.4. The *parity* of $\alpha \in \{0,1\}^*$ is the number mod 2 of zeros 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

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

Let $\leq_1$ be its reverse

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

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 et al. [1995] showed that the relation $\leq$ is equivalent to the order defined by Amadio and Cardelli [1993]. The relation $\leq$ is a partial order, and if $s \to t \leq s' \to t'$, then $s' \leq s$ and $t \leq t'$ [Amadio and Cardelli 1993; Kozen et al. 1995].

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

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

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

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

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

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

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

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 $X_E \cup Y_E$. Each inequality 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 $V, V' \in X_E \cup Y_E$.

—for every occurrence in E of a subterm of the form 0, the inequality

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

—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

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

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

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

—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. For every λ-term E, let $\mathsf{Tmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to T_Σ. The function $\psi \in \mathsf{Tmap}(E)$ is a *solution* of $T(E)$, if it is a solution of each constraint in $T(E)$. Specifically, for $V, V', V'' \in X_E \cup Y_E$, and occurrences of subterms $\lambda x.F$ and GH in E:

The constraint:	has solution ψ if:
$\mathsf{Int} \leq V$	$\mathsf{Int} \leq \psi(V)$
$V \leq \mathsf{Int}$	$\psi(V) \leq \mathsf{Int}$
$(V \rightarrow V')_{\lambda x.F} \leq V''$	$\psi(V) \rightarrow \psi(V') \leq \psi(V'')$
$V \leq (V' \rightarrow V'')_{GH}$	$\psi(V) \leq \psi(V') \rightarrow \psi(V'')$
$V \leq V'$	$\psi(V) \leq \psi(V')$

The solutions of $T(E)$ correspond to the possible type annotations of E in a sense made precise by Theorem 2.3.1.

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 2.3.1. *The judgment $A \vdash E : t$ 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 Kozen et al. [1994], in outline as follows. Given a solution of the constraint system, it is straightforward to construct a derivation of $A \vdash E{:}t$. Conversely, observe that if $A \vdash E{:}t$ is derivable, then there exists a derivation of $A \vdash E{:}t$ 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, Wand [1991] and Palsberg and Schwartzbach [1992], then gives a set of inequalities of the desired form. $\square$

3. THE SAFETY ANALYSIS

Following Palsberg and Schwartzbach [1992; 1995], 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 occurrences 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.

The constraints in the fourth and sixth items reflect some of the significant differences between the type system and the safety analysis. Intuitively, a constraint of the form $[\![G]\!] \subseteq (\mathsf{Abs}(E))_{GH}$ ensures that G does not evaluate to an integer. This is by the type constraints ensured by the constraint $[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ because the type Int is not a subtype of any function type. The constraints of the forms

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

create a connection between the caller GH and the potential callee $\lambda x.F$. Intuitively, if G evaluates to $\lambda x.F$, then the argument H is bound to x, and the result of evaluating the body F becomes the result of the whole application GH.

Denote by $C(E)$ the system of constraints generated from E in this fashion. For every λ-term E, let $\mathsf{Cmap}(E)$ be the set of total functions from $X_E \cup Y_E$ to $\mathsf{Cl}(E)$. The function $\varphi \in \mathsf{Cmap}(E)$ is a *solution* of $C(E)$, if it is a solution of each constraint in $C(E)$. Specifically, for $V, V', V'' \in X_E \cup Y_E$, and occurrences of subterms $\lambda x.F$ and GH in E:

The constraint:	has solution φ if:
$\{\mathsf{Int}\} \subseteq V$	$\{\mathsf{Int}\} \subseteq \varphi(V)$
$V \subseteq \{\mathsf{Int}\}$	$\varphi(V) \subseteq \{\mathsf{Int}\}$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq V$	$\{\lambda x.F\} \subseteq \varphi(V)$
$V \subseteq (\mathsf{Abs}(E))_{GH}$	$\varphi(V) \subseteq \mathsf{Abs}(E)$
$V \subseteq V'$	$\varphi(V) \subseteq \varphi(V')$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq V \Rightarrow V' \subseteq V''$	$\{\lambda x.F\} \subseteq \varphi(V) \Rightarrow \varphi(V') \subseteq \varphi(V'')$

Solutions are ordered by variablewise set inclusion. See Palsberg and Schwartzbach [1994b; 1995] for a cubic time algorithm that given E computes the least solution of $C(E)$ or decides that none exists. See Palsberg [1995] for a proof technique that enables a proof of the following subject reduction property. If E β-reduces to E', and $C(E)$ is solvable, then $C(E')$ is also solvable.

4. EQUIVALENCE

4.1 Deductive Closures

We now introduce two auxiliary constraint systems called $\bar{C}(E)$ and $\bar{T}(E)$. They may be thought of as "deductive closures" of $C(E)$ and $T(E)$. We then show that they are isomorphic (Theorem 4.1.4). For examples of deductive closures, see Section 5.

Definition 4.1.1. For every λ-term E, define $\bar{C}(E)$ to be the smallest set such that:

—The nonconditional constraints of $C(E)$ are members of $\bar{C}(E)$.

—If a constraint $c \Rightarrow K$ is in $C(E)$, and c is in $\bar{C}(E)$, then K is in $\bar{C}(E)$.

—For $s \in X_E \cup Y_E$, if $r \subseteq s$ and $s \subseteq t$ both are in $\bar{C}(E)$, then $r \subseteq t$ is in $\bar{C}(E)$.

Notice that every constraint in $\bar{C}(E)$ 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 $V \in X_E \cup Y_E$.

For every λ-term E, define also the series $C_n(E)$, for $n \geq 0$, of subsets of $\bar{C}(E)$.

—$C_0(E)$ is the set of nonconditional constraint in $C(E)$.

—For $n \geq 0$, $C_{2n+2}(E)$ is the smallest set such that $C_{2n+2}(E) \supseteq C_{2n+1}(E)$ and such that if a constraint $c \Rightarrow K$ is in $C(E)$ and c is in $C_{2n+1}(E)$, then K is in $C_{2n+2}(E)$.

—For $n \geq 0$, $C_{2n+1}(E)$ is the smallest set such that $C_{2n+1}(E) \supseteq C_{2n}(E)$ and such that for $s \in X_E \cup Y_E$, if $r \leq s$ and $s \leq t$ both are in $C_{2n}(E)$, then $r \leq t$ is in $C_{2n+1}(E)$.

Notice that $C_i(E) \subseteq C_j(E)$ for $0 \leq i \leq j$. Clearly, there exists $N \geq 0$ such that for all $n \geq N$, $C_n(E) = \bar{C}(E)$.

Definition 4.1.2. For every λ-term E, define $\bar{T}(E)$ to be the smallest set such that:

—$T(E) \subseteq \bar{T}(E)$.

—If $(s \rightarrow t)_{\lambda x.F} \leq (s' \rightarrow t')_{GH}$ is in $\bar{T}(E)$, then $s' \leq s$ and $t \leq t'$ are in $\bar{T}(E)$.

—For $s \in X_E \cup Y_E$, if $r \leq s$ and $s \leq t$ both are in $\bar{T}(E)$, then $r \leq t$ is in $\bar{T}(E)$.

Notice that every constraint in $\bar{T}(E)$ 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 $V, V' \in X_E \cup Y_E$.

For every λ-term E, define also the series $T_n(E)$, for $n \geq 0$, of subsets of $\bar{T}(E)$.

—$T_0(E) = T(E)$.

—For $n \geq 0$, $T_{2n+2}(E)$ is the smallest set such that $T_{2n+2}(E) \supseteq T_{2n+1}(E)$ and such that if $(s \to t)_{\lambda x.F} \leq (s' \to t')_{GH}$ is in $T_{2n+1}(E)$, then $s' \leq s$ and $t \leq t'$ are in $T_{2n+2}(E)$.

—For $n \geq 0$, $T_{2n+1}(E)$ is the smallest set such that $T_{2n+1}(E) \supseteq T_{2n}(E)$ and such that for $s \in X_E \cup Y_E$, if $r \leq s$ and $s \leq t$ both are in $T_{2n}(E)$, then $r \leq t$ is in $T_{2n+1}(E)$.

Notice that $T_i(E) \subseteq T_j(E)$ for $0 \leq i \leq j$. Clearly, there exists $N \geq 0$ such that for all $n \geq N$, $T_n(E) = \bar{T}(E)$.

We will now present the definition of two functions $\mathcal{I}$ and $\mathcal{J}$, one from $\bar{C}(E)$ to $\bar{T}(E)$ and one from $\bar{T}(E)$ to $\bar{C}(E)$. After the definition we prove that they are well defined and each other's inverses.

Definition 4.1.3. The functions

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

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 4.1.4. *The sets $\bar{C}(E)$ and $\bar{T}(E)$ are isomorphic, and $\mathcal{I}$ and $\mathcal{J}$ are bijections and each other's inverses.*

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

First we show that $\mathcal{I}$ is well defined, i.e., $\mathcal{I}$ maps each element of $\bar{C}(E)$ to an element of $\bar{T}(E)$. It is sufficient to prove that, for $n \geq 0$, $\mathcal{I}$ maps each element of $C_n(E)$ to an element of $\bar{T}(E)$. We proceed by induction on n. In the base case, consider the constraints of $C_0(E)$, i.e., the nonconditional constraints of $C(E)$, and observe that for those we have:

$C_0(E)$	$T_0(E)$
$\{\mathsf{Int}\} \subseteq [\![0]\!]$	$\mathsf{Int} \leq [\![0]\!]$
$\{\mathsf{Int}\} \subseteq [\![\mathsf{succ}\ F]\!]$	$\mathsf{Int} \leq [\![\mathsf{succ}\ F]\!]$
$[\![F]\!] \subseteq \{\mathsf{Int}\}$	$[\![F]\!] \leq \mathsf{Int}$
$(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![\lambda x.F]\!]$	$(x \to [\![F]\!])_{\lambda x.F} \leq [\![\lambda x.F]\!]$
$[\![G]\!] \subseteq (\mathsf{Abs}(E))_{GH}$	$[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$
$x \subseteq [\![x]\!]$	$x \leq [\![x]\!]$

It follows that the lemma holds in the base case.

In the induction step, consider first $C_{2n+2}(E)$ for some $n \geq 0$. Suppose

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

are in $C(E)$, and suppose $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $C_{2n+1}(E)$. By the induction hypothesis, $(x \to [\![F]\!])_{\lambda x.F} \leq [\![G]\!]$ is in $\bar{T}(E)$. Moreover, $[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $T(E)$ and thus in $\bar{T}(E)$. Hence, $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $\bar{T}(E)$, so $[\![H]\!] \leq x$ and $[\![F]\!] \leq [\![GH]\!]$ are also in $\bar{T}(E)$.

Consider then $C_{2n+1}(E)$ for some $n \geq 0$. Suppose $r \subseteq s$ and $s \subseteq t$ are in $C_{2n}(E)$, 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 $\bar{T}(E)$. 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 $\bar{T}(E)$.

Then we show that $\mathcal{J}$ is well defined, i.e., $\mathcal{J}$ maps each element of $\bar{T}(E)$ to an element of $\bar{C}(E)$. It is sufficient to prove that for $n \geq 0$, $\mathcal{J}$ maps each element of $T_n(E)$ to an element of $\bar{C}(E)$. We proceed by induction on n. In the base case, consider the constraints of $T_0(E)$, that is, 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 $T_{2n+2}(E)$ for some $n \geq 0$. Suppose $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $T_{2n+1}(E)$. 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 $\bar{C}(E)$, or equivalently, that $[\![H]\!] \subseteq x$ and $[\![F]\!] \subseteq [\![GH]\!]$ are in $\bar{C}(E)$. In $C(E)$ we have

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

If $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\bar{C}(E)$, then $[\![H]\!] \subseteq x$ and $[\![F]\!] \subseteq [\![GH]\!]$ are in $\bar{C}(E)$. To see that $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\bar{C}(E)$, notice that in $T(E)$, $(x \to [\![F]\!])_{\lambda x.F}$ occurs only in the constraint $(x \to [\![F]\!])_{\lambda x.F} \leq [\![\lambda x.F]\!]$, and $([\![H]\!] \to [\![GH]\!])_{GH}$ occurs only in the constraint $[\![G]\!] \leq ([\![H]\!] \to [\![GH]\!])_{GH}$. Since $(x \to [\![F]\!])_{\lambda x.F} \leq ([\![H]\!] \to [\![GH]\!])_{GH}$ is in $T_{2n+1}(E)$ we get that also $[\![\lambda x.F]\!] \leq [\![G]\!]$ is in $T_{2n+1}(E)$. Hence, $(x \to [\![F]\!])_{\lambda x.F} \leq [\![G]\!]$ is in $T_{2n+1}(E)$, so by the induction hypothesis, $(\{\lambda x.F\})_{\lambda x.F} \subseteq [\![G]\!]$ is in $\bar{C}(E)$.

Consider then $T_{2n+1}(E)$ for some $n \geq 0$. Suppose $r \leq s$ and $s \leq t$ are in $T_{2n}(E)$, 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 $\bar{C}(E)$. 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 $\bar{C}(E)$. □

4.2 The Equivalence Proof

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

Definition 4.2.1. 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:

$$\delta(\{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}, 0) = \bigcap_{i=1}^{n} \varphi(x_i) \text{ for } n > 0$$

$$\delta(\{\lambda x_1.E_1, \ldots, \lambda x_n.E_n\}, 1) = \bigcup_{i=1}^{n} \varphi[\![E_i]\!] \text{ for } n > 0$$

$$\ell(q) = \begin{cases} \bot & \text{if } q = \emptyset \\ \mathsf{Int} & \text{if } q = \{\mathsf{Int}\} \\ \rightarrow & \text{if } q \subseteq \mathsf{Abs}(E) \wedge q \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$

Lemma 4.2.2. *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. $\Box$

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

Theorem 4.2.3. *For every λ-term E, the following seven conditions are equivalent:*

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

Proof. Given a λ-term E, notice that by the isomorphism of Theorem 4.1.4, (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 4.2.2. Consider then $\lambda x.F$ and the constraint $x \rightarrow [\![F]\!] \leq [\![\lambda x.F]\!]$. By the definition of f and Lemma 4.2.2 we get

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

Consider then GH and the constraint $[\![G]\!] \leq [\![H]\!] \rightarrow [\![GH]\!]$. We know that $\varphi([\![G]\!]) \subseteq \mathsf{Abs}(E)$ so there are two cases. Suppose first that $\varphi([\![G]\!]) = \emptyset$. We then have $\psi([\![G]\!]) = \bot \leq \psi([\![H]\!]) \rightarrow \psi([\![GH]\!])$. 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 4.2.2,

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

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

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

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

We will show that $\bar{C}(E)$ has solution $\bar{\varphi}$. To see this, let $W \subseteq Z$ be a constraint in $\bar{C}(E)$. 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 $\bar{\varphi}$. For the remaining cases, notice that by Theorem 4.1.4 $\mathcal{L}_{\mathcal{I}}(W) \leq \mathcal{L}_{\mathcal{I}}(Z)$ is in $\bar{T}(E)$, 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 $\bar{T}(E)$. Since $\bar{T}(E)$ has solution ψ we get that $(\psi(V))(\epsilon) \in \{\mathsf{Int}, \top\}$. Thus, $\bar{\varphi}(V)$ is either $\{\mathsf{Int}\}$ or $\mathsf{Abs}(E) \cup \{\mathsf{Int}\}$, and hence $\{\mathsf{Int}\} \subseteq V$ has solution $\bar{\varphi}$.

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 $\bar{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$. Define

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

We proceed in four steps, as follows.

—First we show that φ' is a solution of $\bar{C}(E)$. We consider in turn each of the seven possible forms of constraints in $\bar{C}(E)$. 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 $\bar{C}(E)$, and hence the constraint $\{k\} \subseteq V'$ is also in $\bar{C}(E)$. 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 $\bar{C}(E)$, and hence the constraint $\{k\} \subseteq \{\mathsf{Int}\}$ is also in $\bar{C}(E)$, a contradiction.

—Next we show that φ' is the least solution of $\bar{C}(E)$. To do this, let φ be any solution of $\bar{C}(E)$, 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 $\bar{C}(E)$. Since φ is a solution of $\bar{C}(E)$, $k \in \varphi(V)$.

—Next we show that φ' is a solution of $C(E)$. Consider first the nonconditional constraints of $C(E)$. Since these constraints are also members of $\bar{C}(E)$, 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 $\bar{C}(E)$, so also K is in $\bar{C}(E)$; 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 $\bar{C}(E)$, as can be proved by straightforward induction on the construction of $\bar{C}(E)$. Since φ' is the least solution of $\bar{C}(E)$, φ' is smaller than or equal to φ.

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

COROLLARY 4.2.4. *The type system accepts the same programs as the safety analysis.*

The equivalence proof is illustrated in Section 5.

4.3 Algorithms

Theorem 4.2.3 yields two cubic time algorithms. Given a λ-term E, first observe that both $\bar{C}(E)$ and $\bar{T}(E)$ 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 $\bar{C}(E)$ 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

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

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 article can be computed in time $O(n^3)$.

It remains open to define a more direct $O(n^3)$ time type inference algorithm, that is, one that does not use the reduction to the safety-checking problem.

5. EXAMPLES

We illustrate the proof of equivalence with two examples. The λ-terms that will be treated are $\lambda x.x(\mathsf{succ}\ x)$, which was also discussed in Section 1, and $(\lambda x.xx)(\lambda y.y)$.

5.1 $\lambda x.x(\mathsf{succ}\ x)$

As in Section 1, 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 $E = \lambda x.x_1(\mathsf{succ}\ x_2)$. Notice that $\mathsf{Abs}(E) = \{E\}$. As stated in Section 1, $C(E)$ looks as follows:

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

The deductive closure $\bar{C}(E)$ is shown in Table I. Intuitively, it is obtained by observing that no constraint matches the condition of any of the two conditional constraints, and by using the transitivity rule twice.

As also stated in Section 1, $T(E)$ looks as follows:

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

The deductive closure $\bar{T}(E)$ is shown in Table I. It is obtained by using the transitivity rule twice.

It can be verified by inspection that Theorem 4.1.4 is true for E, i.e., $\bar{C}(E)$ and $\bar{T}(E)$ are isomorphic. Moreover, $\bar{C}(E)$ does not contain constraints of the forms $\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$, and $\bar{T}(E)$ does not contain constraints of the forms $\mathsf{Int} \leq V \to V'$ or $V \to V' \leq \mathsf{Int}$, where $V, V' \in X_E \cup Y_E$.

Table I. Two Deductive Closures

$\bar{C}(E)$	$\bar{T}(E)$
$\{E\} \subseteq [\![E]\!]$	$x \to [\![x_1(\mathsf{succ}\ x_2)]\!] \leq [\![E]\!]$
$[\![x_1]\!] \subseteq \{E\}$	$[\![x_1]\!] \leq [\![\mathsf{succ}\ x_2]\!] \to [\![x_1(\mathsf{succ}\ x_2)]\!]$
$x \subseteq [\![x_1]\!]$	$x \leq [\![x_1]\!]$
$x \subseteq [\![x_2]\!]$	$x \leq [\![x_2]\!]$
$\{\mathsf{Int}\} \subseteq [\![\mathsf{succ}\ x_2]\!]$	$\mathsf{Int} \leq [\![\mathsf{succ}\ x_2]\!]$
$[\![x_2]\!] \subseteq \{\mathsf{Int}\}$	$[\![x_2]\!] \leq \mathsf{Int}$
$x \subseteq \{E\}$	$x \leq [\![\mathsf{succ}\ x_2]\!] \to [\![x_1(\mathsf{succ}\ x_2)]\!]$
$x \subseteq \{\mathsf{Int}\}$	$x \leq \mathsf{Int}$

We are now ready to focus on the equivalence proof. We will go through that proof and illustrate each construction in the case of E.

The equivalence proof may be summarized as follows. The proof demonstrates several instances of how to transform a solution of one constraint system into a solution of an other constraint system. It may be helpful to think of a step as transforming the output of the previous step, as follows. The starting point is a solution φ of $C(E)$. This φ is then transformed into a solution ψ of $T(E)$. This ψ is also a solution of $\bar{T}(E)$, and it is then transformed into a solution $\bar{\varphi}$ of $\bar{C}(E)$. Having such a solution implies that certain constraints are not in $\bar{C}(E)$ (condition 5), and also that certain constraints are not in $\bar{T}(E)$ (condition 6). The function $\bar{\varphi}$ need not be a solution of $C(E)$, but we can construct the least solution of $C(E)$ from $\bar{C}(E)$. In one picture, the transformations go as follows:

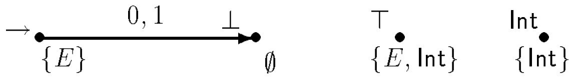
$$\varphi \;\to\; \psi \;\to\; \bar{\varphi}$$

We will now follow a particular φ as it tours this diagram.

For a starting point, we choose the least solution φ of $C(E)$ which was also stated in Section 1. It looks as follows:

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

To get the solution ψ of $T(E)$, we need to construct the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and compose it with φ. The automaton $\mathcal{A}(E, \varphi, S)$ can be illustrated as follows:



Notice that we have not pointed to the start state; it is a parameter of the specification. The illustration gives both the name and the label of each state. There are just two transitions, both from the state $\{E\}$ to the state $\emptyset$. Observe that

$$t_{\mathcal{A}(E,\varphi,S)} = \begin{cases} \perp \to \perp & \text{if } S = \{E\} \\ \mathsf{Int} & \text{if } S = \{\mathsf{Int}\} \\ \perp & \text{if } S = \emptyset \end{cases}$$

We can then obtain the mapping ψ:

$$\psi([\![E]\!]) = \perp \to \perp$$

$$\begin{aligned} \psi(\llbracket \mathsf{succ}\ x_2 \rrbracket) &= \mathsf{Int} \\ \psi(\llbracket x_1(\mathsf{succ}\ x_2) \rrbracket) &= \psi(x) = \psi(\llbracket x_1 \rrbracket) = \psi(\llbracket x_2 \rrbracket) = \bot \end{aligned}$$

It can be verified by inspection that ψ is a solution of $T(E)$ and $\bar{T}(E)$.

To get the solution $\bar{\varphi}$ of $\bar{C}(E)$, we need to compute $(\psi(V))(\epsilon)$ for every $V \in X_E \cup Y_E$. For example,

$$(\psi(\llbracket E \rrbracket))(\epsilon) = (\bot \to \bot)(\epsilon) = \to$$

Plugging this into the definition of $\bar{\varphi}$ yields:

$$\begin{aligned} \bar{\varphi}(\llbracket E \rrbracket) &= \{E\} \\ \bar{\varphi}(\llbracket \mathsf{succ}\ x_2 \rrbracket) &= \{\mathsf{Int}\} \\ \bar{\varphi}(\llbracket x_1(\mathsf{succ}\ x_2) \rrbracket) &= \bar{\varphi}(x) = \bar{\varphi}(\llbracket x_1 \rrbracket) = \bar{\varphi}(\llbracket x_2 \rrbracket) = \emptyset \end{aligned}$$

So, $\varphi = \bar{\varphi}$, and it can be verified by inspection that $\bar{\varphi}$ is a solution of $\bar{C}(E)$.

Finally, to construct φ' where

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

notice that the constraints in $\bar{C}(E)$ that have the form $\{k\} \subseteq V$ where $V \in X_E \cup Y_E$ are:

$$\begin{aligned} \{E\} &\subseteq \llbracket E \rrbracket \\ \{\mathsf{Int}\} &\subseteq \llbracket \mathsf{succ}\ x_2 \rrbracket \end{aligned}$$

So, $\varphi = \bar{\varphi} = \varphi'$, and hence φ' is the least solution of $C(E)$.

It is only in special cases that $\varphi = \bar{\varphi}$. Next we consider a slightly more complicated example where this does not occur.

5.2 $(\lambda x.xx)(\lambda y.y)$

We give each of the two occurrences of x a label so that the λ-term instead reads $(\lambda x.x_1x_2)(\lambda y.y)$. For brevity, let $E = (\lambda x.x_1x_2)(\lambda y.y)$. Notice that $\mathsf{Abs}(E) = \{\lambda x.x_1x_2, \lambda y.y\}$. The constraint system $C(E)$ looks as follows:

$$\begin{array}{ll}
\lambda x.x_1x_2 & \{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \\
\lambda y.y & \{\lambda y.y\} \subseteq \llbracket \lambda y.y \rrbracket \\
E & \llbracket \lambda x.x_1x_2 \rrbracket \subseteq \mathsf{Abs}(E) \\
x_1x_2 & \llbracket x_1 \rrbracket \subseteq \mathsf{Abs}(E) \\
x_1 & x \subseteq \llbracket x_1 \rrbracket \\
x_2 & x \subseteq \llbracket x_2 \rrbracket \\
y & y \subseteq \llbracket y \rrbracket \\
\begin{array}{l} x_1x_2 \text{ and} \\ \lambda x.x_1x_2 \end{array} & \left\{ \begin{array}{l} \{\lambda x.x_1x_2\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket x_2 \rrbracket \subseteq x \\ \{\lambda x.x_1x_2\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket \subseteq \llbracket x_1x_2 \rrbracket \end{array} \right. \\
\begin{array}{l} x_1x_2 \text{ and} \\ \lambda y.y \end{array} & \left\{ \begin{array}{l} \{\lambda y.y\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket x_2 \rrbracket \subseteq y \\ \{\lambda y.y\} \subseteq \llbracket x_1 \rrbracket \Rightarrow \llbracket y \rrbracket \subseteq \llbracket x_1x_2 \rrbracket \end{array} \right. \\
\begin{array}{l} E \text{ and} \\ \lambda x.x_1x_2 \end{array} & \left\{ \begin{array}{l} \{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket \lambda y.y \rrbracket \subseteq x \\ \{\lambda x.x_1x_2\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket x_1x_2 \rrbracket \subseteq \llbracket E \rrbracket \end{array} \right. \\
\begin{array}{l} E \text{ and} \\ \lambda y.y \end{array} & \left\{ \begin{array}{l} \{\lambda y.y\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket \lambda y.y \rrbracket \subseteq y \\ \{\lambda y.y\} \subseteq \llbracket \lambda x.x_1x_2 \rrbracket \Rightarrow \llbracket y \rrbracket \subseteq \llbracket E \rrbracket \end{array} \right.
\end{array}$$

Table II. Two Deductive Closures

$\bar{C}(E)$	$\bar{T}(E)$
$\{\lambda x.x_1x_2\} \subseteq [\![\lambda x.x_1x_2]\!]$	$x \to [\![x_1x_2]\!] \leq [\![\lambda x.x_1x_2]\!]$
$\{\lambda x.x_1x_2\} \subseteq \mathsf{Abs}(E)$	$x \to [\![x_1x_2]\!] \leq [\![\lambda y.y]\!] \to [\![E]\!]$
$[\![\lambda x.x_1x_2]\!] \subseteq \mathsf{Abs}(E)$	$[\![\lambda x.x_1x_2]\!] \leq [\![\lambda y.y]\!] \to [\![E]\!]$
$\{\lambda y.y\} \subseteq [\![\lambda y.y]\!]$	$y \to [\![y]\!] \leq [\![\lambda y.y]\!]$
$\{\lambda y.y\} \subseteq x$	$y \to [\![y]\!] \leq x$
$\{\lambda y.y\} \subseteq [\![x_1]\!]$	$y \to [\![y]\!] \leq [\![x_1]\!]$
$\{\lambda y.y\} \subseteq \mathsf{Abs}(E)$	$y \to [\![y]\!] \leq [\![x_2]\!] \to [\![x_1x_2]\!]$
$\{\lambda y.y\} \subseteq [\![x_2]\!]$	$y \to [\![y]\!] \leq [\![x_2]\!]$
$\{\lambda y.y\} \subseteq y$	$y \to [\![y]\!] \leq y$
$\{\lambda y.y\} \subseteq [\![y]\!]$	$y \to [\![y]\!] \leq [\![y]\!]$
$\{\lambda y.y\} \subseteq [\![x_1x_2]\!]$	$y \to [\![y]\!] \leq [\![x_1x_2]\!]$
$\{\lambda y.y\} \subseteq [\![E]\!]$	$y \to [\![y]\!] \leq [\![E]\!]$
$[\![\lambda y.y]\!] \subseteq x$	$[\![\lambda y.y]\!] \leq x$
$[\![\lambda y.y]\!] \subseteq [\![x_1]\!]$	$[\![\lambda y.y]\!] \leq [\![x_1]\!]$
$[\![\lambda y.y]\!] \subseteq \mathsf{Abs}(E)$	$[\![\lambda y.y]\!] \leq [\![x_2]\!] \to [\![x_1x_2]\!]$
$[\![\lambda y.y]\!] \subseteq [\![x_2]\!]$	$[\![\lambda y.y]\!] \leq [\![x_2]\!]$
$[\![\lambda y.y]\!] \subseteq y$	$[\![\lambda y.y]\!] \leq y$
$[\![\lambda y.y]\!] \subseteq [\![y]\!]$	$[\![\lambda y.y]\!] \leq [\![y]\!]$
$[\![\lambda y.y]\!] \subseteq [\![x_1x_2]\!]$	$[\![\lambda y.y]\!] \leq [\![x_1x_2]\!]$
$[\![\lambda y.y]\!] \subseteq [\![E]\!]$	$[\![\lambda y.y]\!] \leq [\![E]\!]$
$x \subseteq [\![x_1]\!]$	$x \leq [\![x_1]\!]$
$x \subseteq \mathsf{Abs}(E)$	$x \leq [\![x_2]\!] \to [\![x_1x_2]\!]$
$x \subseteq [\![x_2]\!]$	$x \leq [\![x_2]\!]$
$x \subseteq y$	$x \leq y$
$x \subseteq [\![y]\!]$	$x \leq [\![y]\!]$
$x \subseteq [\![x_1x_2]\!]$	$x \leq [\![x_1x_2]\!]$
$x \subseteq [\![E]\!]$	$x \leq [\![E]\!]$
$[\![x_1]\!] \subseteq \mathsf{Abs}(E)$	$[\![x_1]\!] \leq [\![x_2]\!] \to [\![x_1x_2]\!]$
$[\![x_2]\!] \subseteq y$	$[\![x_2]\!] \leq y$
$[\![x_2]\!] \subseteq [\![y]\!]$	$[\![x_2]\!] \leq [\![y]\!]$
$[\![x_2]\!] \subseteq [\![x_1x_2]\!]$	$[\![x_2]\!] \leq [\![x_1x_2]\!]$
$[\![x_2]\!] \subseteq [\![E]\!]$	$[\![x_2]\!] \leq [\![E]\!]$
$y \subseteq [\![y]\!]$	$y \leq [\![y]\!]$
$y \subseteq [\![x_1x_2]\!]$	$y \leq [\![x_1x_2]\!]$
$y \subseteq [\![E]\!]$	$y \leq [\![E]\!]$
$[\![y]\!] \subseteq [\![x_1x_2]\!]$	$[\![y]\!] \leq [\![x_1x_2]\!]$
$[\![y]\!] \subseteq [\![E]\!]$	$[\![y]\!] \leq [\![E]\!]$
$[\![x_1x_2]\!] \subseteq [\![E]\!]$	$[\![x_1x_2]\!] \leq [\![E]\!]$

To the left of the constraints, we have indicated from where they arise.

The constraint system $T(E)$ looks as follows:

From $\lambda x.x_1x_2$	$x \to [\![x_1x_2]\!] \leq [\![\lambda x.x_1x_2]\!]$
From $\lambda y.y$	$y \to [\![y]\!] \leq [\![\lambda y.y]\!]$
From E	$[\![\lambda x.x_1x_2]\!] \leq [\![\lambda y.y]\!] \to [\![E]\!]$
From x_1x_2	$[\![x_1]\!] \leq [\![x_2]\!] \to [\![x_1x_2]\!]$
From x_1	$x \leq [\![x_1]\!]$
From x_2	$x \leq [\![x_2]\!]$
From y	$y \leq [\![y]\!]$

The deductive closures $\bar{C}(E)$ and $\bar{T}(E)$ are shown in Table II.

It can be verified by inspection that Theorem 4.1.4 is true for E, i.e., $\bar{C}(E)$ and $\bar{T}(E)$ are isomorphic. Moreover, $\bar{C}(E)$ does not contain constraints of the forms

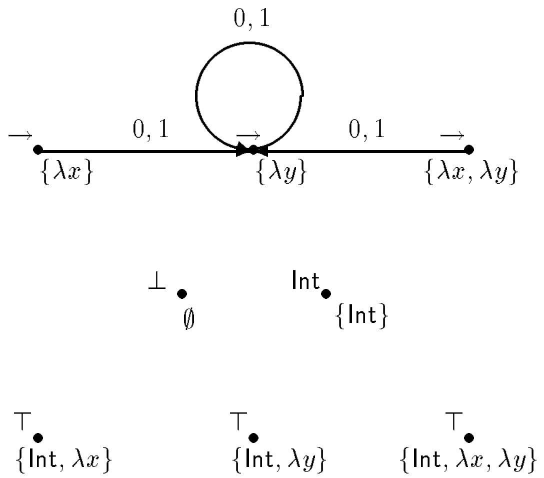
$\{\mathsf{Int}\} \subseteq \mathsf{Abs}(E)$ or $\{\lambda x.F\} \subseteq \{\mathsf{Int}\}$, and $\bar{T}(E)$ does not contain constraints of the forms $\mathsf{Int} \leq V \to V'$ or $V \to V' \leq \mathsf{Int}$, where $V, V' \in X_E \cup Y_E$.

As for the previous example, we will now go through the equivalence proof and illustrate each construction in the case of E.

As starting point, we choose the least solution φ of $C(E)$. It looks as follows:

$$\varphi(V) = \begin{cases} \{\lambda x.x_1x_2\} & \text{if } V = [\![\lambda x.x_1x_2]\!] \\ \{\lambda y.y\} & \text{otherwise} \end{cases}$$

To get the solution ψ of $T(E)$, we need to construct the function $\lambda S.t_{\mathcal{A}(E,\varphi,S)}$ and compose it with φ. The automaton $\mathcal{A}(E,\varphi,S)$ can be illustrated as follows:



As before, notice that we have not pointed to the start state; it is a parameter of the specification. Notice also that we have abbreviated the names of some of the states. Observe that $t_{\mathcal{A}(E,\varphi,S)} = \mu\alpha.\alpha \to \alpha$, if S is a nonempty subset of $\{\lambda x.x_1x_2, \lambda y.y\}$. We can then obtain the mapping ψ. It is a constant function:

$$\psi(V) = \mu\alpha.\alpha \to \alpha$$

It can be verified by inspection that ψ is a solution of $T(E)$ and $\bar{T}(E)$.

As an aside, note that although $\psi(V)$ is an infinite tree for all V, there are other solutions of $T(E)$ and $\bar{T}(E)$ where all the involved types are finite. For example, consider the solution ψ' where

$$\begin{aligned} \psi'([\![\lambda x.x_1x_2]\!]) &= (\top \to \top) \to \top \\ \psi'([\![\lambda y.y]\!]) &= \psi'(x) = \psi'([\![x_1]\!]) = \top \to \top \\ \psi'([\![x_2]\!]) &= \psi'(y) = \psi'([\![y]\!]) = \psi'([\![x_1x_2]\!]) \\ &= \psi'([\![E]\!]) = \top \ . \end{aligned}$$

To get the solution $\bar{\varphi}$ of $\bar{C}(E)$, observe that

$$(\psi(V))(\epsilon) = \to \ .$$

Plugging this into the definition of $\bar{\varphi}$ yields that $\bar{\varphi}$ is a constant function:

$$\bar{\varphi}(V) = \mathsf{Abs}(E)$$

Notice that $\varphi \neq \bar{\varphi}$. It can be verified by inspection that $\bar{\varphi}$ is a solution of $\bar{C}(E)$.

Finally, to construct φ' where

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

notice that the constraints in $\bar{C}(E)$ that have the form $\{k\} \subseteq V$ where $V \in X_E \cup Y_E$ are:

$$\begin{array}{l}
\{\lambda x.x_1x_2\} \subseteq [\![\lambda x.x_1x_2]\!] \\
\{\lambda y.y\} \subseteq [\![\lambda y.y]\!] \\
\{\lambda y.y\} \subseteq x \\
\{\lambda y.y\} \subseteq [\![x_1]\!] \\
\{\lambda y.y\} \subseteq [\![x_2]\!] \\
\{\lambda y.y\} \subseteq y \\
\{\lambda y.y\} \subseteq [\![y]\!] \\
\{\lambda y.y\} \subseteq [\![x_1x_2]\!] \\
\{\lambda y.y\} \subseteq [\![E]\!]
\end{array}$$

So, $\varphi = \varphi'$, and hence φ' is the least solution of $C(E)$.

6. EXTENSIONS

Various extensions to the type system and flow analysis have equivalent typability and safety problems. We now show an example of such an extension: the conditional construct $\mathsf{if0}$. For simplicity, we consider $\mathsf{if0}$ rather than a more usual if, to avoid introducing booleans. Thus, the set of types T_Σ and the abstract domain $\mathsf{Cl}(E)$ for the safety analysis remain the same.

The syntax for the extension of our example language is:

$$E \ ::= \ \ldots \mid \mathsf{if0}\ E_1E_2E_3$$

The intention is that if E_1 evaluates to $\mathsf{0}$, then E_2 is evaluated; if E_1 evaluates to a nonzero integer, then E_3 is evaluated; and if E_1 evaluates to a noninteger, then an error occurs. We need one new type rule:

$$\frac{A \vdash E : \mathsf{Int} \quad A \vdash E_2 : t \quad A \vdash E_3 : t}{A \vdash \mathsf{if0}\ E_1E_2E_3 : t} \tag{7}$$

As before, we can rephrase the type inference problem in terms of solving a system of type constraints. We need three new type constraints:

—for every occurrence of a subterm of the form $\mathsf{if0}\ E_1E_2E_3$, we generate the three inequalities

$$\begin{array}{rcl}
[\![E_1]\!] & \leq & \mathsf{Int} \\
[\![E_2]\!] & \leq & [\![\mathsf{if0}\ E_1E_2E_3]\!] \\
[\![E_3]\!] & \leq & [\![\mathsf{if0}\ E_1E_2E_3]\!]\ .
\end{array}$$

It is straightforward to check that Theorem 2.3.1 remains true, i.e., the solutions of the constraint system correspond to the possible type annotations.

We need three new safety constraints:

—for every occurrence of a subterm of the form $\mathsf{if0}\ E_1 E_2 E_3$, we generate the three constraints

$$\begin{aligned} [\![E_1]\!] &\subseteq \{\mathsf{Int}\} \\ [\![E_2]\!] &\subseteq [\![\mathsf{if0}\ E_1 E_2 E_3]\!] \\ [\![E_3]\!] &\subseteq [\![\mathsf{if0}\ E_1 E_2 E_3]\!] \ . \end{aligned}$$

It is straightforward to check that Theorem 4.1.4 and Theorem 4.2.3 remain true, i.e., $\bar{C}(E)$ are $\bar{T}(E)$ are isomorphic, and the type system and the safety analysis accept the same programs. Moreover, the safety analysis and type inference algorithms remain the same.

The addition of polymorphic let, products, sums, and atomic subtypes with coercions should also be straightforward. Dynamic or soft typing systems are also candidates for formulating equivalent type and flow analysis systems.

A different challenge is to formulate a type system equivalent to the safety analysis for object-oriented languages presented in Palsberg and Schwartzbach [1994b]. Yet another challenge is to find two equivalent binding-time analyses, one based on type systems and one based on flow analysis. Results in this direction were presented in Palsberg and Schwartzbach [1994a].

7. CONCLUSION

We have described a type system and a flow analysis and proved that the corresponding typability and safety problems are equivalent. We also obtained a cubic time algorithm for typability. This problem has been open since the type system was first presented by Amadio and Cardelli in 1991 [Amadio and Cardelli 1993].

For a given program, the system of type constraints and the system of flow constraints are radically different. For the particular language studied in this article, however, we demonstrated that the deductive closures of those systems are isomorphic. This property does not seem to be either a necessary or a sufficient condition for the equivalence result, but in itself it suggests a close relationship between the systems.

Tang and Jouvelot [1994] have demonstrated that type analysis and flow analysis can be combined in a single framework. The type system part of their approach is that of simple types. A challenge is to extend their framework such that the type system part is that of this article. Wand and Steckler [1994] presented a framework for proving correctness of flow-based compiler optimizations. A challenge is to investigate if their framework can be simplified when the flow analysis is replaced by, for example, the framework of Tang and Jouvelot [1994] or possibly an extended one.

An example of an area in which a relationship between a typing problem and a flow problem may be helpful is debugging and explaining type inferencing results for end users. A flow analysis point of view might provide a concrete illustration of why the type inferencer produced a particular type assignment or typing error [Wand 1986].

The two systems considered in this article use inequalities, that is, subtyping in the type system and set inclusion in the flow analysis. One might consider changing the inequalities to equalities and look for an equivalence result similar to the one in this article. On the type system side, this would result in a simply typed lambda-calculus with a type inference algorithm based on unification. On the flow analysis side, it would result in an analysis resembling the one of Bondorf and Jørgensen [1993]. Current work addresses obtaining an equivalence between two such systems (personal communication, N. Heintze, 1994).

In conclusion, we find that a type system and a flow analysis can in some cases be equivalent ways of looking at the same problem.

Acknowledgments

We thank Mitchell Wand for encouragement and helpful discussions. We also thank Torben Amtoft and the anonymous referees for helpful comments on a draft of the article. The results of this article were obtained while the first author was at Northeastern University, Boston.

REFERENCES

Amadio, R. M. and Cardelli, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15,* 4, 575–631. Also in Proc. POPL'91.

Amtoft, T. 1993. Minimal thunkification. In *Proceedings of WSA'93, 3rd International Workshop on Static Analysis.* Lecture Notes in Computer Science, vol. 724. Springer-Verlag, Berlin, 218–229.

Bondorf, A. 1991. Automatic autoprojection of higher order recursive equations. *Sci. Comput. Program. 17,* 1–3 (Dec.), 3–34.

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

Consel, C. 1993. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of PEPM'93, 2nd ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM, New York, 145–154.

Heintze, N. 1994. Set-based analysis of ML programs. In *Proceedings of the ACM Conference on LISP and Functional Programming.* ACM, New York, 306–317.

Kozen, D., Palsberg, J., and Schwartzbach, M. I. 1994. Efficient inference of partial types. *J. Comput. Syst. Sci. 49,* 2, 306–324. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

Kozen, D., Palsberg, J., and Schwartzbach, M. I. 1995. Efficient recursive subtyping. In *Mathematical Structures in Computer Science.* To be published. Also in Proc. POPL'93, 20th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 419–428, Charleston, South Carolina, January 1993.

Kuo, T.-M. and Mishra, P. 1989. Strictness analysis: A new perspective based on type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* ACM, New York, 260–272.

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

Palsberg, J. and Schwartzbach, M. I. 1992. Safety analysis versus type inference for partial types. *Inf. Process. Lett. 43,* 175–180.

Palsberg, J. and Schwartzbach, M. I. 1994a. Binding-time analysis: Abstract interpretation versus type inference. In *Proceedings of ICCL'94, 5th IEEE International Conference on Computer Languages.* IEEE, New York, 289–298.

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

Palsberg, J. and Schwartzbach, M. I. 1995. Safety analysis versus type inference. *Inf. Comput. 118,* 1, 128–141.

Sestoft, P. 1991. Analysis and efficient implementation of functional programs. Ph.D. thesis, DIKU, Univ. of Copenhagen, Denmark.

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

SHIVERS, O. 1991b. Data-flow analysis and type recovery in Scheme. In *Topics in Advanced Language Implementation*, P. Lee, Ed. MIT Press, Cambridge, Mass., 47–87.

TANG, Y. M. AND JOUVELOT, P. 1994. Separate abstract interpretation for control-flow analysis. In *Proceedings of TACS'94, Theoretical Aspects of Computing Sofware*. Lecture Notes in Computer Science, vol. 789. Springer-Verlag, Berlin, 224–243.

WAND, M. 1986. Finding the source of type errors. In the *13th Symposium on Principles of Programming Languages*. ACM, New York, 38–43.

WAND, M. 1991. Type inference for record concatenation and multiple inheritance. *Inf. Comput. 93,* 1, 1–15.

WAND, M. AND STECKLER, P. 1994. Selective and lightweight closure conversion. In *Proceedings of POPL'94, 21st Annual Symposium on Principles of Programming Languages*. ACM, New York, 434–445.

Received February 1995; revised May 1995; accepted June 1995