Information Processing Letters 43 (1992) 175–180.

Safety Analysis versus Type Inference for Partial Types

Jens Palsberg palsberg@daimi.aau.dk
Michael I. Schwartzbach mis@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark

Abstract

Safety analysis is an algorithm for determining if a term in an untyped lambda calculus with constants is *safe*, i.e., if it does not cause an error during evaluation. We prove that safety analysis accepts strictly more safe lambda terms than does type inference for Thatte's partial types.

1 Introduction

We will compare two techniques for analyzing the *safety* of terms in an untyped lambda calculus with constants, see figure 1. The safety we are concerned with is the absence of those run-time errors that arise from the misuse of constants. In this paper we consider just the two constants 0 and succ. They can be misused either by applying a number to an argument, or by applying succ to an abstraction. Safety is undecidable so any analysis algorithm must reject some safe programs.

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

Figure 1: The lambda calculus.

One way of achieving a safety guarantee is to perform *type inference* (TI), because "well-typed programs cannot go wrong". Two examples of type systems for which type inference algorithms exist are those of simple types [4] and Thatte's partial types [9, 5, 3]. Note that any term that has a simple type also has a partial type.

Another way of achieving a safety guarantee is the analysis method of the present authors, simply called *safety analysis* (SA) [6]. In a previous paper [6], we proved that SA accepts strictly more safe terms that does TI for *simple* types. This paper improves our result by proving that SA accepts strictly more safe terms than does TI for *partial* types.

In the following section we recall the definitions of type inference for partial types and of safety analysis. In section 3 we prove our result.

2 The Formal Systems

Both TI and SA can be described as four-step processes, as follows. First, the lambda term is α-converted so that every λ-bound variable is distinct. This means that every abstraction $\lambda x.E$ can be denoted by the unique token λx. Second, a type variable $[\![E]\!]$ is assigned to every subterm E. Third, a finite collection of constraints over these variables is generated from the syntax. Finally, these constraints are solved. This presentation of type inference is due to Wand [10]. Polymorphic let could be treated by both analyses by doing syntactic expansion.

Type inference and safety analysis employ constraints over different domains. In type inference for partial types, type variables range over the following types:

$$\tau ::= \Omega \mid \mathsf{Int} \mid \tau_1 \to \tau_2$$

Types are partially ordered as follows:

$$\begin{array}{l} \tau \leq \Omega \\ \tau \to \ \sigma \leq \tau' \to \sigma' \iff \tau' \leq \tau \wedge \ \sigma \leq \sigma' \\ \mathsf{Int} \leq \mathsf{Int} \end{array}$$

Thus, partial types have a largest type Ω and involve the usual contravariant rule for arrow types. Typical inclusions are $\Omega \to \Omega \leq \Omega$, $\Omega \to \Omega \leq (\Omega \to \Omega) \to \Omega$, and $\mathsf{Int} \leq \Omega$. Intuitively, $\tau \leq \sigma$ allows a coercion from τ to σ that forgets some type structure. The type Ω contains only the information "well-typed".

The constraints are generated inductively in the syntax, see figure 2. A cubic time algorithm for solving such constraints has been presented by Kozen and the present authors [3]. It improved the exponential time algorithm of O'Keefe and Wand [5]. If no solution exists, then the program is not typable. Note that, if the inequalities are strengthened to equalities, then we get the constraints of type inference for simple types. Such equalities are solvable in linear time.

As an example of a term that does not have a simple type but does have a partial type, consider $\lambda f.(K(fI)(f\mathsf{0}))$ where K and I are the usual combinators. This term has type $(\Omega \to \Omega) \to \Omega$ since both I and $\mathsf{0}$ have type Ω.

Phrase:	Constraint:
$\lambda x.E$	$[\![\lambda x.E]\!] \geq [\![x]\!] \rightarrow [\![E]\!]$
$E_1 E_2$	$[\![E_1]\!] \leq [\![E_2]\!] \rightarrow [\![E_1 E_2]\!]$
0	$[\![0]\!] \geq \mathsf{Int}$
$\mathsf{succ}\ E$	$[\![\mathsf{succ}\ E]\!] \geq \mathsf{Int} \wedge [\![E]\!] = \mathsf{Int}$

Figure 2: Type inference for partial types.

Type inference can analyze terms with respect to an arbitrary type environment. This is in contrast to safety analysis which is based on *closure analysis* [7, 1] (also called *control flow analysis* by Jones [2] and Shivers [8]). The *closures* of a term are simply the subterms corresponding to lambda abstractions. A closure analysis approximates for every subterm the set of possible closures to which it may evaluate [2, 7, 1, 8]. Safety analysis is simply a closure analysis that does appropriate safety checks. Our safety analysis requires that the initial type environment only binds variables to base types. This is because it requires knowledge of all closures that may occur during evaluation. Safety analysis thus has its applicability limited to mainly situations where a complete program is to be analyzed.

In safety analysis, type variables range over sets of closures and the base type Int. We denote by LAMBDA the finite set of all lambda tokens in the main term, henceforth called E_0. The constraints are generated from the syntax, see figure 3. As a conceptual aid, the constraints are grouped into *basic*, *safety*, and *connecting* constraints.

The connecting constraints reflect the relationship between formal and actual arguments and results. The condition $\lambda x \in [\![E_1]\!]$ states that the two inclusions are relevant only if the closure denoted by λx is a possible result of E_1.

We let SA denote the global constraint system, i.e., the collection of constraints for every subterm. If the safety constraints are excluded, then the remaining constraint system, denoted CA, yields a closure analysis. The SA constraint system for a simple term is shown in figure 4.

A solution assigns a set to each variable such that all constraints are satisfied. Solutions are ordered by variable-wise set inclusion. The CA system is always solvable: since we have no inclusion of the form $X \subseteq \{\ldots\}$, we can obtain a maximal solution by assigning LAMBDA $\cup \{\mathsf{Int}\}$ to every variable. Thus, closure information can always be obtained for a lambda term.

It is easy to see that if SA has a solution, then it has a unique minimal one. The proof follows from observing that solutions are closed under intersection, see [6]. SA need not be solvable, thus reflecting that not all lambda terms are safe.

Phrase:	Basic constraints:
$\lambda x.E$	$[\![\lambda x.E]\!] \supseteq \{\lambda x\}$
$\mathsf{0}$	$[\![\mathsf{0}]\!] \supseteq \{\mathsf{Int}\}$
$\mathsf{succ}\ E$	$[\![\mathsf{succ}\ E]\!] \supseteq \{\mathsf{Int}\}$

Phrase:	Safety constraints:
$E_1 E_2$	$[\![E_1]\!] \subseteq \text{LAMBDA}$
$\mathsf{succ}\ E$	$[\![E]\!] \subseteq \{\mathsf{Int}\}$

Phrase:	Connecting constraints:
$E_1 E_2$	For every $\lambda x.E$ in E_0, if $\lambda x \in [\![E_1]\!]$ then $[\![E_2]\!] \subseteq [\![x]\!] \wedge [\![E_1 E_2]\!] \supseteq [\![E]\!]$

Figure 3: Safety analysis.

The safety analysis accepts the term $\lambda f.(K(fI)(f\mathsf{0}))$ from before because the constraints reflect that $\mathsf{0}$ will not be applied to anything during evaluation.

There is a cubic time algorithm that, given E_0, computes the minimal solution of SA, or decides that none exists. The algorithm is based on a straightforward fixed-point computation.

In the paper [6] we showed that safety analysis is sound with respect to both a strict and a lazy semantics of the lambda calculus. Soundness means that if a term is accepted, then it is safe. We actually proved the soundness of a strictly better safety analysis, see [6]. The improved safety analysis will for example correctly accept $\lambda x.\mathsf{00}$ because it recognizes that $\mathsf{00}$ is "dead code".

3 The Result

We now show that safety analysis accepts strictly more safe terms than does type inference for partial types.

The proof involves several lemmas, see figure 5. The main technical problem to be solved is that SA and TI are constraint systems over two different domains, sets of closures versus types. This makes a direct comparison hard. We overcome this problem by applying solvability preserving maps into constraints over a common four-point domain.

We first show that the possibly *conditional* constraints of SA are equivalent to a set of *unconditional* constraints (USA). USA is obtained from SA by repeated

Constraints:

$$
\begin{array}{rcl}
 & & [\![\lambda y.y\mathsf{0}]\!] \supseteq \{\lambda y\} \\
 & & [\![\lambda x.x]\!] \supseteq \{\lambda x\} \\
 & & [\![\mathsf{0}]\!] \supseteq \{\mathsf{Int}\} \\
 & & [\![\lambda y.y\mathsf{0}]\!] \subseteq \{\lambda x, \lambda y\} \\
 & & [\![y]\!] \subseteq \{\lambda x, \lambda y\} \\
\lambda x \in [\![\lambda y.y\mathsf{0}]\!] & \Rightarrow & [\![\lambda x.x]\!] \subseteq [\![x]\!] \ \wedge \ [\![(\lambda y.y\mathsf{0})(\lambda x.x)]\!] \supseteq [\![x]\!] \\
\lambda y \in [\![\lambda y.y\mathsf{0}]\!] & \Rightarrow & [\![\lambda x.x]\!] \subseteq [\![y]\!] \ \wedge \ [\![(\lambda y.y\mathsf{0})(\lambda x.x)]\!] \supseteq [\![y\mathsf{0}]\!] \\
\lambda x \in [\![y]\!] & \Rightarrow & [\![\mathsf{0}]\!] \subseteq [\![x]\!] \ \wedge \ [\![y\mathsf{0}]\!] \supseteq [\![x]\!] \\
\lambda y \in [\![y]\!] & \Rightarrow & [\![\mathsf{0}]\!] \subseteq [\![y]\!] \ \wedge \ [\![y\mathsf{0}]\!] \supseteq [\![y\mathsf{0}]\!]
\end{array}
$$

Minimal solution:

$$
\begin{array}{l}
[\![(\lambda y.y\mathsf{0})(\lambda x.x)]\!] = [\![y\mathsf{0}]\!] = [\![\mathsf{0}]\!] = [\![x]\!] = \{\mathsf{Int}\} \\
[\![\lambda x.x]\!] = [\![y]\!] = \{\lambda x\} \\
[\![\lambda y.y\mathsf{0}]\!] = \{\lambda y\}
\end{array}
$$

Figure 4: SA constraints for $(\lambda y.y\mathsf{0})(\lambda x.x)$.

$$
\mathrm{TI} \overset{3}{\Longleftarrow} \overline{\mathrm{TI}} \overset{6}{\Longrightarrow} \mathrm{USA} \overset{1}{\Longleftarrow} \mathrm{SA}
$$

$$
4 \Downarrow \psi \qquad \phi \Updownarrow 2
$$

4-constraints

Figure 5: Solvability of constraints.

transformations. A set of constraints can be described by a pair (C, U) where C contains the conditional constraints and U the unconditional ones. We have two different transformations:

a) If U is solvable and c holds in the minimal solution, then $(C \cup \{c \Rightarrow K\}, U)$ becomes $(C, U \cup \{K\})$.

b) If case a) is not applicable, then (C, U) becomes $(\emptyset, U)$.

This process clearly terminates, since each transformation removes at least one conditional constraint. Note that case b) applies if either U is unsolvable or no condition in C is satisfied in the minimal solution of U.

Lemma 1: SA is solvable *iff* USA is solvable.
Proof: We show that each transformation preserves solvability.

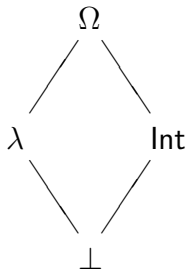
a) We know that U is solvable, and that c holds in the minimal solution, hence in all solutions. Assume that $(C \cup \{c \Rightarrow K\}, U)$ has solution L. Then L

is also a solution of U. Thus, c must hold in L, and so must K. But then $(C, U \cup \{K\})$ also has solution L. Conversely, assume that $(C, U \cup \{K\})$ is solvable. Then so is $(C \cup \{c \Rightarrow K\}, U)$, since K holds whether c does or not.

b) If (C, U) is solvable, then clearly so is $(\emptyset, U)$. Assume now that $(\emptyset, U)$ is solvable, and that no condition in C holds in the minimal solution of U. Then clearly (C, U) can inherit this solution.

It follows that solvability is preserved for any sequence of transformations. □

We now introduce a particularly simple kind of constraints, which we call *4-constraints*. Here variables range over the set $\{\bot, \lambda, \mathsf{Int}, \Omega\}$ which is partially ordered by $\sqsubseteq$ in the following way:



We define a function ϕ which maps USA constraints into 4-constraints. Individual constraints are mapped as follows:

USA	ϕ(USA)
$X \subseteq Y$	$X \sqsubseteq Y$
$X \subseteq$ LAMBDA	$X \sqsubseteq \lambda$
$X \supseteq \{\lambda x\}$	$X \sqsupseteq \lambda$
$X \supseteq \{\mathsf{Int}\}$	$X \sqsupseteq \mathsf{Int}$
$X \subseteq \{\mathsf{Int}\}$	$X \sqsubseteq \mathsf{Int}$

It turns out that ϕ preserves solvability.

Lemma 2: USA is solvable *iff* ϕ(USA) is solvable.
Proof: Assume that L is a solution of USA. We construct a solution of ϕ(USA) as follows:

$$\text{Assign to } X \begin{cases} \bot & \text{if } L(X) = \emptyset \\ \mathsf{Int} & \text{if } L(X) = \{\mathsf{Int}\} \\ \lambda & \text{if } L(X) \text{ is a non-empty subset of LAMBDA} \\ \Omega & \text{if } L(X) = Y \cup \{\mathsf{Int}\}, \text{ where } Y \text{ is a non-empty subset of LAMBDA} \end{cases}$$

Conversely, assume that L is a solution of ϕ(USA). We obtain a (non-minimal) solution of USA as follows:

$$\text{Assign to } X \begin{cases} \emptyset & \text{if } L(X) = \bot \\ \{\mathsf{Int}\} & \text{if } L(X) = \mathsf{Int} \\ \text{LAMBDA} & \text{if } L(X) = \lambda \\ \text{LAMBDA} \cup \{\mathsf{Int}\} & \text{if } L(X) = \Omega \end{cases}$$

This concludes the proof. □

Next, we define the closure $\overline{\text{TI}}$ as the smallest set of constraints that contains TI and is closed under antisymmetry, reflexivity, and transitivity of $\leq$, and the following property: if $\alpha \to \beta \leq \alpha' \to \beta'$, then $\alpha \geq \alpha'$ and $\beta \leq \beta'$. Hardly surprising, this closure preserves solvability.

Lemma 3: TI is solvable *iff* $\overline{\text{TI}}$ is solvable.
Proof: The implication from right to left is immediate. Assume that TI is solvable. The ordering $\leq$ is clearly antisymmetric, reflexive, and transitive. The additional property will also be true for any solution. Hence, $\overline{\text{TI}}$ inherits all solutions of TI. □

We define a function ψ which maps $\overline{\text{TI}}$ into 4-constraints. Individual constraints are mapped as follows:

$\overline{\text{TI}}$	$\psi(\overline{\text{TI}})$
$X \leq Y$	$X \sqsubseteq Y$
$X \leq \alpha \to \beta$	$X \sqsubseteq \lambda$
$X \geq \alpha \to \beta$	$X \sqsupseteq \lambda$
$X = \mathsf{Int}$	$X \sqsubseteq \mathsf{Int}$
$X \geq \mathsf{Int}$	$X \sqsupseteq \mathsf{Int}$

We show that ψ preserves solvability in one direction.

Lemma 4: If $\overline{\text{TI}}$ is solvable, then so is $\psi(\overline{\text{TI}})$.
Proof: Assume that L is a solution of $\overline{\text{TI}}$. We can construct a solution of $\psi(\overline{\text{TI}})$ by assigning Int to X if $L(X) = \mathsf{Int}$, assigning λ to X if $L(X) = \alpha \to \beta$, and assigning Ω to X if $L(X) = \Omega$. □

We now show the crucial connection between type inference and safety analysis.

Lemma 5: The USA constraints are contained in the $\overline{\text{TI}}$ constraints, in the sense that $\phi(\text{USA}) \subseteq \psi(\overline{\text{TI}})$.
Proof: We proceed by induction in the number of transformations performed on SA.

In the base case, we consider the SA configuration (C, U), where U contains all the basic and safety constraints. For any $\mathsf{0}$, SA yields the constraint $[\![\mathsf{0}]\!] \supseteq \{\mathsf{Int}\}$ which by ϕ is mapped to $[\![\mathsf{0}]\!] \sqsupseteq \{\mathsf{Int}\}$. TI yields the constraint $[\![\mathsf{0}]\!] \geq \{\mathsf{Int}\}$ which by ψ is mapped to $[\![\mathsf{0}]\!] \sqsupseteq \{\mathsf{Int}\}$ as well. A similar argument applies to the constraints yielded for $\mathsf{succ}\ E$, $\lambda x.E$, and $E_1 E_2$. Thus, we have established the induction base.

For the induction step we assume that $\phi(U) \subseteq \psi(\overline{\mathrm{TI}})$. If we use the b)-transformation and move from (C, U) to $(\emptyset, U)$, then the result is immediate. Assume therefore that we apply the a)-transformation. Then U is solvable, and some condition $\lambda x \in [\![E_1]\!]$ has been established for the application $E_1 E_2$ in the minimal solution. This opens up for two new connecting constraints: $[\![E_2]\!] \subseteq [\![x]\!]$ and $[\![E_1 E_2]\!] \supseteq [\![E]\!]$. We must show that similar inequalities hold in $\overline{\mathrm{TI}}$. The only way to enable the condition in the minimal solution of U is to have a chain of U-constraints:

$$\{\lambda x\} \subseteq [\![\lambda x.E]\!] \subseteq X_1 \subseteq X_2 \subseteq \cdots \subseteq X_n \subseteq [\![E_1]\!]$$

Since both ϕ and ψ act like the identity on constraints that are inequalities between variables, we know by the induction hypothesis that in $\overline{\mathrm{TI}}$ we have

$$[\![\lambda x.E]\!] \leq X_1 \leq X_2 \leq \cdots \leq X_n \leq [\![E_1]\!]$$

From the TI constraints $[\![\lambda x.E]\!] \geq [\![x]\!] \to [\![E]\!]$ and $[\![E_1]\!] \leq [\![E_2]\!] \to [\![E_1 E_2]\!]$ and the closure properties of $\overline{\mathrm{TI}}$ it follows that $[\![E_2]\!] \leq [\![x]\!]$ and $[\![E_1 E_2]\!] \geq [\![E]\!]$, which was our proof obligation. Thus, we have established the induction step.

As USA is obtained by a finite number of transformations, the result follows. □

This allows us to complete the final link in the chain.

Lemma 6: If $\overline{\mathrm{TI}}$ is solvable, then so is USA.
Proof: Assume that $\overline{\mathrm{TI}}$ is solvable. From lemma 4 it follows that so is $\psi(\overline{\mathrm{TI}})$. Since from lemma 5 $\phi(\mathrm{USA})$ is a subset, it must also be solvable. From lemma 2 it follows that USA is solvable. □

We conclude that SA is at least as powerful as TI.

Theorem: If TI is solvable, then so is SA.
Proof: We need only to bring the lemmas together, as indicated in figure 5. □

Some safe terms are accepted by SA but rejected by TI. As an example, consider the term $(\lambda x.xx)(\lambda x.xx)$. It is accepted by SA because it contains no constants, so no safety constraints will be involved. The term is not accepted by TI, however, as shown by O'Keefe and Wand [5].

The proof of our theorem sheds some light on why and how SA accepts more safe terms than TI. Consider a solution of TI that is transformed into a solution of SA

according to the strategy implied in figure 5. All closure sets will be the maximal set LAMBDA. Thus, the more fine-grained distinction between individual closures is lost.

Our result is still valid if we allow recursive types. Here the TI constraints are exactly the same, but the types are changed from finite to regular trees. This allows solutions to constraints such as $X \leq \mathsf{Int} \to X$. Only lemma 4 is influenced, but the proof carries through with virtually no modifications. Type inference with recursive types will accept all terms without constants, as does SA. It remains to be seen if the containment in SA is still strict. Note though that the containment in the improved safety analysis [6] is trivially strict.

The development in this paper can straightforwardly be extended to an arbitrary signature of constants. The idea is to treat base types like we have treated Int, and to treat structured types, such as List, like we have treated lambda abstractions.

References

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

[2] Neil D. Jones. Flow analysis of lambda expressions. In *Proc. Eighth Colloquium on Automata, Languages, and Programming*, pages 114–128. Springer-Verlag (*LNCS* 115), 1981.

[3] 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.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[5] Patrick M. O'Keefe and Mitchell Wand. Type inference for partial types is decidable. In *Proc. ESOP'92, European Symposium on Programming*, pages 408–417. Springer-Verlag (*LNCS* 582), 1992.

[6] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[7] Peter Sestoft. Replacing function parameters by global variables. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.

[8] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991. CMU–CS–91–145.

[9] Satish Thatte. Type inference with partial types. In *Proc. International Colloquium on Automata, Languages, and Programming 1988*, pages 615–629. Springer-Verlag (*LNCS* 317), 1988.

[10] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.