Safety Anaysis versus Type Inference

Jens Palsberg
palsberg@daimi.aau.dk

Michael I. Schwartzbach
mis@daimi.aau.dk

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

June 1993

Abstract

Safety analysis is an algorithm for determining if a term in the untyped lambda calculus with constants is *safe*, i.e., if it does not cause an error during evaluation. This ambition is also shared by algorithms for type inference. Safety analysis and type inference are based on rather different perspectives, however. Safety analysis is based on closure analysis, whereas type inference attempts to assign a type to all subterms.

In this paper we prove that safety analysis is *sound*, relative to both a strict and a lazy operational semantics, and *superior* to type inference, in the sense that it accepts strictly more safe lambda terms.

The latter result may indicate the relative potentials of static program analyses based on respectively closure analysis and type inference.

1 Introduction

In this paper we compare two techniques for analyzing the *safety* of terms in the untyped lambda calculus with constants. The safety we are concerned with is the absence of "constant misuse", such as an attempt to compute $\sqrt{\mathsf{true}}$.

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

Figure 1: The lambda calculus.

One way of achieving this is to perform a standard *type inference* [7]; if a term is typable, then safety is guaranteed. We propose another technique, which we shall simply call *safety analysis*; it is based on closure analysis [10, 2] and does not perform a type reconstruction. We show that this new technique is sound and superior to type inference, in the sense that it can accept strictly more safe terms. These results are illustrated in figure 2.

Safety analysis may be an alternative to type inference for implementations of untyped functional languages. Apart from the safety property, type inference also computes the actual type information, which may be useful for improving the efficiency of implementations. Safety analysis similarly computes closure information, which is also useful for improving efficiency.

Type inference can be implemented in linear time. Safety analysis can be implemented in worst-case cubic time.

Type inference can analyze terms with respect to an arbitrary type environment. Safety analysis requires the type environment of the "main" term to bind variables to only base types. For simplicity, we will prove the soundness of safety analysis for only *closed* terms.

The language that we are concerned with is shown in figure 1. It is the untyped lambda calculus with two constants: $\mathsf{0}$ and succ. The techniques and results presented in this paper generalize without problems to arbitrary constants. For technical reasons it is convenient for succ to always require an argument; if desired, a combinator version can be programmed as $\lambda x.\mathsf{succ}\ x$.

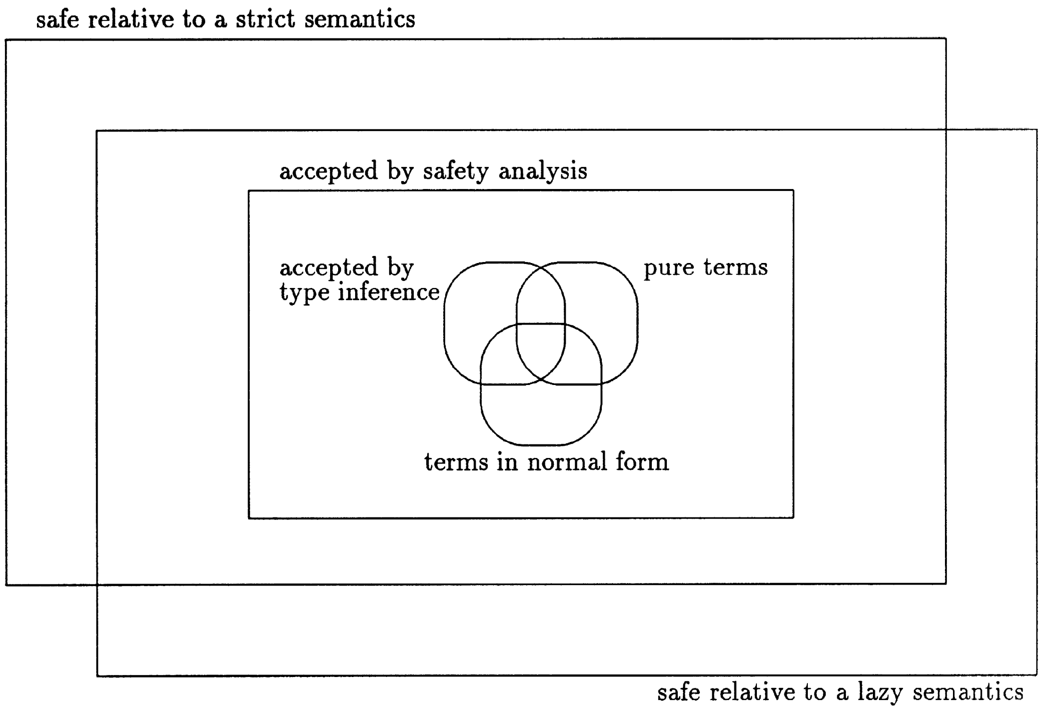

safe relative to a strict semantics
accepted by safety analysis
accepted by type inference
pure terms
terms in normal form
safe relative to a lazy semantics

Figure 2: Sets of safe lambda terms.

2 Type Inference

The most common notion of practical type inference (TI), with which we shall compare our safety analysis, is simple type inference. Note that ML-polymorphism conceptually is just a syntactic expansion of named definitions, followed by a simple type inference. This expansion, which may exponentially increase the size of the program, could similarly be performed before a safety analysis.

$$\tau ::= \alpha \mid \mathsf{Int} \mid \tau_1 \rightarrow \tau_2$$

Figure 3: Type Schemes.

A straightforward presentation of simple type inference, due to Wand [12], is as follows. First, the lambda term is α-converted so that every λ-bound variable is distinct. Second, a type variable $[\![E]\!]$ is assigned to every subterm E; these variables range over type schemes, shown in figure 3. Third, a finite collection of constraints over these variables is generated from the syntax. Finally, these constraints are solved.

The constraints are generated inductively in the syntax, as shown in figure 4.

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

Figure 4: Constraints on type variables.

A finite collection of constraints can be solved by unification yielding a most general solution. If no solution exists, then the program is not typable. Soundness and syntactic completeness of this algorithm is due to Milner [7].

3 Safety Analysis

Safety analysis (SA) is based on a novel algorithm for closure analysis, which slightly improves the algorithm used in the SIMILIX partial evaluator [2]. Safety analysis based on SIMILIX's algorithm corresponds to *primitive* SA, which in section 5 is shown to be weaker than SA.

The *closures* of a term are simply the subterms corresponding to lambda abstraction. A closure analysis approximates for every subterm the set of possible closures to which it may evaluate.

The SA algorithm shares many similarities with that for TI. First, the lambda term is α-converted so that every λ-bound variable is distinct. This means that every closure $\lambda x.E$ can be denoted by the unique token λx. Second, a type variable $[\![E]\!]$ is assigned to every subterm E; these variables range

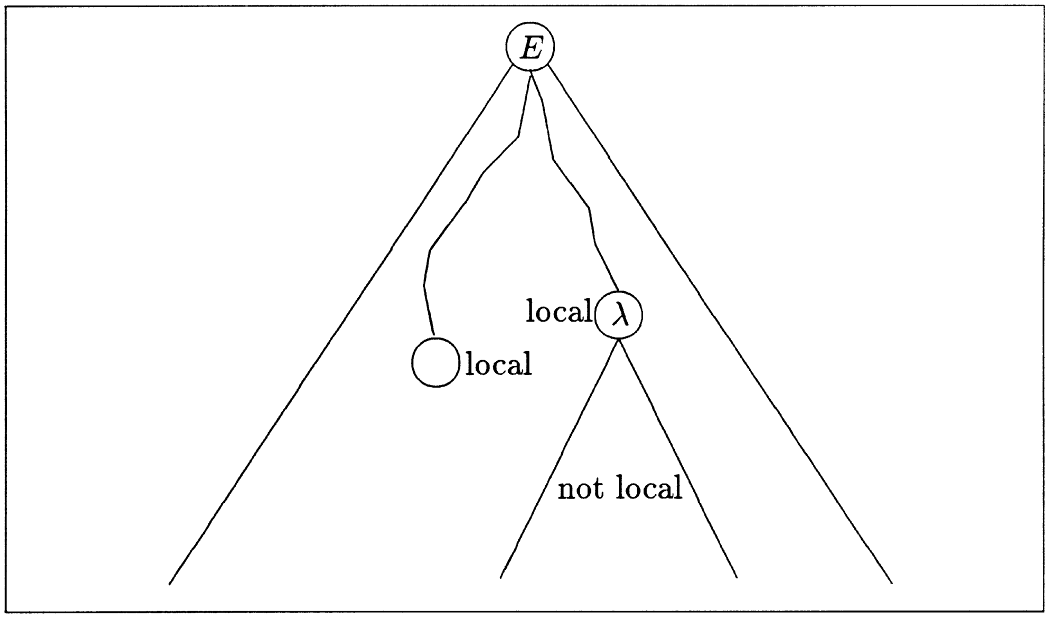


Figure 5: Local nodes in a parse tree.

over sets of closures and the simple "type" Int. Third, a finite collection of constraints over these variables is generated from the syntax, Finally, these constraints are solved.

The two algorithms differ in the domain over which constraints are specified, and in the manner in which these are generated from the syntax.

In the remaining we consider a fixed lambda term E_0. We denote by LAMBDA the finite set of all lambda tokens in E_0. Consider the parse tree for any subterm E of E_0. We shall call a parse tree node *local*, if it can be reached from the root without passing through a lambda abstraction. This is illustrated in figure 5.

The constraints relating to SA are best explained by means of a *trace graph*. It has a node for each closure, denoted by the corresponding lambda token, and one for E_0, denoted MAIN. The edges will reflect possible applications. An example trace graph is sketched in figure 6.

With every trace graph node we associate a set of *local constraints*; some of those will be called local *safety* constraints. The trace graph node corre-

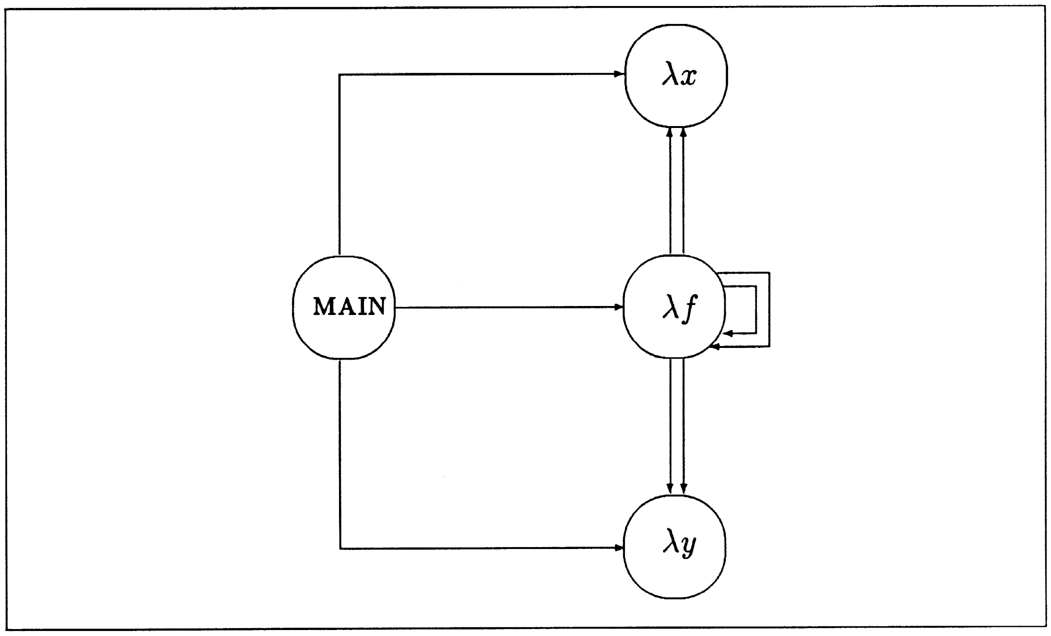


Figure 6: Trace graph for $(\lambda f.(f(\lambda x.\mathsf{succ}\ x))(f\ \mathsf{0}))(\lambda y.y)$.

sponds to a closure. The local constraints are generated from its local parse tree nodes as indicated in figure 7.

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

Phrase:	Constraint
E_1E_2	$[\![E_1]\!] \subseteq \textsc{lambda}$
$\mathsf{succ}\ E$	$[\![E]\!] \subseteq \{\mathsf{Int}\}$

Figure 7: Local constraints.

The outgoing trace graph edges arise from local applications. For every E_1E_2 we have an edge to any other trace graph node. With each trace graph edge we associate a *condition* and two *connecting constraints*. The condition is simply $\lambda x \in [\![E_1]\!]$; it states that this edge is only relevant if the closure of the indicated trace graph node is a possible result of E_1. The connecting constraints reflect the relationship between formal and actual arguments and results. They are $[\![E_2]\!] \subseteq [\![x]\!]$ and $[\![E_1 E_2]\!] \supseteq [\![E]\!]$. The situation is illustrated in figure 8.

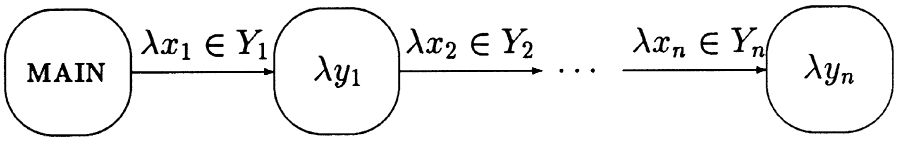
$$\begin{tikzcd}
E_1 E_2 \arrow[rr, "\lambda x \in [\![E_1]\!]", "{\begin{array}{c}[\![E_2]\!] \subseteq [\![x]\!] \\ [\![E_1 E_2]\!] \supseteq [\![E]\!]\end{array}}"'] & & \lambda x.E
\end{tikzcd}$$

Figure 8: Trace graph edges.

From the trace graph we derive a finite set of *global constraints*. Each of these is a *conditional inclusion* of the form

$$\lambda x_1 \in Y_1, \ldots, \lambda x_n \in Y_n \Rightarrow X \subseteq Y$$

Each path in the trace graph from the MAIN node gives rise to global constraints as follows. Suppose the path is



The corresponding global constraints are

$$\lambda x_1 \in Y_1, \ldots, \lambda x_n \in Y_n \Rightarrow \text{LOCAL} \cup \text{CONNECT}$$

where LOCAL are the local constraints of the final node (λy_n) and CONNECT are the connecting constraints of the final edge ($\lambda y_{n-1} \rightarrow \lambda y_n$).

We let SA denote the global constraint system. If the local safety constraints are excluded, then we denote the remaining constraint system by CA (for Closure Analysis). We have some simple observations.

Proposition 3.1: CA is always solvable.
Proof: Since we have no inclusion of the form $X \subseteq \{\ldots\}$, we obtain a maximal solution by assigning LAMBDA $\cup$ $\{\mathsf{Int}\}$ to every variable. $\Box$

This means that closure information always can be obtained for every lambda term. In contrast, SA need not be solvable, since not all lambda terms are safe.

Proposition 3.2: If SA has a solution, then it has a unique minimal one.
Proof: The result follows from solutions being closed under intersection. To see this, consider any conditional inclusion of the form $\lambda x_1 \in Y_1, \lambda x_2 \in Y_2, \ldots, \lambda x_n \in Y_n \Rightarrow X \subseteq Y$, and let $\{L_i\}$ be all solutions. We shall show that $\cap_i L_i$ is a solution. If a condition $\lambda x_j \in \cap_i L_i(Y_j)$ is true, then so are all of $\lambda x_j \in L_i(Y_j)$. Hence, if all the conditions of $X \subseteq Y$ are true in $\cap_i L_i$ then they are true in each L_i. Furthermore, since they are solutions, $X \subseteq Y$ is also true in each L_i. Since in general $A_k \subseteq B_k$ implies $\cap_k A_k \subseteq \cap_k B_k$ it follows that $\cap_i L_i$ is a solution. Hence, if there are any solutions, then $\cap_i L_i$ is the unique smallest one. $\Box$

There is a cubic time algorithm that given E_0 computes the minimal solution of SA, or decides that none exists. We omit the details; the algorithm is based on an incremental fixed-point computation.

4 Soundness

We now show that SA is *sound*, i.e., if a term is accepted, then it is safe. We show the soundness with respect to both a strict (call by value, applicative order reduction) and a lazy (call by name, normal order reduction) semantics of the lambda calculus.

To see that neither of the strict and lazy cases imply the other, consider the two lambda terms in figure 9. Applicative order reduction of the first yields an infinite loop, whereas normal reduction of it yields an error. In contrast, applicative order reduction of the second yields an error, whereas normal reduction of it yields an infinite loop. Thus, the soundness with respect to one of the reduction strategies does not imply the soundness with respect to the other.

$$1)\ (\lambda x.err)(loop)$$
$$2)\ (\lambda x.loop)(err)$$

where $err = \mathsf{0}\mathsf{0}$ and $loop = \Delta\Delta$, with $\Delta = (\lambda x.xx)$

Figure 9: Two lambda terms.

The two semantics of the untyped lambda calculus will be given as natural semantics [6, 4], involving sequents and inference rules. The two proofs of soundness have the same structure, as follows.

First, the soundness of environment lookup is proved, by induction in the structure of derivation trees. Second, the soundness of closure analysis of a term in a so-called E_0-well-formed environment is proved, by structural induction. Third, the E_0-well-formedness of *all* environments occurring in a sequent in a derivation tree is proved, by induction in the depth of sequents. From these lemmas, the soundness of closure and safety analysis easily follows.

4.1 Strict Semantics

We present in figure 10 a strict operational semantics which explicitly deals with constant misuse. An evaluation that misuses constants yields the result *wrong*.

The semantics uses *environments* and *values*, which are simultaneously defined in figure 11.

The entire soundness argument is for a fixed lambda term E_0, in which each λ-bound variable is distinct. Throughout, E_S denotes an arbitrary subterm of E_0. We need some terminology. Let L_0 be any solution of CA. For all subterms E of E_0, we let ambiguously $[\![E]\!]$ denote $L_0([\![E]\!])$. We will say that a sequent $\rho \vdash E : v$ or $\rho \vdash_{\text{val}} x : v$ is *active*, if it occurs in a derivation tree for $\vdash_{\text{main}} E_0 : w$, for some w, and if E or x occur in a trace graph node N where there exists a path from the main node to N whose conditions all hold in L_0. The predicate $\text{ABS}(\_, \_)$ is defined on a constraint variable and value.

$$1.\ \frac{\emptyset \vdash E : v}{\vdash_{\text{main}} E : v} \qquad 2.\ \frac{\rho \vdash_{\text{val}} x : v}{\rho \vdash x : v} \qquad 3.\ \frac{}{\rho \vdash \lambda x.E :< \lambda x.E, \rho >}$$

$$4.\ \frac{\rho \vdash E_1 :< \lambda x.E, \rho_1 > \quad \rho \vdash E_2 : w \quad x \mapsto w \cdot \rho_1 \vdash E : v}{\rho \vdash E_1 E_2 : v}\ w \neq \mathit{wrong}$$

$$5.\ \frac{\rho \vdash E_1 : v}{\rho \vdash E_1 E_2 : \mathit{wrong}}\ v \text{ is not a closure} \qquad 6.\ \frac{\rho \vdash E_2 : \mathit{wrong}}{\rho \vdash E_1 E_2 : \mathit{wrong}}$$

$$7.\ \frac{}{\rho \vdash \mathsf{0} : \mathit{0}} \qquad 8.\ \frac{\rho \vdash E : \mathit{succ}^n\ \mathit{0}}{\rho \vdash \mathsf{succ}\ E : \mathit{succ}^{n+1}\ \mathit{0}}$$

$$9.\ \frac{\rho \vdash E : v}{\rho \vdash \mathsf{succ}\ E : \mathit{wrong}}\ v \text{ is not a number}$$

$$10.\ \frac{}{x \mapsto v \cdot \rho \vdash_{\text{val}} x : v} \qquad 11.\ \frac{\rho \vdash_{\text{val}} x : v}{y \mapsto w \cdot \rho \vdash_{\text{val}} x : y}\ x \neq y$$

Figure 10: Strict semantics.

Intuitively, $\text{ABS}([\![E]\!], v)$ means that $[\![E]\!]$ is an abstract description of v. The precise requirement is that

- if $v = \mathit{succ}^n\ \mathit{0}$ then $\{\mathsf{Int}\} \subseteq [\![E]\!]$, and
- if $v =< \lambda x.E', \rho >$ then $\{\lambda x\} \subseteq [\![E]\!]$.

Notice that $\text{ABS}([\![E]\!], \mathit{wrong})$ always holds.

The E_0-well-formedness (E_0-wf) of environments and values is defined in figure 12. It intuitively states that the environment or value may occur during a safe evaluation of E_0.

Lemma 4.1: If ρ is an E_0-wf environment and $\rho \vdash_{\text{val}} x : v$ is active, then v is E_0-wf and $\text{ABS}([\![x]\!], v)$.
Proof: We proceed by induction in the structure of a derivation of $\rho \vdash_{\text{val}} x : v$. In the base case, consider rule 10. From $x \mapsto v \cdot \rho$ being E_0-wf, it follows that v is E_0-wf. Since $x \mapsto v \cdot \rho \vdash_{\text{val}} x : v$ is active, it follows that $\text{ABS}([\![x]\!], v)$.

1.	a.	$\emptyset$ is an environment
	b.	$x \mapsto w \cdot \rho$ is an environment, *iff*
		• w is a value
		• ρ is an environment
2.	a.	$\mathit{succ}^n\ \mathit{0}$ is a value, called a *number*, for all n
	b.	$< \lambda x.E, \rho >$ is a value, called a *closure*, *iff*
		• ρ is an environment
	c.	*wrong* is a value

Figure 11: Environments and values.

In the induction step, consider rule 11. From $y \mapsto w \cdot \rho$ being E_0-wf, it follows that ρ is E_0-wf. From $y \mapsto w \cdot \rho \vdash_{\text{val}} x : v$ being active, it follows that $\rho \vdash_{\text{val}} x : v$ is active. We can then apply the induction hypothesis, from which the conclusion is immediate. □

Lemma 4.2: If ρ is an E_0-wf environment and $\rho \vdash E_S : v$ is active, then v is either E_0-wf or wrong, and ABS($[\![E_S]\!], v$).
Proof: We proceed by induction in the structure of E_S. In the base, we consider x, $\mathsf{0}$, and $\mathsf{succ}\ E$. First, consider rule 2, the one for x. Since $\rho \vdash x : v$ is active, so is $\rho \vdash_{\text{val}} x : v$, and the conclusion follows from lemma 4.1.

Second, consider rule 7, the one for $\mathsf{0}$. Since $\rho \vdash \mathsf{0} : \mathit{0}$ is active, the constraint $[\![\mathsf{0}]\!] \supseteq \{\mathsf{Int}\}$ is satisfied, so ABC($[\![\mathsf{0}]\!] : \mathit{0}$). It is immediate that $\mathit{0}$ is E_0-wf.

Third, consider rules 8 and 9, those for $\mathsf{succ}\ E$. If rule 9 has been applied, then the conclusion is immediate. If rule 8 has been applied, then we use that $\rho \vdash \mathsf{succ}\ E : v$ is active to conclude that the constraint $[\![\mathsf{succ}\ E]\!] \supseteq \{\mathsf{Int}\}$ is satisfied, so ABS($[\![\mathsf{succ}\ E]\!], \mathit{succ}^{n+1}\mathit{0}$). It is immediate that $\mathit{succ}^{n+1}\mathit{0}$ is E_0-wf.

In the induction step we consider $\lambda x.E$ and $E_1 E_2$.

First, consider rule 3, the one for $\lambda x.E$. Since $\rho \vdash \lambda x.E :< \lambda x.E, \rho >$ is active, the constraint $[\![\lambda x.E]\!] \supseteq \{\lambda x\}$ is satisfied, so ABS($[\![\lambda x.E]\!], < \lambda x.E, \rho >$). To prove that $< \lambda x.E, \rho >$ is E_0-wf, we apply the induction hypothesis to E, from which the conclusion is immediate.

Second, consider rules 4, 5, and 6, those for $E_1 E_2$. If rule 5 or 6 has been applied, then the conclusion is immediate. If rule 4 has been applied, then

1. a. $\emptyset$ is E_0-wf
   b. $x \mapsto w \cdot \rho$ is E_0-wf, *iff*
      - x is λ-bound in E_0
      - w is E_0-wf
      - ρ is E_0-wf
      - if $x \mapsto w \cdot \rho \vdash_{\text{val}} x : w$ is active, then $\textsc{abs}([\![x]\!], w)$
2. a. *succ^n 0* is E_0-wf, for all n
   b. $< \lambda x.E, \rho >$ is E_0-wf, *iff*
      - $\lambda x.E$ is a subterm of E_0
      - ρ is E_0-wf
      - if w is an E_0-wf value and $x \mapsto w \cdot \rho$ is E_0-wf and $x \mapsto w \cdot \rho \vdash E : v$ is an active, then
        - v is either E_0-wf or *wrong*, and
        - $\textsc{abs}([\![E]\!], v)$

Figure 12: E_0-well-formedness.

we use that $\rho \vdash E_1 E_2 : v$ is active to conclude that also $\rho \vdash E_1 :< \lambda x.E, \rho_1 >$ and $\rho \vdash E_2 : w$ are active, and that $w \neq$ *wrong*. By applying the induction hypothesis to E_1 and E_2, we get that $< \lambda x.E, \rho_1 >$ and w are E_0-wf, and that $\textsc{abs}([\![E_1]\!], < \lambda x.E, \rho_1 >)$ and $\textsc{abs}([\![E_2]\!], w)$. From $\textsc{abs}([\![E_1]\!], < \lambda x.E, \rho_1 >)$ we get that $\lambda x \in [\![E_1]\!]$. This means that $x \mapsto w \cdot \rho_1 \vdash_{\text{val}} E : v$ is active, and that the connecting constraints $[\![E_2]\!] \subseteq [\![x]\!]$ and $[\![E_1 E_2]\!] \supseteq [\![E]\!]$ hold. It follows from $< \lambda x.E, \rho_1 >$ being E_0-wf that ρ_1 is E_0-wf. To prove that $x \mapsto w \cdot \rho_1$ is E_0-wf we need to prove that if $x \mapsto w \cdot \rho_1 \vdash_{\text{val}} x : w$ is active, then $\textsc{abs}([\![x]\!], w)$. But $\textsc{abs}([\![x]\!], w)$ is unconditionally true, because $\textsc{abs}([\![E_2]\!], w)$ and $[\![E_2]\!] \subseteq [\![x]\!]$. From $< \lambda x.E, \rho_1 >$ being E_0-wf, we then get that v is either E_0-wf or *wrong*, and $\textsc{abs}([\![E]\!], v)$. It thus remains to be shown that $\textsc{abs}([\![E_1 E_2]\!], v)$. This follows from $([\![E_1 E_2]\!] \supseteq [\![E]\!]$. $\square$

<u>Lemma 4.3</u>: Any sequent, except the root, occurring in a derivation tree for $\vdash_{\text{main}} E_0 : w$, for some w, is active and has an environment component that is E_0-wf.

<u>Proof</u>: Let there be given a w and a derivation tree for $\vdash_{\text{main}} E_0 : w$. It suffices to prove that for all $n \geq 1$, the sequents in distance n from the root are active and have environment components that are E_0-wf. We proceed by induction in n.

In the base, we observe that only one sequent has distance 1 from the root, see rule 1. The expression in this sequent occurs in the *root* node of the trace graph, so the sequent is active. Its environment component is $\emptyset$ which is E_0-wf.

In the induction step, we consider the rules 2,4,5,6,8,9, and 11. In each case we assume that the conclusion sequent is active and has an environment component that is E_0-wf. We must then prove that the same holds for the hypothesis sequents.

Consider first the six cases excluding rule 4. They all have one hypothesis sequent, and in all cases its expression occurs in the *same* trace graph node as the expression of the conclusion sequent. Hence, the hypothesis sequent is also active. In cases 2,5,6,8, and 9, the environment components of the conclusion and hypothesis sequents are identical, so, in particular, that of the hypothesis sequent is E_0-wf. In case 11, it is also immediate that the environment component of the hypothesis sequent is E_0-wf.

Now, consider rule 4. It is immediate the first two hypotheses are active and have environment components that are E_0-wf. Then notice that in the trace graph there is an edge from the node containing E_1E_2 to the $\lambda x.E$-node, labeled with the condition $\lambda x \in [\![E_1]\!]$. By using lemma 4.2, we get that $< \lambda x.E, \rho_1 >$ and w are E_0-wf, that $\textsc{abs}([\![E_2]\!], w)$, and that $\lambda x \in [\![E_1]\!]$. The last condition implies that also the third hypothesis is active, and that the connecting constraint $[\![E_2]\!] \subseteq [\![x]\!]$ holds. It remains to be shown that $x \mapsto w \cdot \rho_1$ is E_0-wf. From $< \lambda x.E, \rho_1 >$ being E_0-wf, we get that ρ_1 is E_0-wf. We then only need to show that if $x \mapsto w \cdot \rho_1 \vdash_{\text{val}} x : w$ is active, then $\textsc{abs}([\![x]\!], w)$. But $\textsc{abs}([\![x]\!], w)$ is unconditionally true, since $\textsc{abs}([\![E_2]\!], w)$ and $[\![E_2]\!] \subseteq [\![x]\!]$. $\Box$

We first show that CA is sound.

Lemma 4.4: If $\rho \vdash E : v$ occurs in a derivation tree for $\vdash_{\text{main}} E_0 : w$, for some w, then $\textsc{abs}([\![E]\!], v)$.
Proof: From lemma 4.3 it follows that $\rho \vdash E : v$ is active and that ρ is E_0-wf. The conclusion then follows from lemma 4.2. $\Box$

We then show that SA is sound.

<u>Theorem 4.5</u>: If SA is solvable and $\vdash_{\text{main}} E_0 : v$, then $v \neq \mathit{wrong}$.
<u>Proof</u>: First note that any solution of SA is also a solution of CA. Now, suppose that $\vdash_{\text{main}} E_0 : \mathit{wrong}$. In the semantics, it is easy to see that *wrong* must have been *introduced* by either rule 5 or rule 9.

Suppose first that it we by rule 5. Theorem 4.4 applied to $\rho \vdash E_1 : \mathit{succ}^n 0$ gives that $\{\mathsf{Int}\} \subseteq [\![E_1]\!]$. Lemma 4.3 gives that $\rho \vdash E_1 E_2 : \mathit{wrong}$ is active, so the local safety constraint $[\![E_1]\!] \subseteq \text{LAMBDA}$ holds. This yields a contradiction.

Suppose next that it was by rule 9. Theorem 4.4 applied to $\rho \vdash E :< \lambda x.E', \rho' >$ gives that $\{\lambda x\} \subseteq [\![E]\!]$. Lemma 4.3 gives that $\rho \vdash \mathsf{succ}\ E : \mathit{wrong}$ is active, so the local safety constraint $[\![E]\!] \subseteq \{\mathsf{Int}\}$ holdup This yields a contradiction. $\Box$

4.2 Lazy Semantics

We present in figure 13 a lazy operational semantics which explicitly deals with constant misuse, as did the strict semantics. There is no rule number 6, to keep the numbering consistent with that in the strict semantics.

The semantics uses *environments* and *values*, which are simultaneously defined in figure 14.

The new sort of value is that of *thunks*, defined in case 2.d in figure 14. Thunks are used to capture that the evaluation of arguments can be delayed and later resumed. In the semantics, thunks are introduced in rule 4, and eliminated using rules 12 and 13. The two last rules may be understood as defining an operation 'res' which evaluates a lambda term to a non-thunk value. Notice that rules 1, 4, 5, 8, end 9 use the 'res' operation.

The soundness argument uses the same terminology as in the strict case. We only need slight modifications of the notion of activeness, the predicate $\text{ABS}(\_,\_)$, and the notion of E_0-well-formedness, as follows.

A sequent $\rho \vdash_{\text{res}} E : v$ may be active in the same way as $\rho \vdash E : v$ and $\rho \vdash_{\text{val}} x : v$.

The predicate $\text{ABS}([\![E]\!], v)$ holds *iff*

- if $v = \mathit{succ}^n 0$ then $\{\mathsf{Int}\} \subseteq [\![E]\!]$,

$$1.\ \frac{\emptyset \vdash_{\text{res}} E : v}{\vdash_{\text{main}} E : v} \qquad 2.\ \frac{\rho \vdash_{\text{val}} x : v}{\rho \vdash x : v} \qquad 3.\ \frac{}{\rho \vdash \lambda x.E :< \lambda x.E, \rho >}$$

$$4.\ \frac{\rho \vdash_{\text{res}} E_1 :< \lambda x.E, \rho_1 > \quad x \mapsto [E_2, \rho] \cdot \rho_1 \vdash E : v}{\rho \vdash E_1 E_2 : v}$$

$$5.\ \frac{\rho \vdash_{\text{res}} E_1 : v}{\rho \vdash E_1 E_2 : \mathit{wrong}}\ v \text{ is not a closure} \qquad \text{(rule 6. ommitted)}$$

$$7.\ \frac{}{\rho \vdash \mathsf{0} : \mathit{0}} \qquad 8.\ \frac{\rho \vdash_{\text{res}} E : \mathit{succ}^n \mathit{0}}{\rho \vdash \mathsf{succ} E : \mathit{succ}^{n+1} \mathit{0}}$$

$$9.\ \frac{\rho \vdash E_{\text{res}} : v}{\rho \vdash \mathsf{succ} E : \mathit{wrong}}\ v \text{ is not a number}$$

$$10.\ \frac{}{x \mapsto v \cdot \rho \vdash_{\text{val}} x : v} \qquad 11.\ \frac{\rho \vdash_{\text{val}} x : v}{y \mapsto w \cdot \rho \vdash_{\text{val}} x : y}\ x \neq y$$

$$12.\ \frac{\rho \vdash E : v}{\rho \vdash_{\text{res}} E : v}\ v \text{ is not a thunk} \qquad 13.\ \frac{\rho \vdash E : [E', \rho'] \ \rho \vdash_{\text{res}} E' : v}{\rho \vdash_{\text{res}} E : v}$$

Figure 13: Lazy semantics.

- if $v =< \lambda x.E', \rho >$ then $\{\lambda x\} \subseteq [\![E]\!]$, and
- if $v = [E', \rho]$ then $[\![E']\!] \subseteq [\![E]\!]$.

The third case is added to handle thunks.

Furthermore, the E_0-well-formedness (E_0-wf) of environments and values needs to be modified, see figure 15. Compared to the notion of E_0-well-formedness used in the strict case, we have added case 2.c to handle thunks.

Note that lemma 4.1 still holds, with an unchanged proof. We need a replacement for lemma 4.2, however, as follows.

Lemma 4.6: Suppose ρ is an E_0-wf environment. 1) If $\rho \vdash E_S : v$ is

1. a. $\emptyset$ is an environment
   b. $x \mapsto w \cdot \rho$ is an environment, *iff*
   - w is an walue
   - ρ is an environment
2. a. *succ*^{n}*0* is a value, called a *number*, for all n
   b. $< \lambda x.E, \rho >$ is a value, called a *closure, iff*
   - ρ is an environment

   c. *wrong* is a value
   d. $[E, \rho]$ is a value, called a *trunk, iff*
   - ρ is an enviroment

Figure 14: Environments and values.

active, then v is either E_0-wf or *wrong*, and ABS$(\llbracket E_S \rrbracket, v)$. Furthermore, 2) if $\rho \vdash_{\text{res}} E_S : v$ is active, then v is either E_0-wf or *wrong*, and ABS$(\llbracket E_S \rrbracket, v)$.
<u>Proof</u>: We proceed by induction in the structure of E_S. In the base, we consider x, 0, and succ E. Case 1) is proved in the same way as the base case of lemma 4.2. To prove case 2), we consider the rules 12 and 13. If rule 12 has been applied, then the conclusion follows from case 1). If rule 13 has been applied, then it follows from case 1) that $[E', \rho']$ is E_0-wf and that ABS$(\llbracket E \rrbracket, [E', \rho'])$. Hence, $\rho' \vdash_{\text{res}} E' : v$ is active, so v is either E_0-wf or wrong, and ABS$(\llbracket E' \rrbracket, v)$. The conclusion now follows, since $\llbracket E' \rrbracket \subseteq \llbracket E \rrbracket$.

In the induction step we consider $\lambda x.E$ and $E_1 E_2$.

First, consider $\lambda x.E$. Case 1) is proved in the same way as in lemma 4.2. Case 2) is proved in the same way as case 2) in the base case above.

Second, consider $E_1 E_2$. In case 1), either rule 4 or 5 has been applied. If rule 5 has been applied then the conclusion is immediate. If rule 4 has been applied, then we use that $\rho \vdash E_1 E_2 : v$ is active to conclude that also $\rho \vdash E_1 :< \lambda x.E, \rho_1 >$ is active. By applying the induction hypothesis to E_1, we get that $< \lambda x.E, \rho_1 >$ is E_0-wf and that ABS$(\llbracket E_1 \rrbracket, < \lambda x.E, \rho_1 >)$. From the latter we get that $\lambda x \in \llbracket E_1 \rrbracket$. This means that $x \mapsto [E_2, \rho] \cdot \rho_1 \vdash_{\text{val}} E : v$ is active, and that the connecting constraints $\llbracket E_2 \rrbracket \subseteq \llbracket x \rrbracket$ and $\llbracket E_1 E_2 \rrbracket \supseteq \llbracket E \rrbracket$ hold. It follows from $< \lambda x.E, \rho_1 >$ being E_0-wf that ρ_1 is E_0-wf. To prove that $x \mapsto [E_2, \rho] \cdot \rho_1$ is E_0-wf we need to prove that $[E_2, \rho]$ is E_0-wf and that if $x \mapsto [E_2, \rho] \cdot \rho_1 \vdash_{\text{val}} x : [E_2, \rho]$ is active, then ABS$(\llbracket x \rrbracket, [E_2, \rho])$. The first follows by applying the induction hypothesis, case 2), to E_2. The second

1. a. $\emptyset$ is E_0-wf
   b. $x \mapsto w \cdot \rho$ is E_0-wf, *iff*
      - x is λ-bound in E_0
      - w is E_0-wf
      - ρ is E_0-wf
      - if $x \mapsto w \cdot \rho \vdash_{\text{val}} x : w$ is active, then
        - $\textsc{abs}([\![x]\!], w)$.
2. a. $succ^n 0$ is E_0-wf, for all n
   b. $< \lambda x.E, \rho >$ is E_0-wf, *iff*
      - $\lambda x.E$ is a subterm of E_0
      - ρ is E_0-wf
      - if w is an E_0-wf value
        and $x \mapsto w \cdot \rho$ is E_0-wf
        and $x \mapsto w \cdot \rho \vdash E : v$ is active, then
        - v is either E_0-wf or *wrong*, and
        - $\textsc{abs}([\![E]\!], v)$
   c. $[E, \rho]$ is E_0-wf, *iff*
      - E is a subterm of E_0 and occurs in a trace graph node N where there exists a path from the main node to N whose conditions all hold in L_0
      - ρ is E_0-wf
      - if $\rho \vdash_{\text{res}} E : v$ is active, then
        - v is either E_0-wf or *wrong*, and
        - $\textsc{abs}([\![E]\!], v)$

Figure 15: E_0-well-formedness.

follows because $[\![E_2]\!] \subseteq [\![x]\!]$ is unconditionally true. From $< \lambda x.E, \rho_1 >$ being E_0-wf, we then get that v is either E_0-wf or *wrong*, and $\textsc{abs}([\![E]\!], v)$. It thus remains to be showy that $\textsc{abs}([\![E_1 E_2]\!], v)$. This follows from $[\![E_1 E_2]\!] \supseteq [\![E]\!]$.

Case 2) is proved in the same way as case 2) in the base case above. □

Note that lemma 4.3 still holds, with only a few simple changes to proof which we leave to the reader.

The soundness of CA is in the lazy case expressed as follows.

<u>Theorem 4.7</u>: If $\rho \vdash E : v$ occurs in a derivation tree for $\vdash_{\text{main}} E_0 : w$, for some w, then $\textsc{abs}([\![E]\!], v)$. Furthermore, if $\rho \vdash_{\text{res}} E : v$ occurs in a derivation tree for $\vdash_{\text{main}} E_0 : w$, for some w, then $\textsc{abs}([\![E]\!], v)$.
<u>Proof</u>: From lemma 4.3 it follows that $\rho \vdash E : v$ is active and that ρ is E_0-wf. The conclusion then follows from lemma 4.6. A similar argument proves the second case. $\Box$

The soundness of SA, theorem 4.5, also holds in the lazy case. The proof is the same, *mutatis mutandis*.

5 Superiority

We now show that SA accepts strictly more safe terms than does TI.

The proof will involve several steps. The main technical problem is that SA and TI are constraint systems over two different domains, sets of closures versus type schemes. This makes a direct comparison hard. Furthermore, the TI constraints are much simpler than those in SA. We overcome these problems by successively weakening SA and strengthening TI, until we finally can apply solvability preserving maps into constraints over a common two-point domain. All the lemmas, which are summarized in figure 16, establish that solvability is preserved as required.

The entire argument is for a fixed lambda term E_0, in which each λ-bound variable is distinct. The common variables of all constraint systems are the $[\![E]\!]$'s, where E is a subterm of E_0.

We first define a restricted version of SA, called *primitive* safety analysis (PSA), which is also superior to TI. All the local constraints are made unconditional, and for connecting constraints the conditions are weakened by discarding all but the final conjunct. Furthermore, constraints are strengthened according to the following table:

SA	PSA
$\{\lambda x\} \subseteq Y$	$\{\lambda x\} \subseteq Y$
$X \subseteq Y$	$X = Y$
$\{\mathsf{Int}\} \subseteq X$	$\{\mathsf{int}\} = X$
$X \subseteq \textsc{lambda}$	$X \subseteq \textsc{lambda}$
$X \subseteq \{\mathsf{Int}\}$	$\{\mathsf{Int}\} = X$

Lemma 5.1: If PSA is solvable, then so is SA.
Proof: This is immediate, since the constraints of PSA logically imply those of SA. □

We next show that the *conditional* constraints of PSA are equivalent to a set of *unconditional* constraints (USA).

USA is obtained from PSA by repeated 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 minimals solution, then $(C \cup \{c \Rightarrow K\}, U)$ becomes $(C, U \cup \{K\})$.

b) Otherwise, (C, U) becomes $(\emptyset, U)$.

This process clearly terminates, since each transformation removes at least one conditional constraint.

Lemma 5.2: PSA 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. Assume that $(C \cup \{c \Rightarrow K\}, U)$ is solvable. The condition c must hold and, hence, so must K. But then $(C, U \cup \{K\})$ is solvable. 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 *2-constraints*. Here variables range over the binary set $\{\lambda, \mathsf{Int}\}$ and constraints are all of the form $X = Y$, $X = \lambda$, or $X = \mathsf{Int}$.

We define a function ϕ which maps USA constraints into 2-constraints. Individual constraints are mapped as follows:

SA	ϕ(USA)
$X = Y$	$X = Y$
$X \subseteq$ LAMBDA	$X = \lambda$
$\{\lambda x\} \subseteq X$	$X = \lambda$
$\{\mathsf{Int}\} = X$	$X = \mathsf{int}$

It turns out that ϕ preserves solvability.

<u>Lemma 5.3:</u> USA is solvable *iff* ϕ(USA) is solvable.
<u>Proof</u>: Assume that L is a solution of USA. We construct a solution of ϕ(USA) by assigning Int to X if $L(X) = \{\mathsf{Int}\}$ and assigning λ to X otherwise. Conversely, assume that L is a solution of ϕ(USA). We obtain a (non-minimal) solution of USA by assigning $\{\mathsf{Int}\}$ to X if $L(X) = \{\mathsf{Int}\}$ and assigning LAMBDA to X otherwise. □

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

<u>Lemma 5.4:</u> TI is solvable *iff* $\overline{\text{TI}}$ is solvable.
<u>Proof:</u> The implication from right to left is immediate. Assume that TI is solvable. Equality is by definition symmetric, reflexives 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 2-constraints. Individual constraints are mapped as follows:

$\overline{\mathrm{TI}}$	$\psi(\overline{\mathrm{TI}})$
$X = Y$	$X = Y$
$X = \alpha \to \beta$	$X = \lambda$
$X = \mathsf{Int}$	$X = \mathsf{int}$

We show that ψ preserves solvability in one direction.

Lemma 5.5: If $\overline{\mathrm{TI}}$ is solvable, then so is $\psi(\overline{\mathrm{TI}})$.
Proof: Assume that L is a solution of $\overline{\mathrm{TI}}$. We can construct a solution of $\psi(\overline{\mathrm{TI}})$ by assigning Int to X if $L(X) = \mathsf{Int}$, and assigning λ to X otherwise. Thus, the function ψ acts as a quotient map on constraint systems. $\Box$

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

Lemma 5.6: The USA constraints are contained in the $\overline{\mathrm{TI}}$ constraints, in the sense that $\phi(\mathrm{USA}) \subseteq \psi(\overline{\mathrm{TI}})$.
Proof: We perform an induction in the number of transformations performed on PSA. A general configuration looks like (C, U); the hypothesis is that $\phi(U) \subseteq \psi(\overline{\mathrm{TI}})$.

The induction base is the PSA configuration (C, U). Here U contain all the local constraints, which are all easy to relate to $\overline{\mathrm{TI}}$: Local constraints of the form $X = \{\mathsf{Int}\}$, always have analogous constraints in $\overline{\mathrm{TI}}$, Similarly, a local constraint like $[\![E_1]\!] \subseteq \textsc{lambda}$ arises from an application E_1E_2. In $\overline{\mathrm{TI}}$ this gives rise to a constraint $[\![E_1]\!] = [\![E_2]\!] \to [\![E_1E_2]\!]$, which by ψ maps to $[\![E_1]\!] = \lambda$. But this is just the image of $[\![E_1]\!] \subseteq \textsc{lambda}$ under ϕ. Also, the local constraint $\{\lambda x\} \subseteq [\![\lambda x.E]\!]$ maps by ϕ to $[\![\lambda x.E]\!] = \lambda$. But this is the image under ψ of the $\overline{\mathrm{TI}}$ constraint $[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!]$. 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_1E_2 in the minimal solution. This opens up for two new connecting constraints $[\![x]\!] = [\![E_2]\!]$ and $[\![E_1E_2]\!] = [\![E]\!]$. We must show that the same equalities 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:

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

Since both ϕ and ψ act like the identity on equality constraints, we know by the induction hypothesis that in $\overline{\text{TI}}$ we have

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

From the TI constraints $[\![\lambda x.E]\!] = [\![x]\!] \to [\![E]\!]$ and $[\![E_1]\!] = [\![E_2]\!] \to [\![E_1 E_2]\!]$ and the closure properties of $\overline{\text{TI}}$ it follows that $[\![x]\!] = [\![E_2]\!]$ and $[\![E_1 E_2]\!] = [\![E]\!]$ 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 5.7: If $\overline{\text{TI}}$ is solvable, then so is USA.
Proof: Assume that $\overline{\text{TI}}$ is solvable. From lemma 5.5 it follows that so is $\psi(\overline{\text{TI}})$. Since from lemma 5.6 $\phi(\text{USA})$ is a subset, it must also be solvable. From lemma 5.3 it follows that USA is solvable. □

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

Theorem 5.8: If TI is solvable, then so is SA.
Proof: We need only to bring the lemmas together, as indicated in figure 16. □

$$\begin{tikzcd}
\text{TI} \arrow[r, Leftrightarrow, "5.4"] & \overline{\text{TI}} \arrow[r, Rightarrow, "5.7"] \arrow[d, Rightarrow, "5.5"', "\psi"] & \text{USA} \arrow[r, Leftrightarrow, "5.2"] \arrow[dl, Leftrightarrow, "\phi"', "5.3"] & \text{PSA} \arrow[r, Rightarrow, "5.1"] & \text{SA} \\
& \text{2-constraints} & & &
\end{tikzcd}$$

Figure 16: Solvability of constraints.

We now show that SA is in fact superior to TI.

Theorem 5.9: There exists a safe term that is accepted by SA but rejected by TI.
Proof: Many such terms exist, since SA accepts any lambda term in normal form that has no safety errors on the outermost level; TI rejects many such terms, such as $\lambda f.(f\mathsf{0})(f(\lambda x.x))$. Similarly, SA accepts any lambda term

that does not contain a constant; again, TI rejects many such terms, such as $\lambda x.xx$. $\square$

We contend, naturally, that the extra power of SA will be significant for numerous useful functional programs.

The above proof also 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 16. All closure sets will be the maximal set LAMBDA. Thus, the more fine-grained distinction between individual closures is lost.

The results are still valid if we allow recursive types, as in the $\lambda\mu$-calculus [1]. Here the TI constraints are exactly the same, but the type schemes are changed from finite to regular trees. This allows solutions to constraints such as $X = X \rightarrow \mathsf{Int}$. Only lemma 5.5 is influenced, but the proof carries through with virtually no modifications. Type inference with recursive types will also accept all terms without constants, but the containment in SA is still strict.

6 Conclusion

We have presented a new algorithm, safety analysis, for deciding safety of lambda terms. It has been proved sound and strictly more powerful than standard type inference. Safety analysis is sound for both lazy and strict semantics, but not for *arbitrary* reduction strategies. To see this, consider the term $\lambda x.\lambda y.\mathsf{0}\mathsf{0}$, which is accepted by the algorithm but will cause an error if $\mathsf{0}\mathsf{0}$ is reduced. We conjecture, however, that primitive safety analysis is sound for β-reduction.

The algorithm for safety analysis can be implemented in cubic time by a technique of local fixed-point computation [9]. This shows that safety analysis realistically can be incorporated into a compiler for an untyped functional language.

Type inference has been used as the basis of *binding time analysis* [5]; so has closure analysis [2]. We hope to use the techniques presented here to formally compare the quality of these analyses.

There are other type systems for the lambda calculus, for which type inference is possible. In particular, we think of *partial types* [11, 8] and *simple intersection types* [3]. Neither encompasses constants in its present form, but this should be easy to remedy. We hope to extend figure 2 by proving more containment results involving these systems.

References

[1] Barendregt and Hemerik. Types in lambda calculi and programming languages. In *Proc. ESOP'90, European Symposium on Programming.* Springer-Verlag (*LNCS* 432), 1990.

[2] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *Proc. ESOP'90, European Symposium on Programming.* Springer-Verlag (*LNCS* 432), 1990.

[3] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *Proc. CAAP'92.* Springer-Verlag (*LNCS* 581), 1992.

[4] Joëlle Despeyroux. Proof of translation in natural semantics. In *LICS'86, First Symposium on Logic in Computer Science*, June 1986.

[5] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Progmmming*, 1(1):21–69, 1991.

[6] Gilles Kahn. Natural semantics. In *Proc. STACS'87* Springer-Verlag (*LNCS* 247), 1987.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

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

[9] Jens Palsberg and Michael I. Schwartzbach. Polyvariant analysis of the untyped lambda calculus. Computer Science Department, Aarhus University. PB-386. Submitted for publication, 1992.

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

[11] Satish Thatte. Type inference with partial types. In *Proc. International Colloquium on Automata, Languages, and Progmmming 1988.* Springer-Verlag (*LNCS* 317), 1988.

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