To appear in *Information and Computation.*

Safety Analysis versus Type Inference

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

Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Århus 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. 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 *global* in that it can only analyze a complete program. In contrast, type inference is *local* in that it can analyze pieces of a program in isolation.

In this paper we prove that safety analysis is *sound*, relative to both a strict and a lazy operational semantics. We also prove that safety analysis accepts strictly more safe lambda terms than does type inference for simple types. The latter result demonstrates that global program analyses can be more precise than local ones.

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, such as an attempt to compute $\sqrt{\mathsf{true}}$. In this paper we consider just the two constants $\mathsf{0}$ and succ. They can be misused by applying a number to an argument, or by applying succ to an abstraction. Safety is undecidable so any sound analysis algorithm must reject some safe programs.

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

Figure 1: The lambda calculus.

One way of achieving a safety guarantee is to perform *type inference* [16]; 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 (also called control flow analysis) [11, 19, 3, 20] and does not perform a type reconstruction.

We prove that this new technique is sound and that it accepts strictly more safe lambda terms than does type inference for simple types. These results are illustrated in figure 2.

We also present examples of lambda terms that demonstrate the strictness of the established inclusions.

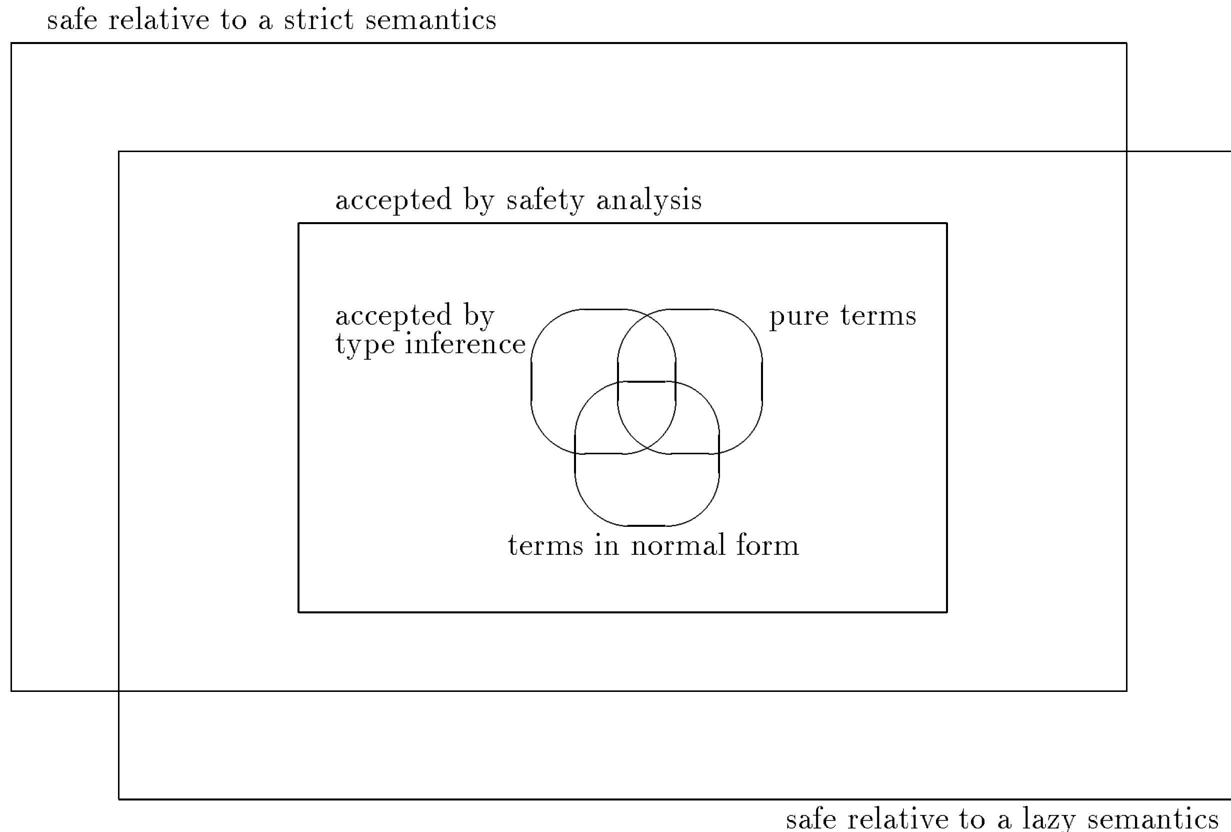


Figure 2: Sets of safe lambda terms.

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 for our lambda calculus can be implemented in linear time (there is no polymorphic let). Safety analysis can be implemented in worst-case cubic time.

In practice, a program is an abstraction, e.g., $\lambda x.E$. The program $\lambda x.E$ takes its input through the variable x and it yields the value obtained by evaluating the body E. Any analysis of such a program should take all possible inputs into account. For technical reasons we will assume that lambda terms to be analyzed take their input through the free variables. This means that if a program $\lambda x.E$ is to be analyzed, then we will analyze only E where indeed the free occurrences of the variable x corresponds to input. For example consider the program

$$\lambda x.\mathsf{if}\ x < \mathsf{0}\ \mathsf{then}\ -x\ \mathsf{else}\ x$$

(It is written in a larger language then the one that we later give a formal treatment). If this program is to be analyzed, then we will analyze only

$$\mathsf{if}\ x < 0\ \mathsf{then}\ -x\ \mathsf{else}\ x$$

The assumption about taking input through the free variables is convenient when defining constraints on the inputs. This is because the notion of free variable is independent of the form of the lambda term to be analyzed. Henceforth we assume that all lambda terms already have been put into the appropriate form.

Safety analysis and type inference are based on rather different perspectives. Safety analysis is *global* in that it can only analyze a complete program that takes only first-order values as inputs. In contrast, type inference is *local* in that it can analyze pieces of a program in isolation. Our comparison of the two techniques thus demonstrates that a global program analysis can be more precise than a local one.

That safety analysis can only analyze programs that take first-order values as inputs is of course a limitation. In practice, however, one can represent higher-order input as a first-order data structure. This is for example done in Bondorf's partial evaluator SIMILIX [3, 4] and in Gomard and Jones' partial evaluator LAMBDA-MIX [9]. The former partial evaluator is applicable to a higher-order subset of SCHEME and the latter to a lambda calculus with constants. SIMILIX for example contains a parser that transforms SCHEME programs into a first-order representation.

One advantage of a local analysis is that it is *modular* (or *incremental*) in that new routines can be added to a program without creating a need to re-analyze the program. If a complete program is to be analyzed, however, then the greater precision of safety analysis may provide a safety guarantee in situations where type inference fails to do so.

Safety analysis may in practice be most useful in a language such a SCHEME that use run-time tagging and tag-checking, rather than type inference. If a safety guarantee is provided, then the run-time tag-checks can be eliminated.

We will present safety analysis is two steps. First we will present a basic form of the analysis which like type inference analyzes all subterms of a given term. Then we extend this analysis with a device for detecting and avoiding the analysis of *dead code*. Dead code is a subterm of a given term that will not be evaluated during neither strict nor lazy evaluation. For example, in $\lambda x.\mathsf{00}$ the subterm $\mathsf{00}$ is dead code. The extended safety analysis will accept all abstractions, such as $\lambda x.\mathsf{00}$, since the body is dead code.

Avoiding the safety analysis of dead code may be useful in practice. For example, if a program uses only a small part of a large library of routines, then only the routines actually used need to be analyzed. This saves time and avoids possible type errors in routines that are never called.

The basic safety analysis may be interesting in itself, if, as we conjecture, this analysis is sound with respect to arbitrary β-reduction.

Although we treat only the two constants $\mathsf{0}$ and succ, the safety analysis technique and our results can be generalized to handle arbitrary constants. For technical reasons it is convenient for succ to always require an argument; if desired, a combinator version can be programmed as $\lambda x.\mathsf{succ}\ x$. Polymorphic constants can be treated in a manner similar to how we treat lambda abstractions.

In the following section we recall the definition of type inference, and in section 3 we introduce the definition of safety analysis. In section 4 we give the soundness proofs for safety analysis, and in section 5 we give the proof of comparison of safety analysis and type inference.

2 Type Inference

The most common notion of practical type inference, with which we shall compare our safety analysis, is type inference for simple types. Polymorphic let could be treated by doing syntactic expansion before the type inference. Kannellakis, Mairson, and Mitchell [13, 15] proved that although this expansion may exponentially increase the size of the program, no type inference algorithm for polymorphic let has better worst-case complexity. Such expansion 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 [22], 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. We let TI denote the collection of constraints for all subterms.

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 well-known [10, 16, 6].

The TI constraint system for the term $(\lambda y.y\mathsf{0})(\lambda x.x)$ is shown in figure 5. This term is used as the running example in the following section.

Constraints:

$$[\![\lambda y.y\mathsf{0}]\!] = [\![\lambda x.x]\!] \rightarrow [\![(\lambda y.y\mathsf{0})(\lambda x.x)]\!]$$
$$[\![\lambda y.y\mathsf{0}]\!] = [\![y]\!] \rightarrow [\![y\mathsf{0}]\!]$$
$$[\![\lambda x.x]\!] = [\![x]\!] \rightarrow [\![x]\!]$$
$$[\![\mathsf{0}]\!] = \mathsf{Int}$$
$$[\![y]\!] = [\![\mathsf{0}]\!] \rightarrow [\![y\mathsf{0}]\!]$$

Solution:

$$[\![(\lambda y.y\mathsf{0})(\lambda x.x)]\!] = [\![y\mathsf{0}]\!] = [\![\mathsf{0}]\!] = [\![x]\!] = \mathsf{Int}$$
$$[\![\lambda x.x]\!] = [\![y]\!] = \mathsf{Int} \rightarrow \mathsf{Int}$$
$$[\![\lambda y.y\mathsf{0}]\!] = (\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}$$

Figure 5: TI constraints for $(\lambda y.y\mathsf{0})(\lambda x.x)$.

3 Safety Analysis

Safety analysis is based on *closure analysis* [19, 3] (also called *control flow analysis* by Jones [11] and Shivers [20]). 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 [11, 19, 3, 20].

The basic form of safety analysis, which we present first, is simply a closure analysis that does appropriate safety checks. This safety analysis is essentially the one used in the SIMILIX partial evaluator [3]. Having presented this basic analysis, we proceed by extending it with detection of dead code. This involves the notion of a trace graph.

The safety analysis algorithms share many similarities with that for type inference. 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 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.

Safety analysis and type inference differ in the domain over which constraints are specified, and in the manner in which these are generated from the syntax. In a previous paper [18] we successfully applied safety analysis to a substantial subset of the object-oriented language SMALLTALK [8], demonstrating how to deal with inheritance, assignments, conditionals, late binding, etc.

3.1 The Basic Safety Analysis

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. In the constraint system that we will generate from E_0, type variables range over subsets of the union of LAMBDA and $\{\mathsf{Int}\}$.

The constraints are generated from the syntax, see figure 6. As a conceptual aid, the constraints are grouped into *basic*, *safety*, and *connecting* constraints.

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:
$E_1 E_2$	$[\![E_1]\!] \subseteq$ 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 6: Safety analysis.

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 the term $(\lambda y.y\mathsf{0})(\lambda x.x)$ is shown in figure 7.

We assume that a lambda term takes its input through the free variables. A term that is to be safety analyzed can only take first-order values as inputs. This means that for every free variable x, if indeed input will be taken through x, we can add the *initial* constraint $[\![x]\!] = \mathsf{Int}$ to the TI constraint system, and we can add the *initial* constraint $[\![x]\!] \supseteq \{\mathsf{Int}\}$ to the SA constraint system.

A solution of a constraint system 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. In contrast, SA need not be solvable, since not all lambda terms are safe. Instead, as proved in the following subsection, if SA has a solution, then it has a minimal one.

Ayers [1] has presented a cubic time algorithm that computes essentially the minimal solution of CA. It is straightforward to incorporate into his algorithm the checks yielded by our safety constraints. Ayers' algorithm also applies to the extension of safety analysis which we consider next. Below we sketch a cubic time algorithm, similar to Ayers', that computes the minimal solution of SA or decides that none exists. We also indicate how it can be modified to deal with the following extension.

Constraints:

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

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

3.2 Detection of Dead Code

Dead code is a subterm of a given term that will not be evaluated during neither strict nor lazy evaluation. Intuitively, dead code may be found near the leaves of syntax trees, since both strict and lazy evaluation are "top-down" evaluation strategies. We will now extend the basic safety analysis such that it detects at least some dead code and avoids analyzing such code.

Our approach to the detection of dead code is essentially to add conditions to some of the constraints yielded by the basic safety analysis. This makes it more likely that the constraint system has a solution, thus more terms will be deemed safe.

We will explain the addition of conditions by means of a *trace graph*. To define that we need the auxiliary notion of local nodes in a parse tree for an arbitrary lambda term E. We shall call a parse tree node *local* in E, if it can be reached from the root of E's parse tree without passing through a lambda abstraction. This is illustrated in figure 8.

We can then define trace graphs. Intuitively, the nodes correspond to functions and the edges correspond to possible applications.

Definition 3.2: The trace graph associated with a lambda term E_0 is a directed graph with:

- **Nodes**. For each abstraction in E_0 there is a node, denoted by the corresponding lambda token, and there is a node for E_0 itself, denoted MAIN. Local subterms of E_0 are said to *occur* in MAIN, and similarly are local subterms of an abstraction said to occur in the trace graph node for that abstraction. A trace graph node is labeled by those basic and safety constraints (see the previous subsection) that are generated from its local expressions.

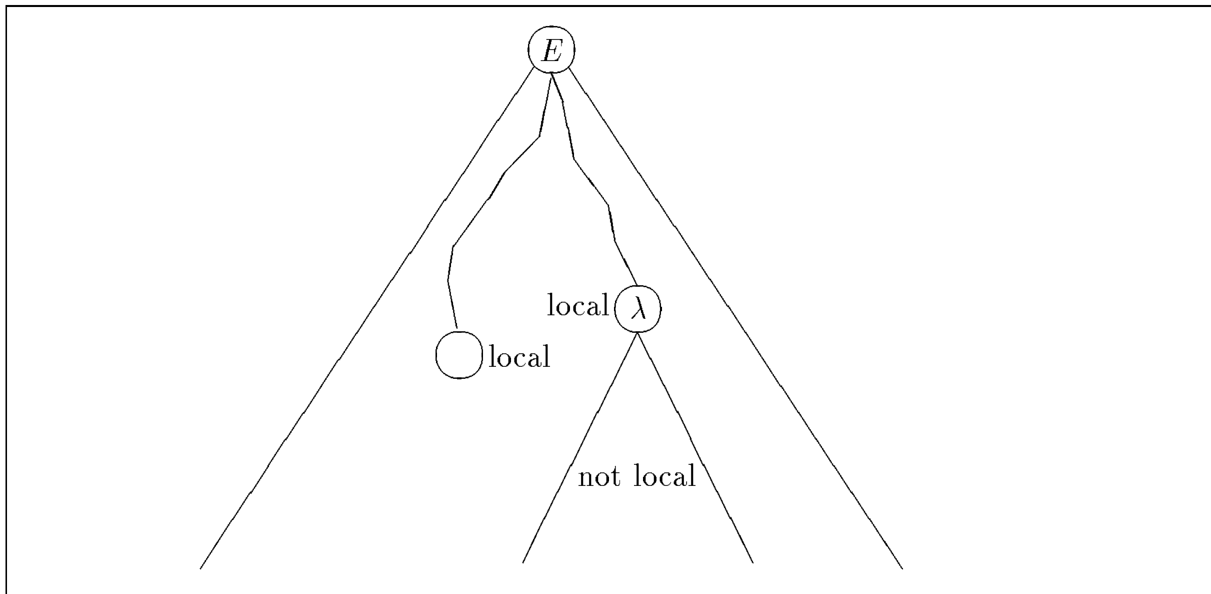


Figure 8: Local nodes in a parse tree.

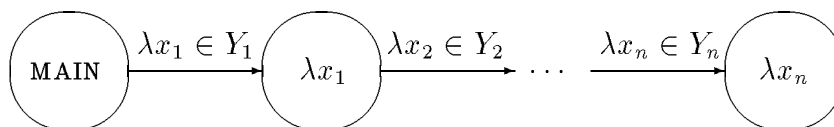
- **Edges**. For each trace graph node N, if E_1E_2 occurs in N, then there is a directed edge from N to every trace graph node for abstractions (but not to MAIN). Notice that from a node there is an outgoing edge for each local application. Edges are labeled by conditions and connecting constraints, as follows. If an edge is yielded by the application E_1E_2 and the edge leads to the node for the abstraction $\lambda x.E$, then the edge is labeled with the condition $\lambda x \in [\![E_1]\!]$ and the connecting constraints $[\![E_2]\!] \subseteq [\![x]\!]$ and $[\![E_1E_2]\!] \supseteq [\![E]\!]$.

Notice that the number of edges is at most quadratic in the number of nodes. □

The trace graph for the term $(\lambda y.y0)(\lambda x.x)$ is shown in figure 9 (omitting connecting constraints to avoid clutter).

From a trace graph we derive a finite set of *global constraints*. Intuitively, this set is the union of the constraints for every potential "top-down" evaluation sequence.

A potential "top-down" evaluation sequence is represented in the trace graph by a path from the MAIN node. Such a path is illustrated in the following figure which omits the constraints to avoid clutter:



The constraints that we derive from this path are:

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

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

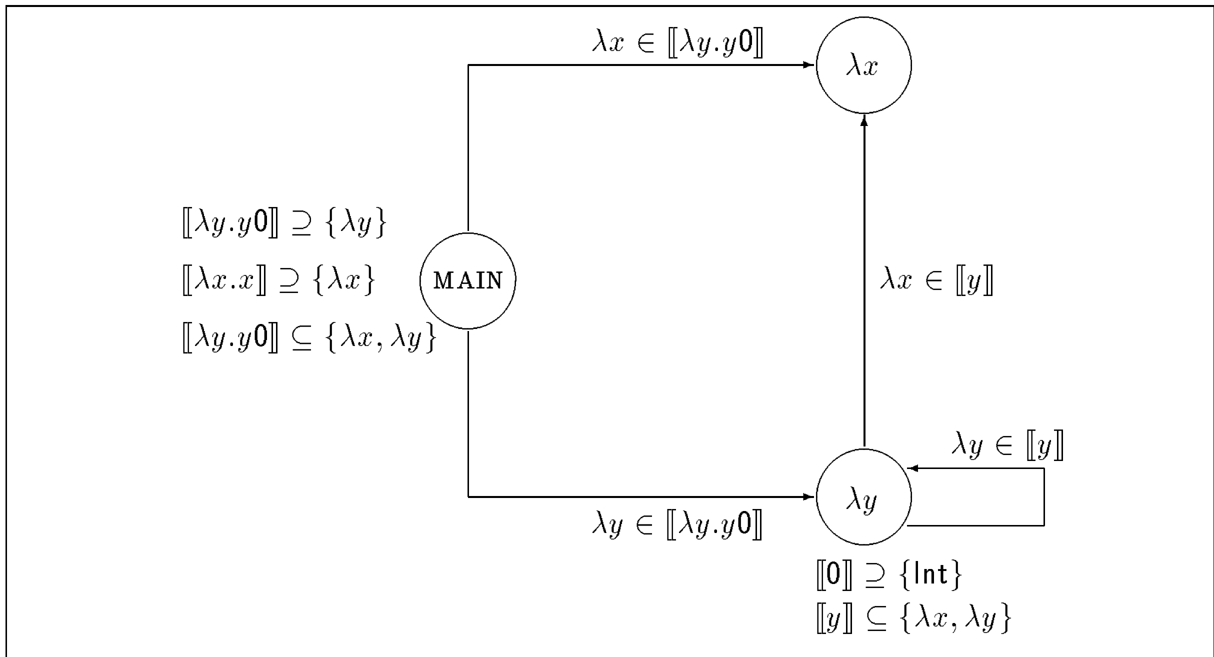


Figure 9: Trace graph for $(\lambda y.y0)(\lambda x.x)$.

Notice that there may be infinitely many paths from the MAIN node. Many of them yield redundant constraints, however, namely those where a condition appears more than once. Thus, we derive constraints for only those paths that use edges at most once. This yields a finite constraint system which is solvable if and only if the constraint system generated from *all* paths is solvable.

Notice also that the constraints can be normalized in linear time to a set of constraints of the form:

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

The normalization proceeds by rewriting constraints of the form $C \Rightarrow Y \supseteq X$ to $C \Rightarrow X \subseteq Y$ and rewriting $C \Rightarrow D \wedge D'$ to the two constraints $C \Rightarrow D$ and $C \Rightarrow D'$, where C is a (possibly empty) conjunction of conditions and D, D' are inclusions.

We let SA_R (R for Reachability) denote the global, finite constraint system. We let CA_R denote the subset of SA_R where the safety constraints are excluded.

Proposition 3.1: If SA_R 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. □

A subset of the SA constraint system from the previous subsection can be obtained from SA_R by deleting the conditions on basic and safety constraints and by deleting all but

Constraints:

$$\text{local (MAIN)} \quad \left[\begin{array}{l} [\![\lambda y.y0]\!] \supseteq \{\lambda y\} \\ [\![\lambda x.x]\!] \supseteq \{\lambda x\} \\ [\![\lambda y.y0]\!] \subseteq \{\lambda x, \lambda y\} \end{array} \right.$$

$$\text{connecting (MAIN to } \lambda x) \quad \left[\; \lambda x \in [\![\lambda y.y0]\!] \Rightarrow \left\{ \begin{array}{l} [\![\lambda x.x]\!] \subseteq [\![x]\!] \\ [\![(\lambda y.y0)(\lambda x.x)]\!] \supseteq [\![x]\!] \end{array} \right. \right.$$

$$\text{connecting (MAIN to } \lambda y) \quad \left[\; \lambda y \in [\![\lambda y.y0]\!] \Rightarrow \left\{ \begin{array}{l} [\![\lambda x.x]\!] \subseteq [\![y]\!] \\ [\![(\lambda y.y0)(\lambda x.x)]\!] \supseteq [\![y0]\!] \end{array} \right. \right.$$

$$\text{local } (\lambda y) \quad \left[\; \lambda y \in [\![\lambda y.y0]\!] \Rightarrow \left\{ \begin{array}{l} [\![0]\!] \supseteq \{\mathsf{Int}\} \\ [\![y]\!] \subseteq \{\lambda x, \lambda y\} \end{array} \right. \right.$$

$$\text{connecting } (\lambda y \text{ to } \lambda y) \quad \left[\; \lambda y \in [\![\lambda y.y0]\!] \wedge \lambda y \in [\![y]\!] \Rightarrow \left\{ \begin{array}{l} [\![0]\!] \subseteq [\![y]\!] \\ [\![y0]\!] \supseteq [\![y0]\!] \end{array} \right. \right.$$

$$\text{connecting } (\lambda y \text{ to } \lambda x) \quad \left[\; \lambda y \in [\![\lambda y.y0]\!] \wedge \lambda x \in [\![y]\!] \Rightarrow \left\{ \begin{array}{l} [\![0]\!] \subseteq [\![x]\!] \\ [\![y0]\!] \supseteq [\![x]\!] \end{array} \right. \right.$$

$$\text{connecting } (\lambda y \text{ to } \lambda x) \quad \left[\; \lambda y \in [\![\lambda y.y0]\!] \wedge \lambda y \in [\![y]\!] \wedge \lambda x \in [\![y]\!] \Rightarrow \left\{ \begin{array}{l} [\![0]\!] \subseteq [\![x]\!] \\ [\![y0]\!] \supseteq [\![x]\!] \end{array} \right. \right.$$

Minimal solution:

$$[\![(\lambda y.y0)(\lambda x.x)]\!] = [\![y0]\!] = [\![0]\!] = [\![x]\!] = \{\mathsf{Int}\}$$
$$[\![\lambda x.x]\!] = [\![y]\!] = \{\lambda x\}$$
$$[\![\lambda y.y0]\!] = \{\lambda y\}$$

Figure 10: SA_R constraints for $(\lambda y.y0)(\lambda x.x)$.

the final conjunct of the conditions on connecting constraints. Only a subset is obtained, because the constraints for dead code may not appear at all in SA_R. It follows that if SA_R is solvable, then so is SA.

The set of global constraints for the term $(\lambda y.y0)(\lambda x.x)$ is presented in figure 10. Notice the similarities and differences from figure 7 which shows the SA constraints for the same term. As an example of a term that is accepted by safety analysis *with* detection of dead code, but is rejected without, consider $\lambda x.00$. In the trace graph there is no edge from MAIN, so the unsafe application 00 is unreachable. This is reflected in the global constraint system that only consists of the one constraint $[\![\lambda x.00]\!] \supseteq \{\lambda x\}$. Clearly, this constraint system is solvable.

3.3 Solving Constraints

We now sketch a cubic time algorithm that computes the minimal solution of an SA constraint system or decides that none exists. We also indicate how it can be modified to handle detection of dead code.

The input to the algorithm is a finite set of constraints where each constraint is of one of the forms: $C \subseteq V$, $V \subseteq V'$, $V \subseteq C$, and $c \in V \Rightarrow V' \subseteq V''$, where c is a lambda

token, C is a set of lambda tokens, and V, V', V'' are variables. Notice that there are $O(n^2)$ constraints if n is the size of the lambda term from which the constraint system was generated.

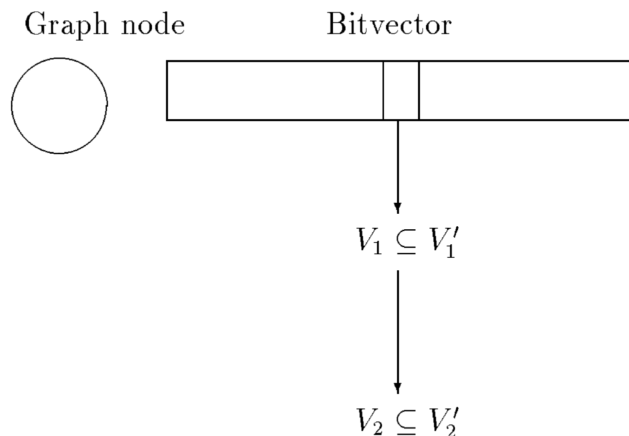
The algorithm has two phases. In the first phase, each constraint, except those of the form $V \subseteq C$, are inserted in a data structure Solver. Notice that when we omit the constraints of the form $V \subseteq C$, then the remaining constraints do have a solution. In the second phase we extract from the Solver the minimal solution of the inserted constraints and check whether the constraints of the form $V \subseteq C$ are satisfied.

During the process of inserting constraints, the Solver represents the minimal solution of the constraints inserted so far. The implementation of the solver uses a directed graph, henceforth called the *graph*.

- *Nodes* of the graph correspond to type variables; and
- *Edges* of the graph correspond to inclusions between type variables.

Graph nodes and type variables are in one-to-one correspondence. The graph node for a type variable represents the set of lambda tokens which the type variable currently is assigned. We represent a set of lambda tokens as a bitvector with an entry for each lambda token.

With each entry in the bitvectors we associate a list of constraints of the form $V' \subseteq V''$. We use this list to handle insertion of constraints of the form $c \in V \Rightarrow V' \subseteq V''$, as explained below. The organization of a graph node can be illustrated as follows.



The idea behind the graph is that when a lambda token is inserted into a set, then all inclusions are automatically maintained. This involves propagating bits along edges. When a bit becomes set, each constraint in the associated list is removed and inserted into the Solver.

Initially, each graph node represents the empty set, and all lists of constraints are empty. We will now consider how to insert constraints.

- First, consider a constraint of the form $C \subseteq V$. We union the set C to the bitvector of the graph node of V, maintain all inclusions, and recursively insert the constraints that are contained in the lists associated with newly set bits.

- Second, consider a constraint of the form $V \subseteq V'$. We create a graph edge from the node of V to the node of V', and do maintenance of inclusions and recursive insertion like in the previous case.

- Third, consider a constraint of the form $c \in V \Rightarrow V' \subseteq V''$. If the set for V contains c, then we recursively insert the constraint $V' \subseteq V''$ into the Solver. Otherwise we insert $V' \subseteq V''$ into the list associated with the bit for c in the graph node for V.

With this implementation of the Solver, the overall time complexity of inserting the $O(n^2)$ constraints into the Solver is $O(n^3)$ where n is the size of the lambda term from which the constraint system was generated. To see this, first observe that each bit is propagated along an edge at most once. Since there are $O(n^2)$ edges, the overall cost of maintaining inclusions is $O(n^3)$ time. Since the remaining work is constant for each constraint, and since there are $O(n^2)$ constraints, we arrive at $O(n^3)$ time.

In the second phase of the algorithm we extract in $O(n^3)$ time the minimal solution from the Solver and we check in $O(n)$ time if the constraints of the form $V \subseteq c$ are satisfied. In total, the algorithm runs in $O(n^3)$ time.

To handle detection of dead code, we modify the first phase of the algorithm as follows.

The idea is to mark each trace graph node as either *dead* or *live*. Initially, only the MAIN node is live.

We extend the Solver with the operation live-trace-graph-nodes. With this operation we can extract the set of trace graph nodes that so far has been marked live. The algorithm also uses a variable L which holds a set of trace graph nodes. The algorithm is as follows:

1. L := $\emptyset$
2. Insert all constraints from the MAIN node into the Solver.
3. while L $\neq$ Solver.live-trace-graph-nodes do
   (a) Choose m from (Solver.live-trace-graph-nodes $\setminus$ L).
   (b) Insert all constraints from m into the Solver.
   (c) L := L $\cup$ {m}
4. end while

The Solver maintains a bitvector representing the set of trace graph nodes marked live. We let each conditional constraint carry the lambda token of the potentially invoked lambda abstraction. Each insertion operation in the Solver can then maintain the set of live trace graph nodes as follows. If we insert a constraint $c \in V \Rightarrow V' \subseteq V''$ into the Solver and the condition $c \in V$ at some point becomes satisfied, then the potentially invoked trace graph node is marked live.

Clearly, the modified algorithm runs in $O(n^3)$ time.

4 Soundness

We now show that safety analysis 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. For simplicity, we prove the soundness of safety analysis for only *closed* terms.

To see that neither of the strict and lazy cases imply the other, consider the two lambda terms in figure 11. 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.

$$
\begin{array}{ll}
1) & (\lambda x.err)(loop) \\
2) & (\lambda x.loop)(err)
\end{array}
$$

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

Figure 11: Two lambda terms.

The two semantics of the untyped lambda calculus will be given as natural semantics [12, 7], 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.

We give the proofs for the safety analysis extended with detection of dead code. This result immediately implies the soundness of the basic safety analysis.

4.1 Strict Semantics

We present in figure 12 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 13.

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_R. 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 $\rho \vdash_{\mathrm{val}} x : v$ is *active*, if it occurs in a derivation tree for $\vdash_{\mathrm{main}} E_0 : w$, for some w, and if E or x occur in a trace

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

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

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

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

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

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

Figure 12: Strict semantics.

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 13: Environments and values.

graph node N where there exists a path from the main node to N whose conditions all hold in L_0.

The predicate $\textsc{abs}(_,_)$ is defined on a constraint variable and value. Intuitively, $\textsc{abs}([\![E]\!], v)$ 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 $\textsc{abs}([\![E]\!], \mathit{wrong})$ always holds.

The E_0-well-formedness (E_0-wf) of environments and values is defined in figure 14. 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 $\textsc{abs}([\![x]\!], v)$.

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 $\text{ABS}([\![x]\!], w)$
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
        - $\text{ABS}([\![E]\!], v)$

Figure 14: E_0-well-formedness.

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 $\text{ABS}([\![x]\!], v)$. 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 $\text{ABS}([\![E_S]\!], v)$.
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 $\rho \vdash \mathsf{0} : 0$ is active, the constraint $[\![\mathsf{0}]\!] \supseteq \{\mathsf{Int}\}$ is satisfied, so $\text{ABS}([\![\mathsf{0}]\!], 0)$. It is immediate that 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 $\text{ABS}([\![\mathsf{succ}\ E]\!], succ^{n+1}\, 0)$. It is immediate that $succ^{n+1}\, 0$ 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 $\text{ABS}([\![\lambda x.E]\!], < \lambda x.E, \rho >)$. 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 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 $\text{ABS}([\![E_1]\!], < \lambda x.E, \rho_1 >)$ and $\text{ABS}([\![E_2]\!], w)$. From $\text{ABS}([\![E_1]\!], < \lambda x.E, \rho_1 >)$ we get that $\lambda x \in [\![E_1]\!]$. This means that $x \mapsto w \cdot \rho_1 \vdash E : v$ 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 $\text{ABS}([\![x]\!], w)$. But $\text{ABS}([\![x]\!], w)$ is unconditionally true, because $\text{ABS}([\![E_2]\!], w)$ 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 $\text{ABS}([\![E]\!], v)$. It thus remains to be shown that $\text{ABS}([\![E_1 E_2]\!], v)$. This follows from $[\![E_1 E_2]\!] \supseteq [\![E]\!]$. □

Lemma 4.3: 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.
Proof: 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 $E_1 E_2$ 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 $\text{ABS}([\![E_2]\!], w)$, 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 $\text{ABS}([\![x]\!], w)$. But $\text{ABS}([\![x]\!], w)$ is unconditionally true, since $\text{ABS}([\![E_2]\!], w)$ and $[\![E_2]\!] \subseteq [\![x]\!]$. □

We first show that the closure analysis is sound.

Theorem 4.4: If $\rho \vdash E : v$ occurs in a derivation tree for $\vdash_{\text{main}} E_0 : w$, for some w, then $\text{ABS}([\![E]\!], v)$.
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. □

We then show that the safety analysis is sound.

Theorem 4.5: If SA_R is solvable and $\vdash_{\text{main}} E_0 : v$, then $v \neq \mathit{wrong}$.
Proof: First note that any solution of SA_R is also a solution of CA_R. 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 was 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 $[\![E]\!] \subseteq \{\mathsf{Int}\}$ holds. This yields a contradiction. □

4.2 Lazy Semantics

We present in figure 15 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.

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

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

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

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

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

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

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

Figure 15: Lazy semantics.

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.	$succ^n\, 0$ 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 *thunk*, *iff*
		• ρ is an environment

Figure 16: Environments and values.

The semantics uses *environments* and *values*, which are simultaneously defined in figure 16.

The new sort of value is that of *thunks*, defined in case 2.d in figure 16. 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, and 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 $\textsc{abs}(_,_)$, 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 $\textsc{abs}([\![E]\!], v)$ holds *iff*

- if $v = succ^n\, 0$ then $\{\mathsf{Int}\} \subseteq [\![E]\!]$,
- 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 17. 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 active, then v is either E_0-wf or *wrong*, and $\textsc{abs}([\![E_S]\!], v)$. Furthermore, 2) if $\rho \vdash_{\text{res}} E_S : v$ is active, then v is either E_0-wf or *wrong*, and $\textsc{abs}([\![E_S]\!], v)$.
Proof: We proceed by induction in the structure of E_S. In the base, we consider x, $\mathsf{0}$, and $\mathsf{succ}\ E$. 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 $[E', \rho']$

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 $x \mapsto w \cdot \rho \vdash_{\mathrm{val}} x : w$ is active, then
        - $\mathrm{ABS}([\![x]\!], w)$.
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
        - $\mathrm{ABS}([\![E]\!], v)$
   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 $\rho \vdash_{\mathrm{res}} E : v$ is active, then
        - v is either E_0-wf or *wrong*, and
        - $\mathrm{ABS}([\![E]\!], v)$

Figure 17: E_0-well-formedness.

is E_0-wf and that $\mathrm{ABS}([\![E]\!], [E', \rho'])$. Hence, $\rho' \vdash_{\mathrm{res}} E' : v$ is active, so v is either E_0-wf or wrong, and $\mathrm{ABS}([\![E']\!], v)$. The conclusion now follows, since $[\![E']\!] \subseteq [\![E]\!]$.

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 $\mathrm{ABS}([\![E_1]\!], < \lambda x.E, \rho_1 >)$. From the latter we get that $\lambda x \in [\![E_1]\!]$. This means that $x \mapsto [E_2, \rho] \cdot \rho_1 \vdash_{\mathrm{val}} E : v$ 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 $x \mapsto [E_2, \rho] \cdot \rho_1$ is E_0-wf we need to prove that $[E_2, \rho]$ is E_0-wf and that if $x \mapsto [E_2, \rho] \cdot \rho_1 \vdash_{\mathrm{val}} x : [E_2, \rho]$ is active, then $\mathrm{ABS}([\![x]\!], [E_2, \rho])$. The first follows by applying the induction hypothesis, case 2), to E_2. The second 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 $\mathrm{ABS}([\![E]\!], v)$. It thus remains to be shown that $\mathrm{ABS}([\![E_1 E_2]\!], v)$. 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 the closure analysis is in the lazy case expressed as follows.

Theorem 4.7: If $\rho \vdash E : v$ occurs in a derivation tree for $\vdash_{\mathbf{main}} E_0 : w$, for some w, then $\text{ABS}([\![E]\!], v)$. Furthermore, if $\rho \vdash_{\mathbf{res}} E : v$ occurs in a derivation tree for $\vdash_{\mathbf{main}} E_0 : w$, for some w, then $\text{ABS}([\![E]\!], v)$.
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 4.6. A similar argument proves the second case. □

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

5 Comparison

We now show that safety analysis accepts strictly more safe terms than does type inference for simple types. We will do this by proving that for every lambda term E_0, if the TI constraint system for E_0 is solvable, then the SA_R constraint system for E_0 is solvable.

The proof involves several lemmas, see figure 18. The main technical problem to be solved is that SA_R and TI are constraint systems over two different domains, sets of closures versus types schemes. This makes a direct comparison hard. We overcome this problem by applying solvability preserving maps into constraints over a common two-point domain.

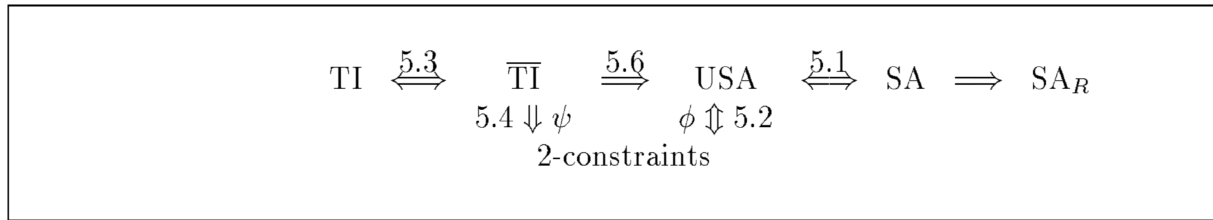


Figure 18: Solvability of constraints for a fixed term E_0.

The entire argument is for a fixed lambda term E_0. It is sufficient to prove that if the TI constraint system for E_0 is solvable, then the SA constraint system for E_0 is solvable. The main result then follows, since if SA is solvable, then so is SA_R.

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 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 5.1: 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 *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:

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

It turns out that ϕ preserves solvability.

Lemma 5.2: USA is solvable *iff* ϕ(USA) is solvable.
Proof: Assume that L is a solution of USA. We construct a solution of ϕ(USA) 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.

Lemma 5.3: TI is solvable *iff* $\overline{\text{TI}}$ is solvable.

Proof: The implication from right to left is immediate. Assume that TI is solvable. Equality is by definition symmetric, 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 2-constraints. Individual constraints are mapped as follows:

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

We show that ψ preserves solvability in one direction.

Lemma 5.4: 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 $L(X) = \mathsf{Int}$, and assigning λ to X otherwise. □

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

Lemma 5.5: 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 $[\![\mathsf{0}]\!] = \{\mathsf{Int}\}$. TI yields the constraint $[\![\mathsf{0}]\!] = \{\mathsf{Int}\}$ which by ψ is mapped to $[\![\mathsf{0}]\!] = \{\mathsf{Int}\}$ as well. A similar argument applies to the constraints yielded for $\mathsf{succ}\ E$, $\lambda x.E$, and $E_1 E_2$, and to possible initial constraints. Thus, we have established the induction base.

For the induction step we assume that $\phi(U) \subseteq \psi(\overline{\text{TI}})$. 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: $[\![x]\!] \subseteq [\![E_2]\!]$ and $[\![E_1 E_2]\!] \supseteq [\![E]\!]$. We must show that the corresponding equalities hold in $\overline{\text{TI}}$. The only way to enable the condition in the minimal solution of U is to have a chain of U-constraints:

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

From the definition of ϕ and ψ and by applying the induction hypothesis, we get 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 5.6: If $\overline{\text{TI}}$ is solvable, then so is USA.
Proof: Assume that $\overline{\text{TI}}$ is solvable. From lemma 5.4 it follows that so is $\psi(\overline{\text{TI}})$. Since from lemma 5.5 $\phi(\text{USA})$ is a subset, it must also be solvable. From lemma 5.2 it follows that USA is solvable. □

We conclude that safety analysis is at least as powerful as type inference for simple types.

Theorem 5.8: Every lambda term accepted by type inference for simple types will also be accepted by both the basic and the extended safety analysis.
Proof: We need only to bring the lemmas together, as indicated in figure 18, and combine them with the observation from section 3 that if the SA constraint system for a lambda term is solvable, then so is the SA_R constraint system for that term. □

We now show that both the basic and the extended safety analyses accept *strictly* more lambda terms than type inference for simple types.

Theorem 5.9: There exists a safe term that is accepted by the basic safety analysis but rejected by type inference for simple types.
Proof: The basic safety analysis accepts all terms without constants. Some of them are rejected by type inference for simple types, for example $\lambda x.xx$. □.

It is easy to see that the safety analysis extended with detection of dead code accepts all terms in normal form that has no safety errors at the outermost level. Type inference for simple types rejects some of these terms, for example $\lambda f.(f\,(\lambda x.x))(f\,\mathsf{0})$.

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

The above proof also sheds some light on why and how safety analysis accepts more safe terms than type inference. Consider a solution of TI that is transformed into a solution of SA according to the strategy implied in figure 18. 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 [2]. 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 5.4 is influenced, but the proof carries through with virtually no modifications. Type inference with recursive types will, like the basic safety analysis, accept all terms without constants. Still, it does not accept for example $\lambda f.(f\,(\lambda x.x))(f\,\mathsf{0})$.

We conclude this section with two example terms that do not have simple types, not even if we allow recursive types, and that are not pure terms or in normal form. The first term is:

$$(\lambda y.y)(\lambda f.(f\,(\lambda x.x))(f\,\mathsf{0}))$$

This term will be accepted by the basic safety analysis, hence also by the extended safety analysis. The second term is:

$$(\lambda f.(f\,(\lambda x.x))(f\,\mathsf{0}))(\lambda y.y)$$

This term will not be accepted by either safety analysis. To see why, observe that no code in this term is dead. Hence, it is sufficient to show that the SA constraint system for the term is unsolvable. Consider then the following subset of this constraint system:

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

Clearly, any solution would have to satisfy:

$$\{\mathsf{Int}\} \subseteq [\![\mathsf{0}]\!] \subseteq [\![y]\!] \subseteq [\![f(\lambda x.x)]\!] \subseteq \{\lambda f, \lambda x, \lambda y\}$$

Since this is impossible, the SA constraint system is unsolvable, hence the term

$$(\lambda f.(f\,(\lambda x.x))(f\,\mathsf{0}))(\lambda y.y)$$

will not be accepted by safety analysis.

6 Conclusion

We have presented a new algorithm, safety analysis, for deciding the safety of lambda terms. It has been proved sound and strictly more powerful than type inference for simple types. The latter result demonstrates that the global safety analysis is more precise than the local type inference. Safety analysis is sound for both lazy and strict semantics, but not for *arbitrary* reduction strategies. For example, the term $\lambda x.\mathsf{00}$ is accepted by safety analysis, but will cause an error if $\mathsf{00}$ is reduced. We conjecture, however, that the basic form of safety analysis, without detection of dead code, is sound for β-reduction.

The algorithm for safety analysis can be implemented in cubic time, by a slight modification of Ayers' algorithm [1]. 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* [9]; so has closure analysis [3]. 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* [21, 17, 14] and *simple intersection types* [5]. 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.

Acknowledgements. The authors thank Mitchell Wand and the anonymous referees for a wealth of helpful comments on drafts of the paper.

References

[1] Andrew Ayers. Efficient closure analysis with reachability. In *Proc. WSA'92, Analyse Statique*, pages 126–134, 1992.

[2] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In *Proc. ESOP'90, European Symposium on Programming*, pages 1–35. Springer-Verlag (*LNCS* 432), 1990.

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

[4] Anders Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, Denmark, April 1993. Included in Similix 5.0 distribution.

[5] Mario Coppo and Paola Giannini. A complete type inference algorithm for simple intersection types. In *Proc. CAAP'92*, pages 102–123. Springer-Verlag (*LNCS* 581), 1992.

[6] Luis Damas and Robin Milner. Principle type-schemes for functional programs. In *Ninth Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, January 1982.

[7] Joëlle Despeyroux. Proof of translation in natural semantics. In *LICS'86, First Symposium on Logic in Computer Science*, pages 193–205, June 1986.

[8] Adele Goldberg and David Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.

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

[10] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathmatical Society*, 146:29–60, 1969.

[11] 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.

[12] Gilles Kahn. Natural semantics. In *Proc. STACS'87*, pages 22–39. Springer-Verlag (*LNCS* 247), 1987.

[13] Paris C. Kannellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan J. Robinson*, chapter 13. MIT Press, 1991.

[14] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences.* To appear. Also in Proc. FOCS'92, 33rd IEEE Symposium on Foundations of Computer Science, pages 363–371, Pittsburgh, Pennsylvania, October 1992.

[15] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

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

[17] 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.

[18] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

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

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

[21] 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.

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