Minimal Typing Derivations

Nikolaj Skallerud Bjørner
Stanford University
nikolaj@cs.stanford.edu

Abstract

We present an algorithm which finds typing derivations for ML typeable expressions, such that polymorphic abstraction is minimized where possible. Consequently, unnecessary boxing or code duplication can be avoided, allowing more efficient ML implementations.

1 Introduction

This paper has been inspired by the use of typing derivations in the analysis of ML programs. In several applications, polymorphism imposes a strong restriction on the kinds of optimizations one can perform. Unboxing analysis [5] is based on typing derivations. It inserts suitable coercions such that boxed value passing is mostly avoided. Further boxing can be reduced by avoiding unnecessary polymorphism. In the specialization of ad-hoc polymorphic primitives (such as equality), more efficient code can be generated when the types of the primitives are known at compile-time [1]. Tag-free garbage collection [9] also benefits from a typing derivation that avoids polymorphism. Finally, extensions of the results presented here indicate ways to resolve instances of overloading resolution, i.e. determining the types of overloaded operators (+,∗) and resolving flex records. A full version of this extended abstract is found in [2].

The type inference algorithm W presented by [4, 3] generates most general type schemes for ML programs, allowing maximal flexibility in program composition. As a side-effect, W also provides a typing derivation or, equivalently, a type annotation. However, the type schemes associated with let-bound variables generally will contain type abstractions where they may not be necessary. For instance, consider the program:

let
  $mkpair$ = $\lambda x.(x,x)$
in
  $mkpair$ 3.14

Algorithm W would produce the type annotation:

let
  $mkpair : \forall\alpha.\alpha \to \alpha \times \alpha$ =
    $\Lambda\alpha.\lambda x : \alpha.(x,x)$
in
  $mkpair$ [$real$] 3.14 : $real \times real$

instead of the more refined

let
  $mkpair : real \to real \times real$ =
    $\lambda x : real.(x,x)$
in
  $mkpair$ 3.14 : $real \times real$

The algorithm M presented in this paper will assign the latter annotation, so that the polymorphic abstraction is avoided.

It turns out that typeable expressions can be conveniently divided into two classes. One of the classes is that of programs containing no *dead code*, i.e. no code redundant in the evaluation of the expression. For this class, we show that there is a unique *minimal* derivation, in the sense that no other typing derivation of the same expression would avoid type abstractions, where the one found could not.

For the other class, consisting of programs containing dead code, or what we will call cut-type variables, unfortunately we can not always find a unique minimal derivation. We argue that programs occurring in practice are in the first class, and therefore the algorithm proposed here is a practical solution to the problem we formulate.

The rest of the paper is organized as follows: In Section 2 we formalize the problem precisely, and introduce the necessary notation. Section 3 summarizes a generalized result on algorithm W's soundness and completeness with respect to typing derivations. Section 4 introduces algorithm M, which finds minimal typing derivations. Section 5 discusses soundness and relative completeness with respect to the notion of cut-type variables. In Section 6 we investigate further the rôle of cut-type variables. Section 7 discusses applications.

2 Problem Formalization

As in the introductory example, we use type-annotated expressions instead of typing derivations. It is simple to establish the equivalence between these two representations [6].

Types and expressions are thus of the form:

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid int \mid real \mid bool \mid \tau \times \tau' \mid \tau \to \tau' \\
\sigma & ::= & \forall\vec{\alpha}.\tau \\
e & ::= & x\,\iota \mid (e,e') \mid e\,e' \mid \lambda x:\tau.e \\
 & & \mid \mathtt{let}\,x:\sigma = \Lambda\vec{\alpha}.e\,\mathtt{in}\,e'
\end{array}
$$

where $\vec{\alpha}$ is a list of type variables and ι is an *instantiation*, i.e. a map from type variables to types. When there is no confusion, instantiations are written as lists of types enclosed in square brackets; if the list is empty, it is simply omitted. We will assume that there are no unrelated uses of the same type variable in e. If the type scheme $\forall\vec{\alpha}.\tau$ is bound to x, the instantiation ι instantiates the variables in $\vec{\alpha}$ to types. With the expression e we associate a type environment, TE, containing the types of e's free variables and some predefined constants, such as the projections π_1 and π_2. Formally, TE is a finite map from variables to type schemes. The statement $\mathrm{TE} \vdash \Lambda\vec{\alpha}.e : \sigma$, is called a *typing judgement*; it asserts that the type annotations in e are well-formed under TE, and that the derivable type-scheme of $\Lambda\vec{\alpha}.e$ is σ. The top level type abstraction $\Lambda\vec{\alpha}$ is used to abstract variables bound in σ. Type substitution into annotated expressions is standard: $\theta\, e$ indicates that θ is applied to every type occurring in e, where the usual renaming and scoping rules apply to Λ-abstracted type variables. A generalization of the standard substitution theorem is now immediate: If the judgement $\mathrm{TE} \vdash e : \tau$ is a well-formed, then $\theta\mathrm{TE} \vdash \theta e : \theta\tau$ is well-formed too.

We take $\overline{e}$ to be the expression where typing information has been erased, i.e.:

$$\overline{e} \;::=\; x \,|\, (\overline{e},\overline{e}') \,|\, \overline{e}\,\overline{e}' \,|\, \lambda x.\overline{e} \,|\, \texttt{let}\ x = \overline{e}\ \texttt{in}\ \overline{e}'$$

Type schemes are ordered as in [3]: let $\sigma = \forall\vec{\alpha}.\tau$ and $\sigma' = \forall\vec{\alpha}'.\tau'$ be type schemes, and let θ be a substitution with $\mathbf{dom}\,\theta \subseteq \vec{\alpha}'$, then: $\sigma <_\theta \sigma'$ iff $\tau = \theta\tau'$ and $\vec{\alpha}$ does not contain variables free in σ'.

This ordering is generalized to an ordering $\preceq_\theta$ over annotations e, e' with the same erasure $\overline{e}$, and free variables typed by TE and TE$'$, respectively:

$$(\mathrm{TE} \vdash x\,\iota) \preceq_\theta (\mathrm{TE}' \vdash x\,\iota') \text{ iff } \mathrm{TE}(x)\iota = \theta\,(\mathrm{TE}'(x)\iota')$$

$$(\mathrm{TE} \vdash e_1\,e_2) \preceq_\theta (\mathrm{TE}' \vdash e_1'\,e_2') \text{ iff } (\mathrm{TE} \vdash e_1) \preceq_\theta (\mathrm{TE}' \vdash e_1') \text{ and } (\mathrm{TE} \vdash e_2) \preceq_\theta (\mathrm{TE}' \vdash e_2')$$

$$(\mathrm{TE} \vdash (e_1,e_2)) \preceq_\theta (\mathrm{TE}' \vdash (e_1',e_2')) \text{ iff } (\mathrm{TE} \vdash e_1) \preceq_\theta (\mathrm{TE}' \vdash e_1') \text{ and } (\mathrm{TE} \vdash e_2) \preceq_\theta (\mathrm{TE}' \vdash e_2')$$

$$(\mathrm{TE} \vdash \lambda x:\tau.e) \preceq_\theta (\mathrm{TE}' \vdash \lambda x:\tau'.e') \text{ iff } (\mathrm{TE}_x \cup \{x:\tau\} \vdash e) \preceq_\theta (\mathrm{TE}'_x \cup \{x:\tau'\} \vdash e') \text{ and } \tau = \theta\,\tau'$$

$$(\mathrm{TE} \vdash \texttt{let}\,x:\sigma = \Lambda\vec{\alpha}\,e_1\,\texttt{in}\,e_2) \preceq_\theta (\mathrm{TE}' \vdash \texttt{let}\,x:\sigma' = \Lambda\vec{\alpha}'e_1'\,\texttt{in}\,e_2') \text{ iff } (\mathrm{TE}_x \cup \{x:\sigma\} \vdash e_2) \preceq_\theta (\mathrm{TE}'_x \cup \{x:\sigma'\} \vdash e_2') \text{ and there is a } \theta' \text{ such that } \mathbf{dom}\,\theta' \subseteq \vec{\alpha}' \text{ and } (\mathrm{TE} \vdash e_1) \preceq_{\theta'} (\theta\mathrm{TE}' \vdash \theta e_1')$$

By a simple induction on the structure of expressions, it follows that in the last rule we furthermore have the constraint $\sigma <_{\theta'} \theta\sigma'$. This establishes the connection between the ordering relation on type schemes and the one defined here for annotated expressions.

When θ is clear from the context, or is irelevant, then we write $\preceq$ instead of $\preceq_\theta$.

A comparison of two annotations illustrates the main ideas of the ordering:

$$\begin{array}{l} \{x:int\} \vdash \texttt{let}\ \ id: int \to int = \lambda x:int.x \\ \qquad\qquad \texttt{in}\ id\,x \\ \preceq_{[\alpha\mapsto int]} \\ \{x:\alpha\} \vdash\ \texttt{let}\ \ id: \forall\beta.\beta \to \beta = \Lambda\beta\lambda x:\beta.x \\ \qquad\qquad \texttt{in}\ id\,[\alpha]\,x \end{array}$$

There are 2 places where the top-most annotation is more specific than the lower. The first place is in the types associated with the free variable x: the type *int* is a substitution instance of the type variable α. Secondly the substitution θ' introduced in the rule for let-expressions is $[\beta \mapsto int]$, thus instantiating the polymorphic abstracted type variable β.

Speaking in more general terms we note that the ordering relation $\preceq$ reflects the degree of type abstraction in typing derivations. The smaller an annotation is in the ordering $\preceq$, the more refined the types are. Ideally, the smallest annotations would favor type constants (such as *real*, *int*) with fixed-length storage requirements, and they would introduce polymorphism only where it is really needed.

Although the requirements imposed by the ordering relation may seem overly restrictive (every annotation in e equals the corresponding annotation in e', modulo a uniform substitution θ) we show that it captures the true properties of principal derivability and minimal derivability: algorithm W finds the greatest element in the ordering, and M will (under certain conditions) find the least element. More precisely, when M is given e, where e does not contain dead code, and the type judgement TE $\vdash \Lambda\vec{\alpha}.e : \forall\vec{\alpha}.\tau$, it finds the least annotation e' preserving $\mathrm{TE} \vdash \Lambda\vec{\alpha}.e' : \forall\vec{\alpha}.\tau$.

In the next section we will shortly motivate the claim about algorithm W, before treating the dual M in depth.

3 Generalized completeness for W

With some trivial modifications of W we obtain an algorithm which, besides producing most general type schemes, also annotates ground expressions $\overline{e}$.

$$\begin{array}{l}
\mathrm{W} : \mathrm{TypeEnvironment} \times \overline{\mathrm{Expr}} \\
\qquad \xrightarrow{\sim} \mathrm{Substitution} \times \mathrm{Expr} \times \mathrm{Type} \\
\mathrm{W}(\mathrm{TE},x) = \\
\quad \textbf{let} \\
\qquad \forall\alpha_1,\ldots,\alpha_n.\tau' = \mathrm{TE}(x) \\
\qquad \beta_1,\ldots,\beta_n \text{ be new type variables} \\
\quad \textbf{in} \\
\qquad (id, x\,[\beta_1,\ldots,\beta_n], [\alpha_i \mapsto \beta_i | i \in \{1,\ldots,n\}]\tau') \\
\mathrm{W}(\mathrm{TE},\lambda x.\overline{e}_1) = \\
\quad \textbf{let} \\
\qquad (S_1, e_1, \tau_1) = \mathrm{W}(\mathrm{TE}_x \cup \{x:\beta\}, \overline{e}_1), \beta \text{ is new} \\
\quad \textbf{in} \\
\qquad (S_1, \lambda x: S_1\beta.e_1, S_1\,\beta \to \tau_1) \\
\mathrm{W}(\mathrm{TE},\overline{e}_1\overline{e}_2) = \\
\quad \textbf{let} \\
\qquad (S_1, e_1, \tau_1) = \mathrm{W}(\mathrm{TE}, \overline{e}_1) \\
\qquad (S_2, e_2, \tau_2) = \mathrm{W}(S_1\mathrm{TE}, \overline{e}_2) \\
\qquad V = \mathrm{unify}(S_2\tau_1, \tau_2 \to \beta), \beta \text{ is new} \\
\quad \textbf{in} \\
\qquad (V S_2 S_1, (V S_2 e_1)\,(V\,e_2), V\,\beta) \\
\mathrm{W}(\mathrm{TE},(\overline{e}_1,\overline{e}_2)) = \\
\quad \textbf{let} \\
\qquad (S_1, e_1, \tau_1) = \mathrm{W}(\mathrm{TE}, \overline{e}_1)
\end{array}$$

$$
\begin{array}{l}
\quad\quad (S_2, e_2, \tau_2) = \mathrm{W}(S_1\mathrm{TE}, \overline{e}_2) \\
\quad \textbf{in} \\
\quad\quad (S_2 S_1, (S_2 e_1, e_2), S_2 \tau_1 \times \tau_2) \\
\mathrm{W}(\mathtt{let}\, x = \overline{e}_1\, \mathtt{in}\, \overline{e}_2) = \\
\quad \textbf{let} \\
\quad\quad (S_1, e_1, \tau_1) = \mathrm{W}(\mathrm{TE}, \overline{e}_1) \\
\quad\quad \forall\vec{\alpha}.\tau_1 = \mathrm{Gen}(S_1\mathrm{TE}, \tau_1) \\
\quad\quad (S_2, e_2, \tau_2) = \mathrm{W}(S_1\mathrm{TE}_x \cup \{x : \forall\vec{\alpha}.\tau_1\}, \overline{e}_2) \\
\quad \textbf{in} \\
\quad\quad (S_2\, S_1, \mathtt{let}\, x : (S_2 \forall\vec{\alpha}.\tau_1) = (S_2 \Lambda\vec{\alpha}\, e_1)\, \mathtt{in}\, e_2, \tau_2)
\end{array}
$$

Note: If unification fails, then the entire algorithm fails.

The established soundness and completeness properties of the original algorithm W can also be generalized appropriately to the modified formulation:

Theorem 1 (Soundness of W) *If W(TE,$\overline{e}$) succeeds with (S, e, τ) then the type annotation $S\,TE \vdash e : \tau$ is well formed.*

The original soundness statement in [3] asserts the existence of a typing derivation. Here this derivation is constructed explicitly in the form of an annotated expression.

When rephrasing the completeness statement, we replace the original ordering on type schemes by the ordering on annotated expressions. Thus the principal typing property generalizes neatly to principal typing annotation.

Theorem 2 (Completeness of W) *Given TE and $\overline{e}$, let TE′ be an instance of TE and e′ be an annotation of $\overline{e}$ such that*

$$\mathrm{TE}' \vdash e' : \tau'$$

is a well-formed annotation (with external type τ'). Then (i) W(TE,$\overline{e}$) succeeds (ii) If W(TE,$\overline{e}$)= (S, e, τ) then, for some substitution R, $\mathrm{TE}' = RS\,(\mathrm{TE})$ *and*

$$\mathrm{TE}' \vdash e' \preceq_R S\,\mathrm{TE} \vdash e$$

A simple rephrasing of the original proof of W's completeness establishes this theorem. The original proof can be found in [3] and the detailed rephrasing is found in [2]. Here, however, we will mainly concentrate on the development of algorithm M.

4 Minimal Derivations via Algorithm M

Dual to W's dependence on unification, M will heavily rely on anti-unification. The original algorithms for anti-unification are given in [7, 8]. Here, we assume that we are given an algorithm AntiUnify: Type* $\rightarrow$ Type $\times$ Substitution*, which takes a finite non-empty list of types $\langle\tau_1, \ldots, \tau_n\rangle$ and returns a type τ and substitutions $\langle\theta_1, \ldots, \theta_n\rangle$ with the same domains, such that $\tau_i = \theta_i \tau$, and τ is the most specific generalizer of $\langle\tau_1, \ldots, \tau_n\rangle$. Unlike unifiers, generalizers always exist.

The idea underlying M is quite simple: A recursive descent through the expression e collects for each let-bound variable $x : \sigma$ all the associated type instantiations $\iota_1, \ldots, \iota_n$. When σ is applied to these instantiations, a list of types is formed ($\tau_1 = \sigma\iota_1, \ldots, \tau_n = \sigma\iota_n$). Using anti-unification on this list makes it possible to extract a least general type scheme for x.

M's auxiliary result A collects the instantiations of let-bound variables. It is a map from variables to lists of instantiations. Two maps A and A' are merged by the appending operation $A + A' \stackrel{\text{def}}{=} [x \mapsto A(x)@A'(x)]$, where we define $A(x) \stackrel{\text{def}}{=} \langle\rangle$ if $x \notin \mathbf{dom}\, A$. To ensure that type variables are abstracted appropriately, M also collects bound type variables in the set B.

The algorithm is described more precisely below:

$$
\begin{array}{l}
\mathrm{M} : \mathrm{Expr} \rightarrow \mathrm{Expr} \times \mathrm{InstEnvironment} \times \mathrm{TyVar}\mathbf{-set} \\
\mathrm{M}(x\,\iota) = (e, [x \mapsto \langle\iota\rangle], \{\,\}) \\
\mathrm{M}(\lambda x : \tau.e_1) = \\
\quad \textbf{let} \\
\quad\quad (e_1', A, B) = \mathrm{M}(e_1) \\
\quad \textbf{in} \\
\quad\quad (\lambda x : \tau.e_1', A \setminus \{x\}, B) \\
\mathrm{M}(e_1\, e_2) = \\
\quad \textbf{let} \\
\quad\quad (e_1', A_1, B_1) = \mathrm{M}(e_1) \\
\quad\quad (e_2', A_2, B_2) = \mathrm{M}(e_2) \\
\quad \textbf{in} \\
\quad\quad (e_1'\, e_2', A_1 + A_2, B_1 \cup B_2) \\
\mathrm{M}((e_1, e_2)) = \\
\quad \textbf{let} \\
\quad\quad (e_1', A_1, B_1) = \mathrm{M}(e_1) \\
\quad\quad (e_2', A_2, B_2) = \mathrm{M}(e_2) \\
\quad \textbf{in} \\
\quad\quad ((e_1', e_2'), A_1 + A_2, B_1 \cup B_2) \\
\mathrm{M}(\mathtt{let}\, x : \forall\vec{\alpha}.\tau = \Lambda\vec{\alpha}\, e_1\, \mathtt{in}\, e_2) = \\
\quad \textbf{if}\ x \notin FV(e_2)\ \textbf{then} \\
\quad \textbf{let} \\
\quad\quad (e_2', A_2, B_2) = \mathrm{M}(e_2) \\
\quad\quad (e_1', A_1, B_1) = \mathrm{M}(e_1) \\
\quad \textbf{in} \\
\quad\quad (\mathtt{let}\, x : \forall\vec{\alpha}.\tau = \Lambda\vec{\alpha}\, e_1'\, \mathtt{in}\, e_2', \\
\quad\quad A_1 + (A_2 \setminus \{x\}), B_1 \cup B_2 \cup \vec{\alpha}) \\
\quad \textbf{else let} \\
\quad\quad (e_2', A_2, B_2) = \mathrm{M}(e_2) \\
\quad\quad \langle\tau_1, \ldots, \tau_n\rangle = \mathrm{map}\ (\lambda\iota.(\forall\vec{\alpha}.\tau)\,\iota)\ A_2(x) \\
\quad\quad (\tau', \rho) = \mathrm{AntiUnify}\langle\tau_1, \ldots, \tau_n\rangle \\
\quad\quad \{\gamma_1, \ldots, \gamma_m\} = \mathrm{FV}(\tau') \cap B_2 \\
\quad\quad \gamma_1', \ldots, \gamma_m'\ \text{be new variables.} \\
\quad\quad \tau'' = [\gamma_i \mapsto \gamma_i']\tau' \\
\quad\quad \theta = \mathrm{unify}_{\vec{\alpha}}(\tau, \tau'') \\
\quad\quad \vec{\beta} = \mathbf{dom}\ \rho\ \cup \{\gamma_1', \ldots, \gamma_m'\} \\
\quad\quad (e_1', A_1, B_1) = \mathrm{M}(\theta\, e_1) \\
\quad \textbf{in} \\
\quad\quad (\mathtt{let}\, x : \forall\vec{\beta}.\theta\tau = \Lambda\vec{\beta}\, e_1'\ \mathtt{in}\ subst(x, \theta, \vec{\beta}, e_2'), \\
\quad\quad A_1 + (A_2 \setminus \{x\}), B_1 \cup B_2 \cup \vec{\beta})
\end{array}
$$

The auxiliary function *subst* updates the instantiations of x with respect to the new type-scheme. Hence each instantiation ι associated with an occurrence of x in e_2' is changed to the instantiation $\bigcup[\mathrm{unify}_{\vec{\beta}}(\theta\,\alpha, \iota'\,\alpha) | \alpha \in \mathbf{dom}\,\theta]$, where the subscript $\vec{\beta}$ indicates which variables are unifiable. For example let $\theta = [\alpha \mapsto \beta]$, $\vec{\beta} = \{\beta, \gamma\}$ and $e_2' = \lambda y : \tau.x\,[\alpha \mapsto int]$, then $subst(x, \theta, \vec{\beta}, e_2') = \lambda y : \tau.x\,[\beta \mapsto int]$.

More formally, we define *subst* as follows.

$$
\begin{array}{l}
\mathrm{subst} : \mathrm{Var} \times \mathrm{Substitution} \times \mathrm{TyVar}\mathbf{-set} \times \mathrm{Expr} \rightarrow \mathrm{Expr} \\
\mathrm{subst}(x, \theta, \vec{\beta}, x'\,\iota') = \\
\quad \textbf{if}\ x = x'
\end{array}
$$

$\quad\quad$ **then** $x \bigcup\{\text{unify}_{\vec{\beta}}(\theta\,\alpha, \iota'\,\alpha) | \alpha \in \mathbf{dom}\,\theta\}$
$\quad\quad$ **else** $x'\,\iota'$
$\text{subst}(x, \theta, \vec{\beta}, \lambda x' : \tau.e_1) =$
$\quad$ **if** $x = x'$
$\quad\quad$ **then** $\lambda x' : \tau.e_1$
$\quad\quad$ **else** $\lambda x' : \tau.\text{subst}(x, \theta, \vec{\beta}, e_1)$
$\text{subst}(x, \theta, \vec{\beta}, e_1\, e_2) = \text{subst}(x, \theta, \vec{\beta}, e_1)\ \text{subst}(x, \theta, \vec{\beta}, e_2)$
$\text{subst}(x, \theta, \vec{\beta}, (e_1, e_2)) = (\text{subst}(x, \theta, \vec{\beta}, e_1),\ \text{subst}(x, \theta, \vec{\beta}, e_2))$
$\text{subst}(x, \theta, \vec{\beta}, \mathtt{let}\, x' : \sigma = \Lambda\vec{\alpha}\, e_1\, \mathtt{in}\, e_2) =$
$\quad$ **if** $x = x'$
$\quad\quad$ **then** $\mathtt{let}\, x' : \sigma = \Lambda\vec{\alpha}\, \text{subst}(x, \theta, \vec{\beta}, e_1)\, \mathtt{in}\, e_2$
$\quad\quad$ **else**
$\quad\quad$ $\mathtt{let}\, x' : \sigma = \Lambda\vec{\alpha}\, \text{subst}(x, \theta, \vec{\beta}, e_1)\, \mathtt{in}\, \text{subst}(x, \theta, \vec{\beta}, e_2)$

When applying algorithm M at the top level, there is no need for the extra results A and B, we therefore write $\text{M}(e)$ instead of **let** $(e', A, B) = \text{M}(e)$ **in** e' to save notation.

5 Soundness and Completeness

In this section we will prove the soundness of algorithm M, and identify a class of expressions, for which M is complete.

5.1 Soundness

Algorithm M is sound, in the sense that given a well formed type annotation $\text{TE} \vdash \Lambda\vec{\alpha}.e : \sigma$ it produces another well formed annotation preserving the outer type scheme.

Theorem 3 (Soundness of M) *Let σ be of the form $\forall\vec{\alpha}.\tau$. If $\text{TE} \vdash \Lambda\vec{\alpha}.e : \sigma$, then $\text{TE} \vdash \Lambda\vec{\alpha}.M(e) : \sigma$.*

The relatively easy proof of soundness uses a more subtle soundness of *subst*.

Lemma 1 *Let $\sigma = \forall\vec{\alpha}.\tau$, $\sigma' = \forall\vec{\beta}.\tau'$ and BV be the set of bound type variables in the expression e. Furthermore let $\iota_1, \ldots, \iota_n$ be the instantiations of the type-scheme σ of the variable x in e.*
If

1. $\text{TE}_x \cup \{x : \sigma\} \vdash e : \nu$
2. $\text{FV}(\sigma')$ *is disjoint from* BV
3. $\sigma' <_\theta \sigma$ *and each of* $\iota_1\tau$ *to* $\iota_n\tau$ *is an instance of* σ'

then the annotation induced by subst is well formed and

$$\text{TE}_x \cup \{x : \sigma'\} \vdash \text{subst}(x, \theta, \vec{\beta}, e) =_{id} \text{TE}_x \cup \{x : \sigma\} \vdash e$$

In the lemma $=_{id}$ abbreviates $\preceq_{id} \wedge \succeq_{id}$, where id is the identity substitution.

The proof of this lemma is by routine induction over the structure of $\overline{e}$.

It is now also routine to prove the soundness of M, similarly over the structure of $\overline{e}$. The most involved case is where e is of the form $\mathtt{let}\, x : \forall\vec{\alpha}.\nu = \Lambda\vec{\alpha}\, e_1\, \mathtt{in}\, e_2$, where $x \in FV(e_2)$, and we have the assumption $\text{TE} \vdash e : \tau$. Recursive calls of M produce the expressions e_1' and e_2', the set $\vec{\beta} = \mathbf{dom}\, m$ and the substitution θ.

By the substitution theorem and, the fact that **dom** $\theta \cap \text{FV(TE)} = \{\}$, (since **dom** $\theta = \vec{\alpha}$ and generalization upon free variables in the type environment is excluded) we have:

$$\text{TE} \vdash \theta e_1 : \theta\nu$$

which by the induction hypothesis gives

$$\text{TE} \vdash e_1' : \theta\nu$$

The induction hypothesis furthermore ensures

$$\text{TE}_x \cup \{x : \forall\vec{\alpha}.\nu\} \vdash e_2' : \tau$$

and the formation of θ, $BV = B_2$, $\sigma' = \forall\vec{\beta}.\theta\nu$ in M, ensures that the conditions of lemma 2 hold, so:

$$\text{TE}_x \cup \{x : \forall\vec{\beta}.\theta\nu\} \vdash \text{subst}(x, \theta, \vec{\beta}, e_2') : \tau$$

□

5.2 Cut-type variables

Algorithm M is complete only for programs not containing dead code. To make this notion precise, we introduce the notion of cut-type variables, which capture formally what is meant by dead code.

Define inductively the (free) type variables of e by:

$\text{TV}(\text{TE}, x\,\iota) = \text{FV}(\text{TE}(x)\,\iota)$
$\text{TV}(\text{TE}, e_1\, e_2) = \text{TV}(\text{TE}, e_1) \cup \text{TV}(\text{TE}, e_2)$
$\text{TV}(\text{TE}, (e_1, e_2)) = \text{TV}(\text{TE}, e_1) \cup \text{TV}(\text{TE}, e_2)$
$\text{TV}(\text{TE}, \lambda x : \tau.e) = \text{FV}(\tau) \cup \text{TV}(\text{TE}_x \cup \{x : \tau\}, e)$
$\text{TV}(\text{TE}, \mathtt{let}\, x : \forall\vec{\alpha}.\tau = \Lambda\vec{\alpha}.e_1\, \mathtt{in}\, e_2) =$
$\quad \text{TV}(\text{TE}, e_1)\backslash\vec{\alpha} \cup \text{TV}(\text{TE}_x \cup \{x : \forall\vec{\alpha}.\tau\}, e_2)$

If the judgement $\text{TE} \vdash e : \tau$ holds, it is trivial to verify that $\text{FV}(\tau) \subseteq \text{TV}(\text{TE}, e)$. The converse inclusion is not always true, as illustrated in Section 6, thus leading to the notion of cut-type variables. The set of cut-type variables of $\text{TE} \vdash e : \tau$ is the set:

$$\text{TV}(\text{TE}, e) \setminus (\text{FV}(\tau) \cup \text{FV}(\text{TE})).$$

In words: it is the set of hidden type variables of e, i.e. those free type variables occuring free in the annotated expression e, but neither visible in the type environment TE, nor the resulting type τ.

It is easy to decide whether a given expression $\overline{e}$ and type environment TE have an annotation containing cut-type variables. Since our version of algorithm W gives a most general annotation of an expression $\overline{e}$, with type environment TE, any other annotation of $\overline{e}$ will be a substitution instance of the one found by W. One may therefore analyze the result of W for containing cut-type variables or having unused let-bound variables with associated type-schemes.

As the cut-type variables are invisible in e's type τ, the code associated with these variables represents redundant β-redexes, typically of the form of a projection $\pi_2(e_1, e_2)$, where e_1 contains cut-type variables, or of the form $\mathtt{let}\, x : \tau = e_1\, \mathtt{in}\, e_2$ where $x \notin FV(e_2)$, which is treated as a special case in algorithm M.

In proof-theoretic terms, cut-type variables correspond to cuts on formulas which are not in the language of the final sequent. Similarly, it is easy to show that expressions in normal form do not contain cut-type variables. In the formulas-as-types analogy this amounts to the fact that normal form expressions represent cut-free proofs.

If it is not possible to annotate an expression $\overline{e}$ with cut-type variables, then there is only one way to annotate the 'let-free' subexpressions, i.e. subexpressions $\overline{e}_2$ for which $\mathtt{let}\, x = \overline{e}_1 \,\mathtt{in}\, \overline{e}_2$ is a let-free subexpression of $\overline{e}$. To make this statement precise, we introduce the equivalence relation:

$$
\begin{aligned}
&\mathrm{TE} \vdash x\,\iota \approx \mathrm{TE}' \vdash x\,\iota' \text{ iff} \\
&\qquad \mathrm{TE}(x)\iota = \mathrm{TE}'(x)\iota'
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{TE} \vdash e_1\, e_2 \approx \mathrm{TE}' \vdash e_1'\, e_2' \text{ iff} \\
&\qquad \mathrm{TE} \vdash e_1 \approx \mathrm{TE}' \vdash e_1' \\
&\quad \text{and } \mathrm{TE} \vdash e_2 \approx \mathrm{TE}' \vdash e_2'
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{TE} \vdash (e_1, e_2) \approx \mathrm{TE}' \vdash (e_1', e_2') \text{ iff} \\
&\qquad \mathrm{TE} \vdash e_1 \approx \mathrm{TE}' \vdash e_1' \\
&\quad \text{and } \mathrm{TE} \vdash e_2 \approx \mathrm{TE}' \vdash e_2'
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{TE} \vdash \lambda x : \tau . e \approx \mathrm{TE}' \vdash \lambda x : \tau' . e' \text{ iff} \\
&\qquad \mathrm{TE}_x \cup \{x : \tau\} \vdash e \approx \mathrm{TE}'_x \cup \{x : \tau'\} \vdash e' \\
&\quad \text{and } \tau = \tau'
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{TE} \vdash \mathtt{let}\, x : \sigma = \Lambda\vec{\alpha}\, e_1 \,\mathtt{in}\, e_2 \approx \\
&\mathrm{TE}' \vdash \mathtt{let}\, x : \sigma' = \Lambda\vec{\alpha}'\, e_1' \,\mathtt{in}\, e_2' \text{ iff} \\
&\qquad \mathrm{TE}_x \cup \{x : \sigma\} \vdash e_2 \approx \mathrm{TE}'_x \cup \{x : \sigma'\} \vdash e_2'
\end{aligned}
$$

Notice that the relation $=_{id} = \preceq_{id} \wedge \succeq_{id}$ is finer than $\approx$, that is: $\mathrm{TE} \vdash e =_{id} \mathrm{TE}' \vdash e'$ implies $\mathrm{TE} \vdash e \approx \mathrm{TE}' \vdash e'$. A more useful observation is that:

Lemma 2 *Assume there are no annotations of $\overline{e}$ under TE containing cut-type variables. Let TE $\vdash e : \tau$ and TE $\vdash e' : \tau$ be two annotations of $\overline{e}$ resulting in the same type τ. Then:*

$$\mathrm{TE} \vdash e \approx \mathrm{TE} \vdash e'$$

Again the proof is by structural induction on $\overline{e}$. The cases for variables, pairs and λ-abstractions are straightforward. Let-expressions and applications require some care, so we sketch the principal ideas:

Let $e = \mathtt{let}\, x : \sigma = \Lambda\vec{\alpha}\, e_1 \,\mathtt{in}\, e_2$, and $e' = \mathtt{let}\, x : \sigma' = \Lambda\vec{\beta}\, e_1' \,\mathtt{in}\, e_2'$. By the principal typing property there is a type-scheme $\sigma_{\mathrm{p}} = \forall\vec{\alpha}_{\mathrm{p}}.\nu_{\mathrm{p}}$ for $\overline{e}_1$ under TE that is more general than $\sigma = \forall\vec{\alpha}.\nu$ and $\sigma' = \forall\vec{\beta}.\nu'$, i.e. there are substitutions θ_1 and θ_2 satisfying: $\nu = \theta_1\nu_{\mathrm{p}}$ and $\nu' = \theta_2\nu_{\mathrm{p}}$. It follows that there are annotations e_3 and e_3' of $\overline{e}_2$ such that $\mathrm{subst}(x, \theta_1, \vec{\alpha}, e_3) = e_2$ and $\mathrm{subst}(x, \theta_2, \vec{\beta}, e_3') = e_2'$. Establishing the result is now immediate:

$$\mathrm{TE}_x \cup \{x : \sigma\} \vdash e_2$$

is by Lemma 1 $\approx$ to:

$$\mathrm{TE}_x \cup \{x : \sigma_{\mathrm{p}}\} \vdash e_3$$

which by the induction hypothesis is $\approx$ to:

$$\mathrm{TE}_x \cup \{x : \sigma_{\mathrm{p}}\} \vdash e_3'$$

which as again by Lemma 1 is $\approx$ to:

$$\mathrm{TE}_x \cup \{x : \sigma'\} \vdash e_2'$$

When $e = e_1\, e_2$, and $e' = e_1'\, e_2'$ we may assume we are given well-formed typing judgements, $\mathrm{TE} \vdash e_1 : \tau_1 \to \tau$, $\mathrm{TE} \vdash e_2 : \tau_1$, $\mathrm{TE} \vdash e_1' : \tau_1' \to \tau$ and $\mathrm{TE} \vdash e_2' : \tau_1'$, but τ_1 and τ_1' could be different. We will show that they are equal, so one can apply the induction hypothesis on the subexpressions to obtain the result.

By the principal typing property, there are principal annotations such that $\mathrm{TE} \vdash \overline{1}_2 : \tau_{1p} \to \tau_p$ and $\mathrm{TE} \vdash \overline{e}_2 : \tau_{1p}$ are well formed judgements. Using the existence of an most specific generalizer τ_{c} for τ_1 and τ_1', which is subsumed by τ_{1p} it is possible to conclude that there is an annotation leading to the well formed judgements: $\mathrm{TE} \vdash e_{1\mathrm{c}} : \tau_{\mathrm{c}} \to \tau$ and $\mathrm{TE} \vdash e_{2\mathrm{c}} : \tau_{\mathrm{c}}$.

Now $\mathrm{FV}(\tau_{\mathrm{c}}) \subseteq \mathrm{TV}(\mathrm{TE}, e_{2\mathrm{c}}) \subseteq \mathrm{TV}(\mathrm{TE}, e_{1\mathrm{c}}\, e_{2\mathrm{c}}) \subseteq \mathrm{FV}(\mathrm{TE}) \cup \mathrm{FV}(\tau)$, as we apply the hypothesis of the Lemma. Hence, as the substitutions θ, and θ' satisfying $\theta(\tau_{\mathrm{c}} \to \tau) = \tau_1 \to \tau$ and $\theta'(\tau_{\mathrm{c}} \to \tau) = \tau_1' \to \tau$, may not involve type variables from $\mathrm{FV}(\mathrm{TE}) \cup \mathrm{FV}(\tau)$, it follows that $\tau_1 = \tau_1' = \tau_{\mathrm{c}}$.

We can now apply the induction hypothesis to e_1 and e_2 and draw the desired conclusion. □

5.3 Relative Completeness

Using the Lemma 2 above, the relative completeness of M can now be stated and proved:

Theorem 4 (Relative Completeness of M) *Suppose that the judgements $\mathrm{TE} \vdash \Lambda\vec{\alpha}.e : \sigma$ and $\mathrm{TE} \vdash \Lambda\vec{\alpha}.e' : \sigma$ hold, and that there are no typing derivations for $\overline{e} = \overline{e}'$ that introduce cut-type variables. Then*

$$\mathrm{TE} \vdash \mathrm{M}(e) \preceq_{id} \mathrm{TE} \vdash e'$$

Again we sketch the main ideas of the proof, which proceeds by induction on the structure of $\overline{e}$. For variables, abstractions and pairs the induction argument is straightforward. Lemma 2 is crucial for applications: since $\mathrm{TE} \vdash e_1\, e_2 \approx \mathrm{TE} \vdash e_1'\, e_2'$ is given, it follows that e_1 and e_1', respectively e_2 and e_2' result in the same types, the case can therefore be established by a straight forward application of the induction hypothesis. Let-expressions are the most elaborate to handle.

Thus assume that $e = \mathtt{let}\, x : \sigma = \Lambda\vec{\alpha}\, e_1 \,\mathtt{in}\, e_2 : \tau$ and $e' = \mathtt{let}\, x : \sigma = \Lambda\vec{\alpha}\, e_1' \,\mathtt{in}\, e_2' : \tau$, where $\sigma = \forall\vec{\alpha}.\nu$ and $\sigma' = \forall\vec{\gamma}.\nu'$. By the principal typing property a principal type-scheme, say $\sigma_{\mathrm{p}} = \forall\vec{\alpha}_{\mathrm{p}}.\nu_{\mathrm{p}}$, is associated with $\overline{e}_1$ under TE, such that substitutions ψ and ψ' establish $\sigma <_{\psi} \sigma_{\mathrm{p}}$ and $\sigma' <_{\psi'} \sigma_{\mathrm{p}}$. Hence instead of using σ and σ' to type e_2 and e_2', one could use σ_{p} on expressions e_3 and e_3' in which only the instantiation of x had been changed from e_2 and e_2' (like done in the proof of the previous theorem). More precisely the expressions would be interrelated as:

$$
\begin{aligned}
\mathrm{subst}(x, \psi, \vec{\alpha}, e_3) &= e_2 \\
\mathrm{subst}(x, \psi', \vec{\gamma}, e_3') &= e_2'
\end{aligned}
$$

moreover $(*)$:

$$
\begin{aligned}
&\mathrm{TE}_x \cup \{x : \sigma_p\} \vdash \mathrm{M}(e_3) \\
=\ &\mathrm{TE}_x \cup \{x : \sigma\} \vdash \mathrm{subst}(x, \psi, \vec{\alpha}, \mathrm{M}(e_3)) \\
=\ &\mathrm{TE}_x \cup \{x : \sigma\} \vdash \mathrm{M}(e_2)
\end{aligned}
$$

as we use Lemma 1 for the first equality and, since e_2 and e_3 only differ with respect to the type-scheme of x, we can easily justify the second equality.

Recursive calls of M on e_2 also result in the substitution θ and the set of type variables $\vec{\beta}$ used in the resulting expression, where we can reason:

$$\begin{array}{lll} & \mathrm{TE}_x \cup \{x : \forall\vec{\beta}.\theta\nu\} & \vdash \mathrm{subst}(x,\theta,\vec{\beta},\mathrm{M}(e_2)) = \\ \text{by Lemma 2} & & \\ & \mathrm{TE}_x \cup \{x : \sigma\} & \vdash \mathrm{M}(e_2) = \\ \text{by } (*) & & \\ & \mathrm{TE}_x \cup \{x : \sigma_{\mathrm{P}}\} & \vdash \mathrm{M}(e_3) \preceq_{id} \\ \text{by the induction hypothesis} & & \\ & \mathrm{TE}_x \cup \{x : \sigma_{\mathrm{P}}\} & \vdash e_3' = \\ \text{by Lemma 2} & & \\ & \mathrm{TE}_x \cup \{x : \sigma'\} & \vdash e_2' \end{array}$$

We will now establish the existence of a substitution θ' such that $\forall\vec{\beta}.\theta\nu <_{\theta'} \sigma'$. Let $\tau_1, \ldots, \tau_n$ be the instantiated types of x in $\mathrm{subst}(x,\theta,\vec{\beta},\mathrm{M}(e_2))$, and $\tau_1', \ldots, \tau_n'$ be the corresponding types of x in e_2'. The following properties now hold:

1. $\forall i : \tau_i < \forall\vec{\beta}.\theta\nu$
2. $\forall i : \tau_i' < \sigma'$
3. $\exists\theta_1, \ldots \theta_n : \forall i : \tau_i = \theta_i\tau_i'$ and $\mathbf{dom}\theta_i \cap \mathrm{FV}(\mathrm{TE}_x \cup \{x : \sigma'\}) \cup \mathrm{FV}(\tau) = \emptyset$

The first two hold, since each τ_i is an instance of the type scheme associated with the let-bound variable x. The last holds since we have established $\mathrm{subst}(x,\theta,\vec{\beta},\mathrm{M}(e_2)) \preceq_{id} e_2'$.

From 2 and 3 we conclude that

$$\forall i : \tau_i < \sigma'$$

Hence σ' is an upper bound for the τ_i's. However with the antiunification applied in algorithm M, we constructed $\forall\vec{\beta}.\theta\nu$ to be the least upper bound for the τ_i's. It follows that there is a θ', such that $\forall\vec{\beta}.\theta\nu <_{\theta'} \sigma'$.

The last steps in the proof can now be applied:

$$\begin{array}{lll} \mathrm{TE} \vdash \mathrm{M}(\theta e_1) & \preceq_{id} & \text{by the induction hypothesis} \\ \mathrm{TE} \vdash \theta' e_1' & \preceq_{\theta'} & \text{since } \mathbf{dom}\ \theta' \cap \mathrm{FV}(\mathrm{TE}) = \{\} \\ \mathrm{TE} \vdash e_1' & & \end{array}$$

In summary we have shown that:

$$\mathrm{TE} \vdash \mathrm{M}(e) \quad \preceq_{id} \quad \mathrm{TE} \vdash e'$$

which completes this case, and hereby the proof. □

A direct corollary is that $e_M = \mathrm{M}(e)$ is the minimal typing derivation for $\overline{e}$ when the associated type-scheme of $\Lambda\vec{\alpha}.e_M$ is required to be σ.

The theorem is stated and proved for an arbitrary derivable type scheme σ for $\overline{e}$ under TE. If in particular we seek the minimal annotation for $\overline{e}$ supporting the principal type-scheme of $\overline{e}$, we can first apply W, to obtain an annotation, and then M to eliminate redundant polymorphism.

6 Presence of Cut-type Variables

The condition on cut-type variables might have seemed not to play a central rôle in the completeness proof. One might therefore hope to be able to do without it. However the conditions is really necessary as illustrated by the following examples:

$$\begin{array}{l} \texttt{let} \\ \quad f = \lambda x.x, \\ \quad g = \lambda x.(x,x) \\ \texttt{in} \\ \quad \pi_2(\lambda x.((f\,x, f\,3),(g\,x, g\,\mathrm{true})), 4) \end{array}$$

In this example there are two minimal but incompatible typing derivations. The derivation found by W assigns polymorphic types $\forall\alpha.\alpha \to \alpha$ to f and $\forall\beta.\beta \to \beta\times\beta$ to g. A cut-type variable δ is introduced in the part:

$$\lambda x : \delta.((f\,[\delta]\,x, f\,[int]\,3),(g[\delta]\,x, g\,[bool]\,\mathrm{true}))$$

By substituting either int for δ or $bool$ for δ, one can either specialize f to the monomorphic type $int \to int$, or g to $bool \to bool \times bool$, but not both.

For an example, where there is no minimal derivation at all, take:

$$\begin{array}{l} \texttt{let} \\ \quad id = \lambda x.x, \\ \texttt{in} \\ \quad \pi_2(\lambda x_1.(id\,x_1, id\,\lambda x_2.\pi_1(x_1, x_2+1)), 0) \end{array}$$

in which id can be typed:

$$\begin{array}{ll} & \forall\alpha.\alpha \to \alpha \\ > & \forall\alpha.(int \to \alpha) \to int \to \alpha \\ > & \forall\alpha.(int \to int \to \alpha) \to int \to int \to \alpha \\ > & \forall\alpha.(int \to int \to int \to \alpha) \\ & \quad \to int \to int \to int \to \alpha \\ > & \ldots \end{array}$$

6.1 How to fail minimality

The two examples above capture precisely the only ways minimality may fail. They correspond closely to the ways unification may fail: by name clash and occurs check. The first example illustrates name clash; the latter illustrates occurs check failure.

To illustrate this claim we regard the instantiations $\tau_1, \ldots, \tau_n$ of a type-scheme σ, and search for the least generalizer of those. In the algorithm we obtain the generalizer τ by applying $\mathrm{AntiUnify}\langle\tau_1, \ldots, \tau_n\rangle$. However, as the examples illustrate, the types $\tau_1, \ldots, \tau_n$ may contain cut-type variables, which we can instantiate without affecting the type at the top level. Hence there could very well be a substitution θ of the cut-type variables, such that $\tau > \mathrm{AntiUnify}\langle\theta\tau_1, \ldots, \theta\tau_n\rangle$.

For an example where there is a unique such substitution, take the expression annotated by W:

$$\begin{array}{l} \texttt{let} \\ \quad zero : \forall\alpha.\alpha \to int = \Lambda\alpha\lambda x : \alpha.0 \\ \texttt{in} \\ \quad \pi_2(\lambda x : \delta.zero\,[\delta]\,x, zero\,[real]\,3.14) : int \end{array}$$

Here δ is a cut-type variable and the type scheme associated with $zero$ is instantiated to $\tau_1 = \gamma \to int$ and $\tau_2 = real \to int$. In algorithm M we find $\mathrm{AntiUnify}\langle\tau_1,\tau_2\rangle = \beta \to int$. However, when we instantiate the cut-type variable γ by $\theta = [\delta \mapsto real]$, then $\mathrm{AntiUnify}\langle\theta\tau_1,\theta\tau_2\rangle = real \to int$.

The two previous examples in this section illustrate that there might not be a unique substitution θ yielding minimal $\mathrm{AntiUnify}\langle\theta\tau_1, \ldots \theta\tau_n\rangle$.

The first example illustrated name clash. In the attempt to find a θ, (with domain $\{\delta\}$) such that each row

$$\begin{array}{ll}\delta \to \delta, & int \to int \\ \delta \to \delta, & bool \to bool\end{array}$$

would have least possible generalizers, we failed because the type constants *int* and *bool* could not be unified. More advanced type systems might be able to optimize on this case when regarding *bool* as a subtype of *int*.

By the second example, occurs check failure is illustrated. We tried to solve the system

$$\delta \to \delta, \quad (int \to \delta) \to int \to \delta$$

but failed, since we could iterate the substitution $[\delta \mapsto (int \to \delta)]$ and continue getting more and more refined instantiations.

In the full paper we attempt to optimize M to treat examples where proper instantiation of cut-type variables leads to more refined annotations. However the algorithm is quite involved, and it is not clear that we can always find optimal solutions there. This contrasts with the easy implementation of M and its applications.

7 Applications

Algorithm M is designed to serve as a back-end to annotations produced by any typing algorithm, for instance algorithm W. Thus it should be relatively easy to add M to any front end type inference algorithm. This has been demonstated in the smaller scale, as M has been implemented and composed with an implementation of W for a simple, but representative part of core ML. Implementing M was a trivial programming exercise.

The claim that cut-type variables represent dead code is formally justified by the formulas as types correspondence, where cut-type variables represent cutformulas containing redundant information. In practice programs admitting cut-type variables simply do not occur. The claim is illustrated by a large programming project in SML under current development at Stanford.

The simple examples of M's applications given so far illustrate the principal ideas behind the algorithm, but may not appeal much to applications. In all these simplified examples, elementary compile-time partial evaluation would provide a more efficient analysis. A more realistic example, still somewhat simplified, is the often encountered figuration:

```
fun foobar xs =
    let
        fun len [] n = n
          | len (_::ys) n = len ys (n+1)
    in
        map (len xs) xs
    end
```

By W `len` is typed $\forall\alpha.\alpha\ list \to int \to int$, instead of $int\ list \to int \to int$.

Another application indicated in the introduction is to resolve flex records. This aims more at enhancing the type inference, rather than optimizing the compilation. Examples like

```
fun flex (mix :{a:int,b:real,c:bool} list) =
    let
        fun take r = (#a r,#c r)
    in
        map take mix
    end
```

cannot be compiled by the SML New Jersey compiler version 0.93. An error "unresolved flex record in let pattern" is reported. To make it compile, the programmer has to provide the type annotation

```
take (r:{a:int,b:real,c:bool}) = (#a r,#c r)
```

in the definition of `take`. With minimal typing derivations, the compiler can resolve the record type, since `take` is only used for the input mix. This problem was recently noted by several developers involved in a programming project using SML at Stanford.

Acknowledgements

The author would like to thank Mads Tofte for his consistent encouragement, Fritz Henglein for posing the problem, Xavier Leroy for numerous comments and suggestions on this paper and Tomás E. Uribe, Hugh McGuire and Eddie Chang for their proof-reading.

References

[1] Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[2] Nikolaj S. Bjørner. Minimal typing derivations. DIKU, 1993. Student Project. Available by ftp from `theory.stanford.edu.` in `/pub/nikolaj/mtd.dvi.Z`

[3] Luis Manuel Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[4] Luis Manuel Martins Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[5] Xavier Leroy. Unboxed objects for polymorphic typing. In *POPL*, 1992.

[6] John C. Mitchell and Robert Harper. The Essence of ML. In *POPL*, 1988.

[7] G. Plotkin. A note on inductive generalization. *Mach. Intell.*, 5, 1970.

[8] J. C. Reynolds. Transformation systems and the algebraic structure of atomic formulas. *Mach. Intell.*, 5, 1970.

[9] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Lisp and Functional Programming*, 1994. To appear.