# Proof of the Principal Type Property for System O

Technical Report

Martin Wehr* Martin Odersky*

June 10, 1996

**Abstract**

We study a minimal extension of the Hindley/Milner system that supports overloading and polymorphic records. We also show that every typable term in this system has a principal type and give an algorithm to reconstruct that type. We give the proofs for termination, soundness and correctness for the constrained unification and the type reconstruction algorithm.

Here you can find the full proof for termination, soundness and correctness of the type reconstruction algorithm of system O. If you are interested in a dynamic semantics and a denotational semantics for system O and the connection to polymorphic records take a look in [OWW95].

This paper contains the definition of system O, the definition of its type reconstruction algorithm, all notions to define the properties of type reconstruction and the proofs of all properties. The proofs are inspired by the techniques used in [Che94] and [Jon92].

The first section defines the language and the type system which is an extension of the Hindley/Milner system [Mil78]. Furthermore an example gives a motivation for the use of system O.

In section 2 the type reconstruction algorithm is defined. The algorithm is an extension of Milners algorithm $\mathcal{W}$ where unification must satisfy constraints on type variables. The presentation of the unification algorithm is slightly changed compared to the original, to simplify the proofs for the function *unify*. The new *unify* has the property that the computation introduces never a free type variable.

Section 3 presents the proofs for termination, soundness and correctness of the constrained unification. The idea is to define the term *typing state* which represents the state of a unification computation. Then an ordering on typing states is defined so that minimal typing states result in trivial computations. Therefore every proof is done per induction on the typing state ordering. So it remains to show that every computation step is decreasing in the typing state ordering.

The last section is the proof of the main result, namely soundness and correctness of type reconstruction. The properties are known from the Hindley-Milner system. To state these properties the notions of substitutions and more general type schemes are needed. This is the contribution of this paper, we identify and motivate the new notions and extend proof techniques in that environment. In Section 4 we first refine these notions in a system with constrained type variables.

The idea of the proof is to divide it into two steps. First we prove the soundness and completeness of a system where every derivation is determined by the expression which should be typed. Second we prove the soundness and completeness of the inference algorithm relative to the deterministic system. The properties between system O and the algorithm are then simple corollaries.

*Institut für Programmstrukturen, Universität Karlsruhe, 76128 Karlsruhe, Germany; e-mail:odersky,wehr@ira.uka.de

| | | | | |
|---|---|---|---|---|
| Unique variables | $u$ | $\in$ | $\mathcal{U}$ | |
| Overloaded variables | $o$ | $\in$ | $\mathcal{O}$ | |
| Constructors | $k$ | $\in$ | $\mathcal{K} = \bigcup\{\mathcal{K}_D \mid D \in \mathcal{D}\}$ | |
| Variables | $x$ | $=$ | $u \mid o \mid k$ | |
| Terms | $e$ | $=$ | $x \mid \lambda u.e \mid e\,e' \mid \mathsf{let}\ u = e\ \mathsf{in}\ e'$ | |
| Programs | $p$ | $=$ | $e \mid \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$ | |
| | | | | |
| Type variables | $\alpha$ | $\in$ | $\mathcal{A}$ | |
| Data type constructors | $\mathcal{D}$ | $\in$ | $\mathcal{D}$ | |
| Type constructors | $T$ | $\in$ | $\mathcal{T} = D \cup \{\to\}$ | |
| Types | $\tau$ | $=$ | $\alpha \mid \tau \to \tau' \mid D\ \tau_1\ ...\ \tau_n$ | where $n = \text{arity}(D)$ |
| Type schemes | $\sigma$ | $=$ | $\tau \mid \forall\alpha.\pi_\alpha \Rightarrow \sigma$ | |
| Constraints on $\alpha$ | $\pi_\alpha$ | $=$ | $o_1 : \alpha \to \tau_1, \ldots, o_n : \alpha \to \tau_n$ | ($n \geq 0$, with $o_1, \ldots, o_n$ distinct) |
| Typotheses | , | $=$ | $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ | ($n \geq 0$) |

Figure 1: Abstract syntax of System O.

# 1 Type System

In this section we define System O, a simple functional language with overloaded identifiers. Figure 1 gives the syntax of terms and types. We split the variable alphabet into subalphabets $\mathcal{U}$ for *unique* variables, ranged over by $u$, $\mathcal{O}$ for *overloaded* variables, ranged over by $o$, and $\mathcal{K}$ for data constructors, ranged over by $k$. The letter $x$ ranges over both unique and overloaded variables as well as constructors. We assume that every non-overloaded variable $u$ is bound at most once in a program.

The syntax of *terms* is identical to the language *Exp* in [Mil78]. A *program* consists of a sequence of instance declarations and a term. An *instance declaration* ($\mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p$) overloads the meaning of the identifier $o$ with the function given by $e$ on all arguments that are constructed from the type constructor $T$.

A *type* $\tau$ is a type variable, a function type, or a data type. Data types are constructed from *data type constructors* $D$. For simplicity, we assume that all value constructors and selectors of a data type $D\ \tau_1\ ...\ \tau_n$ are predefined, with bindings in some fixed initial typothesis $,_0$. With user-defined type declarations, we would simply collect in $,_0$ all selectors and constructors actually declared in a given program. Let $\mathcal{K}_D$ be the set of all value constructors that yield a value in $D\,\tau_1, ...., \tau_n$ for some types $\tau_1, ...., \tau_n$. We assume that there exists a *bottom* data type $\perp\!\!\!\perp \in D$ with $\mathcal{K}_{\perp\!\!\!\perp} = \emptyset$. Note that this type is present in Miranda, where it is written (), but is absent in Haskell, where () has a value constructor. We let $T$ range over data type constructors as well as the function type constructor $(\to)$, writing $(\to)\,\tau\,\tau'$ as a synonym for $\tau \to \tau'$.

A *type scheme* $\sigma$ consists of a type $\tau$ and quantifiers for some of the type variables in $\tau$. Unlike with Hindley/Milner polymorphism, a quantified variable $\alpha$ comes with a *constraint* $\pi_\alpha$, which is a (possibly empty) set of bindings $o : \alpha \to \tau$. An overloaded variable $o$ can appear at most once in a constraint. Constraints restrict the instance types of a type scheme by requiring that overloaded identifiers are defined at given types. The Hindley/Milner type scheme $\forall\alpha.\sigma$ is regarded as syntactic sugar for $\forall\alpha.() \Rightarrow \sigma$.

Figure 2 defines the typing rules of System O. The type system is identical to the original Hindley/Milner system, as presented in in [DM82], except for two modifications.

- In rule ($\forall$I), the constraint $\pi_\alpha$ on the introduced bound variable $\alpha$ is traded between typothesis and type scheme. Rule ($\forall$E) has as a premise an instantiation of the eliminated constraint. Constraints are derived using rule (SET). Note that this makes rules ($\forall$I) and ($\forall$E) symmetric to rules ($\to$I) and ($\to$E).

$$\text{(TAUT)} \qquad , \vdash x : \sigma \; (x : \sigma \in , ) \qquad\qquad \frac{, \vdash x_1 : \sigma_1 \quad \dots \quad , \vdash x_n : \sigma_n}{, \vdash x_1 : \sigma_1, \dots, x_n : \sigma_n} \qquad \text{(SET)}$$

$$\text{(}\forall\text{I)} \qquad \frac{, , \pi_\alpha \vdash e : \sigma \qquad (\alpha \notin \mathrm{tv}(, ))}{, \vdash e : \forall\alpha.\pi_\alpha \Rightarrow \sigma} \qquad\qquad \frac{, \vdash e : \forall\alpha.\pi_\alpha \Rightarrow \sigma \qquad , \vdash [\tau/\alpha]\pi_\alpha}{, \vdash e : [\tau/\alpha]\sigma} \qquad \text{(}\forall\text{E)}$$

$$\text{(}{\to}\text{I)} \qquad \frac{, , u : \tau \vdash e : \tau'}{, \vdash \lambda u.e : \tau \to \tau'} \qquad\qquad \frac{, \vdash e : \tau' \to \tau \qquad , \vdash e' : \tau'}{, \vdash e\,e' : \tau} \qquad \text{(}{\to}\text{E)}$$

$$\text{(LET)} \qquad \frac{, \vdash e : \sigma \qquad , , u : \sigma \vdash e' : \tau}{, \vdash \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau} \qquad\qquad \frac{\begin{array}{c}(o : \sigma_{T'} \in , \ \Rightarrow\ T \neq T') \\ , \vdash e : \sigma_T \qquad , , o : \sigma_T \vdash p : \tau\end{array}}{, \vdash \mathsf{inst}\ o : \sigma_T = e\ \mathsf{in}\ p : \tau} \qquad \text{(INST)}$$

Figure 2: Typing rules for System O.

- There is an additional rule (INST) for instance declarations. The rule is similar to (LET), except that the overloaded variable $o$ has an explicit type scheme $\sigma_T$ and it is required that the type constructor $T$ is different in each instantiation of a variable $o$.

We let $\sigma_T$ range over closed type schemes that have $T$ as outermost argument type constructor:

$$\begin{array}{rcll} \sigma_T & = & T\,\alpha_1\,...\,\alpha_n \to \tau & (\mathrm{tv}(\tau) \subseteq \{\alpha_1, \dots, \alpha_n\}) \\ & \mid & \forall\alpha.\pi_\alpha \Rightarrow \sigma'_T & (\mathrm{tv}(\pi_\alpha) \subseteq \mathrm{tv}(\sigma'_T)). \end{array}$$

The explicit declaration of $\sigma_T$ in rule (INST) is necessary to ensure that principal types always exist. Without it, one might declare an instance declaration such as

$$\mathsf{inst}\ o = \lambda x.x\ \mathsf{in}\ p$$

where the type constructor on which $o$ is overloaded cannot be determined uniquely.

The syntactic restrictions on type schemes $\sigma_T$ enforce three properties: First, overloaded instances must work uniformly for all arguments of a given type constructor. Second the argument type must determine the result type uniquely. Finally, all constraints must apply to component types of the argument. The restrictions are necessary to ensure termination of the type reconstruction algorithm. An example is given in Section 2.

The syntactic restrictions on type schemes $\sigma_T$ also explain why the overloaded variables of a constraint $\pi_\alpha$ must be pairwise different. A monomorphic argument to an overloaded function completely determines the instance type of that function. Hence, for any argument type $\tau$ and overloaded variable $o$, there can be only one instance type of $o$ on arguments of type $\tau$. By embodying this rule in the form of type variable constraints we enforce it at the earliest possible time.

**Example 1.1** The following program fragment gives instance declarations for the equality function (==). We adapt our notation to Haskell's conventions, writing `::` instead of : in a typing; writing `(o::a->t1)=>t2` instead of $\forall\alpha.(o : a \to \tau_1) \Rightarrow \tau_2$; and writing `inst o :: s\; o = e` instead of $\mathsf{inst}\ o : \sigma = e$.

```
inst (==) :: Int -> Int -> Bool
     (==) = primEqInt

listEq :: ((==)::a->a->Bool) => [a]->[a]->Bool
```

```
$unify\ :\ (\tau,\tau) \rightarrow (,\ ,S) \rightarrow (,\ ,S)$
$unify\,(\tau_1,\tau_2)\,(,\ ,S)\ =$ case $(S\tau_1, S\tau_2)$ of
      $(\alpha,\alpha) \Rightarrow$
         $(,\ ,S)$
      $(T\,\overline{\tau}_1, T\,\overline{\tau}_2) \Rightarrow$
         $foldr\ unify\ (,\ ,S)\ (zip\ (\overline{\tau}_1,\overline{\tau}_2))$
      $(\alpha,\tau),(\tau,\alpha)$ where $\alpha \notin \mathrm{tv}(\tau) \Rightarrow$
         $foldr\ mkinst\ (,\ \backslash,\ _{\alpha}, [\tau/\alpha]\circ S)\ ,\ _{\alpha}$


$mkinst\ :\ (o : \alpha \rightarrow \tau) \rightarrow (,\ ,S) \rightarrow (,\ ,S)$
$mkinst\,(o : \alpha \rightarrow \tau)\,(,\ ,S)\ =$ case $S\alpha$ of
      $\beta \Rightarrow$
         if $\exists o : \beta \rightarrow \tau' \in\ ,$
         then $unify\ (\tau,\tau')\ (,\ ,S)$
         else $(,\ \cup \{o : \beta \rightarrow [\beta/\alpha]\tau\}, S)$
      $T\ \overline{\tau} \Rightarrow$
         if $!\exists o : \sigma_T \in\ ,$
         then let $(p, T\alpha_{p(1)} \ldots \alpha_{p(n)} \rightarrow \tau', C) = struct(\sigma_T, ,\ ,S)$
            in $uninfy\ (\tau,\tau')(doinst(1,p,C,\alpha)(,\ ,S))$


$doinst\ :\ (\mathbf{N}, \mathbf{\Pi_N}, \mathbf{C}, \alpha) \rightarrow (,\ ,S) \rightarrow (,\ ,S)$
$doinst\,(i,p,C,\alpha)(,\ ,S)\ =$ case $C$ of
      $C'.\pi_\beta \Rightarrow$
            $doinst\,(i+1,p,C',\alpha)\,(fold\,mkinst\,(,\ ,[S\alpha \downarrow p(i)/\beta]\circ S)\pi_\beta)$
         $\emptyset \Rightarrow (,\ ,S)$
```

Figure 3: Algorithm for constrained unification

```
listEq [] []          =  True
listEq (x:xs) (y:ys)  =  x == y && listEq xs ys

inst (==) :: ((==):: a->a->Bool) => [a]->[a]->Bool
     (==) =  listEq
```

Note that using (==) directly in the second instance declaration would not work, since instance declarations are not recursive. An extension of System O to recursive instance declaration would be worthwhile but is omitted here for simplicity.

## 2 Type Reconstruction

Figures 3 and 4 present type reconstruction and unification algorithm for System O. Compared to Milner's algorithm $\mathcal{W}$ [Mil78] there are two extensions.

- The case of binding a type variable in the unification algorithm is extended. To bind a type variable $\alpha$ to a type $\tau$ the constraints of $,\ _\alpha$ have to be satisfied. The function *mkinst* ensures that type $\tau$ statisfies the constraints $,\ _\alpha$.
- The function *tp* is extended with a branch for instance declarations inst $o : \sigma_T = e$ in $p$. In this case it must be checked that the inferred type $\sigma'_T$ for the overloading term $e$ is less

```
$tp$                 :  $(p, , , S) \rightarrow (\tau, , , S)$

$tp\ (u, , , S)$     =  if $u : \sigma \in ,$
                        then $newinst\ (\sigma, , , S)$

$tp\ (o, , , S)$     =  $newinst\ (\forall\beta\forall\alpha.(o : \alpha \rightarrow \beta) \Rightarrow \alpha \rightarrow \beta, , , S)$

$tp\ (\lambda u.e, , , S)$
                     =  let   $\alpha$ a new type variable
                              $(\tau, , _1, S_1) = tp\ (e, , \cup \{u : \alpha\}, S)$
                        in    $(\alpha \rightarrow \tau, , _1, S_1)$

$tp\ (e\,e', , , S)$
                     =  let   $(\tau_1, , _1, S_1) = tp\ (e, , , S)$
                              $(\tau_2, , _2, S_2) = tp\ (e', , _1, S_1)$
                              $\alpha$ a new type variable
                              $(, _3, S_3) = unify\ (\tau_1, \tau_2 \rightarrow \alpha)\ (, _2, S_2)$
                        in    $(\alpha, , _3, S_3)$

$tp$ (let $u = e$ in $e', , , S)$
                     =  let   $(\sigma, , _1, S_1) = gen\ (tp\ (e, , , S))$
                        in    $tp\ (e', , _1 \cup \{u : \sigma\}, S_1)$

$tp$ (inst $o : \sigma_T = e$ in $p, , , S)$
                     =  let   $(\sigma'_T, , _1, S_1) = gen\ (tp\ (e, , , S))$
                              $(\tau_2, , _2, S_2)\ =\ skolemize\ (\sigma_T, , _1, S_1)$
                              $(\tau_3, , _3, S_3)\ =\ newinst\ (\sigma'_T, , _2, S_2)$
                        in    if $\forall o{:}\sigma_{T'} \in , .\ T \neq T' \wedge$
                                $unify(\tau_2, \tau_3)(, _3, S_3)$ defined then
                                    $tp\ (p, , _1 \cup \{o : \sigma_T\}, S_1)$
```

Figure 4: Type reconstruction algorithm for System O

general then the given type $\sigma_T$.

We use the following abbreviations:

$$\begin{array}{rcl} ,_{\alpha} & = & \{o : \alpha \rightarrow \tau \mid o : \alpha \rightarrow \tau \in , \} \\ ,_{A} & = & \cup_{\alpha \in A}\ ,_{\alpha} \end{array}$$

where $A$ is a set of type variables.

The termination of $unify$ and $mkinst$ critically depends on the form of overloaded type schemes $\sigma_T$:

$$\begin{array}{rcll} \sigma_T & = & T\ \alpha_1\ ...\ \alpha_n \rightarrow \tau & (\mathrm{tv}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_n\}) \\ & \mid & \forall\alpha.\pi_\alpha \Rightarrow \sigma'_T & (\mathrm{tv}(\pi_\alpha) \subseteq \mathrm{fv}(\sigma'_T)). \end{array}$$

We show with an example why $\sigma_T$ needs to be parametric in the arguments of $T$. Consider the following program, where $k \in \mathcal{K}_T$.

$$\begin{array}{rcl} p & = & \mathsf{let}\ \ (;)\,x\,y = y\mathsf{in} \\ & & \mathsf{inst}o : \forall\alpha.o : \alpha \rightarrow \alpha \Rightarrow T(T\alpha) \rightarrow \alpha \\ & & \quad = \lambda k(k\,x).o\,x \\ & & \mathsf{in}\ \ \lambda x.\lambda y.\lambda f.\ o\,x\,;o\,y\,;f\,(k\,y)\,;f x \end{array}$$

```
$newinst$            :  $(\sigma, , , S) \rightarrow (\tau, , , S)$
$newinst\ (\forall\alpha.\pi_\alpha \Rightarrow \sigma, , , S)$
                     =  let  $\beta$ a new type variable
                        in   $newinst$
                             $([\beta/\alpha]\sigma, , \cup [\beta/\alpha]\pi_\alpha, S)$
$newinst\ (\tau, , , S)$
                     =  $(\tau, , , S)$


$skolemize$          :  $(\sigma, , , S) \rightarrow (\tau, , , S)$
$skolemize\ (\forall\alpha.\pi_\alpha \Rightarrow \sigma, , , S)$
                     =  let  $T$ a new 0-ary type constructor
                        in   $skolemize$
                             $([T/\alpha]\sigma, , \cup [T/\alpha]\pi_\alpha, S)$
$skolemize\ (\tau, , , S)$
                     =  $(\tau, , , S)$

$gen$                :  $(\tau, , , S) \rightarrow (\sigma, , , S)$
$gen\ (\sigma, , , S)$    =  if $\exists\alpha.\alpha \in \mathrm{tv}(S\sigma) \setminus \mathrm{tv}(S(, \setminus, _\alpha))$
                        then $gen\ (\forall\alpha., _\alpha \Rightarrow \sigma, , \setminus, _\alpha, S)$
                        else $(\sigma, , , S)$
```

Figure 5: Functions needed to define the type reconstruction algorithm

Then computation of $tp(p, \emptyset, id)$ leads to a call $tp(f\,x, , , S)$ with $x : \alpha, y : \beta, f : T\beta \rightarrow \delta \in , .$ This leads in turn to a call $unify(\alpha, T\beta)(, , S)$ where the following assumptions hold:

- $\sigma_T \ = \ \forall\alpha.o : \alpha \rightarrow \alpha \Rightarrow T(T\alpha) \rightarrow \alpha$
- $, \ \supseteq \ \{o : \alpha \rightarrow \alpha, o : \beta \rightarrow \beta, o : \sigma_T\}$,
- $S$ is a substitution with $\alpha, \beta \notin dom(S)$.

Unfolding *unify* gives $mkinst(o : \alpha \rightarrow \alpha)(, \setminus, _\alpha, S')$ where $S' = [T\beta/\alpha] \circ S$, which leads in turn to the following two calls:

1. $newinst(\sigma_T, , \setminus, _\alpha, S') \ = \ (T(T\gamma) \rightarrow \gamma, , ', S')$
   where $, ' \ \supseteq \ \{o : \beta \rightarrow \beta, o : \gamma \rightarrow \gamma, o : \sigma_T\}$ and $\gamma$ is a fresh type variable, and
2. $unify(\alpha \rightarrow \alpha, T(T\gamma) \rightarrow \gamma)(, ', S')$.

Since $S'\alpha = T\beta$, unfolding of 2. results in an attempt to unify $T\beta$ and $T(T\gamma)$, which leads to the call $unify(\beta, T\gamma)(, ', S')$. This is equivalent to the original call $unify(\alpha, T\beta)(, , S)$ modulo renaming of $\alpha, \beta$ to $\beta, \gamma$. Hence, $unify$ would loop in this situation.

The need for the other restrictions on $\sigma_T$ are shown by similar constructions. It remains to be seen whether a more general system is feasible that lifts these restrictions, *e.g.* by extending unification to regular trees [Kae92].

# 3 Proofs for the Unification with restricted Type variables

A type variable substitution is an idempotent mapping from type variables to types that maps all but a finite number of type variables to themselves. A substitution can be extended homomorphical on types, type schemes, typotheses and judgments. Note that applying a substitution on a O derivation delivers a correct new substituted O derivation.

We assume all bound type variables are different to free type variables. This can always be achieved by renaming the bound type variables.

Let $id$ be the identity mapping and $[\tau/\alpha]$ the replacement of $\alpha$ by $\tau$. Juxtaposition $RS$ of substitutions $R$ and $S$ denote the composition of mappings. We define $S \leq^T R$ iff $TS = R$ and as short form $S \leq R$ iff $\exists T.S \leq^T R$. In [LMM87] it is stated that the set of substitutions with the relation $\leq$ is a complete lower semi-lattice.

Give two types $\tau_1, \tau_2$ a unifier is a substitution $S$ with $S\tau_1 = S\tau_2$. A most general unifier $S$ has property $S \leq S'$ for every other unifier $S'$. We denote this as $mgu(\tau_1, \tau_2) = S$.

**Definition.** A *configuration* $(, , S)$ is a pair consisting of a typotheses , and a substitution $S$ such that, for all $\alpha \in \mathrm{dom}(S)$, $,_{\alpha} = \emptyset$.

**Definition.** (*typing state*, *unify* and *mkinst* on typing states, measure on typing states)
Let $\mathcal{T}$ be the *typing state* $(\tau_1, \tau_2, , , S, r)$ over two types $\tau_1, \tau_2$ a configuration $(, , S)$ and a variable restriction $r \;=\; o : \alpha \rightarrow \tau$.

we define $\searrow$ to insert a new configuration in a typing state

$$(, ', S') \searrow (\tau_1, \tau_2, , , S, r) \;:=\; (\tau_1, \tau_2, , ', S', r)$$

we define the following :

$$\begin{array}{lcl} dom(S) & := & \{\alpha | S\alpha \neq \alpha\} \\ Var(\mathcal{T}) & := & tv(S(\tau_1, \tau_2, \alpha \rightarrow \tau, , )) \end{array}$$

Note $\alpha \in dom(S)$ implies $\alpha \notin Var(\mathcal{T})$, so every binding decreases the amount of free variables.

We define a typing function as a partial map $f$ from a typing state $\mathcal{T}_1$ to a typing state $\mathcal{T}_2$ (in short: $f\mathcal{T}_1 \rightsquigarrow \mathcal{T}_2$) iff

$$f\ \mathcal{T}_1 = \begin{cases} & \text{stop with an error} \\ \mathcal{T}_2 & \text{terminate with result } \mathcal{T}_2 \end{cases}$$

Now we can define $unify$ and $mkinst$ as functions on typing states .
if $unify(\tau_1, \tau_2)(, , S)$ terminates with result $(, ', S')$ or stops with error then

$$unify\ \mathcal{T} \rightsquigarrow (, ', S') \searrow \mathcal{T}$$

if $mkinst(o : \alpha \rightarrow \tau)(, , S)$ terminates with result $(, ', S')$ or stops with error then

$$mkinst\ \mathcal{T} \rightsquigarrow (, ', S') \searrow \mathcal{T}$$

We define a recursive measure $|\tau|$ on types $\tau$

$$\begin{array}{lcl} \text{case } \tau = \alpha & : & |\tau| := 1 \\ \text{case } \tau = T\tau_1 \ldots \tau_n & : & |\tau| := 1 + \sum_{i=1}^{n} |\tau_i| \\ \text{case } \tau = \tau_1 \rightarrow \tau_2 & : & |\tau| := 1 + |\tau_1| + |\tau_2| \end{array}$$

We define a measure on typing state and use it to prove inductive properties of *unify* and *mkinst*.

$$|\mathcal{T}| \;:=\; (|Var(\mathcal{T})|, |S\alpha|, |S\tau_1|, |S\tau_2|)$$

where we assume a lexicographical ordering on integer typing states , and $\alpha$ is the restricted variable in the restriction $r \;=\; o : \alpha \rightarrow \tau$ .

As mentioned in Section 2 type schemes $\sigma_T$ for a type constructor $T$ are defined by

$$\begin{array}{rcll} \sigma_T & = & T\ \alpha_1\ ...\ \alpha_n \to \tau & (\mathrm{tv}(\tau) \subseteq \{\alpha_1,\ \ldots,\ \alpha_n\}) \\ & \mid & \forall\alpha.\pi_\alpha \Rightarrow \sigma'_T & (\mathrm{tv}(\pi_\alpha) \setminus \{\alpha\} \subseteq \mathrm{fv}(\sigma'_T)). \\ \pi_\alpha & = & o_1 : \alpha \to \tau_1,\ \ldots,\ o_n : \alpha \to \tau_n & (n \geq 0, \text{ with } o_1,\ \ldots,\ o_n \text{ distinct}) \end{array}$$

This gives the following Lemma.

**Lemma 3.1** (structure of $\sigma_T$) For every type scheme $\sigma_T$ let $n = ar(T)$ be the arity of type constructor $T$. Then there is a permutation $p : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $\sigma_T$ has the following structure :

$$\sigma_T = \forall\alpha_1.\pi_{\alpha_1} \Rightarrow \ldots \forall\alpha_n.\pi_{\alpha_n} \Rightarrow T\alpha_{p(1)} \ldots \alpha_{p(n)} \to \tau$$

where $tv(\pi_{\alpha_i}) \subseteq \{\alpha_1, \ldots, \alpha_i\}$ for $i \in \{1, \ldots, n\}$.

**Definition.** (Structure of a type scheme) Let $(, , S)$ be a configuration. Define

$$struc(\sigma_T, , , S) \mapsto (p, T\alpha_{p(1)} \ldots \alpha_{(n)} \to \tau, C)$$

where $C = \pi_{\alpha_1} \ldots \pi_{\alpha_n}$ is the sequence of the renamed variable restrictions such that w.l.o.g. the bound variables in $\sigma_T$ and the free variables in $(, , S)$ differ.

Let further $T\tau_1 \ldots \tau_n \downarrow k \mapsto \tau_k$ for $k \in \{1,\ \ldots, n\}$.

**Definition.** ( *doinst* as typing function) Let $(\tau_1, \tau_2, , , S, o : \alpha \to \tau)$ be a typing state with :

- $S\alpha = T\overline{\tau}$
- $!\exists o : \sigma_T \in ,$
- $\sigma_T = \forall\alpha_1.\pi_{\alpha_1} \Rightarrow \ldots \forall\alpha_n.\pi_{\alpha_n} \Rightarrow T\alpha_{p(1)} \ldots \alpha_{p(n)} \to \tau$
- $\{\alpha_1, \ldots, \alpha_n\} \cap Var(\mathcal{T}) = \emptyset$

then $doinst; \mathcal{T} \rightsquigarrow (, ', S') \searrow \mathcal{T}$ is a short form for $doinst\,(1, p, \pi_{\alpha_1} \ldots \pi_{\alpha_n}, \alpha)(, , S) \rightsquigarrow (, ', S')$

## 3.1 Termination of *unify*

Figure 3 is a slight variant of the original proposal for the *mkinst* algorithm. Observe there is never a introduction of free type variable in the configuration. The only place where new type variable are introduced is in the branch of *doinst* but here the type variable is bound.

Because the definition of *unify*, *mkinst* and *doinst* is mutual recursive the termination of them is proved in one step. By induction on $|\mathcal{T}|$ it is shown that this measure decreases. Because every decreasing sequence of typing states is finitary this implies the termination of the computation.

The cases considered in this proof are the same as in the proof of soundness and completeness of *unify*.

**Lemma 3.2** For a typing state $\mathcal{T}$ the computation $unify\,\mathcal{T}$ terminates with result $(, ', S')$ or stops with error.

(in short: $\forall\mathcal{T}\exists(, ', S').\ unify\,\mathcal{T} \rightsquigarrow (, ', S') \searrow \mathcal{T}$)

**Proof:** Consider the cases for the values of $(\tau_1, \tau_2, , , S, r)$ in the *unify* algorithm :

$\boldsymbol{S\tau_1 = \alpha, S\tau_2 = \beta}$ Termination is trivial.

$\boldsymbol{S\tau_1 = T\overline{\tau_1}, S\tau_2 = T\overline{\tau_2}}$ Here a sequence of *unify* calls is generated. The new arguments $\tau_1, \tau_2$ are subtrees of the given type trees $T\overline{\tau_1}, T\overline{\tau_2}$. So definition of the measure on typing states gives *unify* calls with typing state arguments of lower measure. A straightforward inductive argument on the sequence of types $\overline{\tau_1}, \overline{\tau_2}$ gives termination for that case.

**$S\tau_1 = \alpha, S\tau_1 = \tau$** We have to consider two cases for , $_\alpha$

- **$\Gamma_\alpha = \emptyset$** Termination is trivial there is only the operation of binding $[\tau/\alpha]S$ to do.
- **$\Gamma_\alpha \neq \emptyset$** Now $fold\ mkinst$ is called with the new typing state $\mathcal{T}' = (\tau_1, \tau_2, , ', S', r)$ where $S' = [\tau/\alpha]S$ and , $'$ = , $\setminus$ , $_\alpha$. Because the binding $[\tau/\alpha]$ we have $\alpha \notin Var(\mathcal{T}')$ for the new typing state $\mathcal{T}'$. So the first component of the measure $|\mathcal{T}'|$ is decreased. Therefore in the following we can use the induction hypotheses that $unify$ terminates (1) .

  Next we have to consider the cases of $\tau$ and $r' = o : \alpha \to \tau' \in$ , $_\alpha$ as argument of $mkinst$ :

  - **$\tau = \beta$** There are two cases to consider look at the corresponding branches in the *mkinst* code :
    - **$\nexists o : \beta \to \tau'' \in S\Gamma$** Termination is trivial.
    - **$\exists o : \beta \to \tau'' \in S\Gamma$** Now we use (1) to get termination of the $unify(\tau', \tau'')(, ', S')$ call.
  - **$\tau = T\overline{\tau}$** We have to prove the termination of *doinst*. Observe that the structure of $\sigma_T$ is finite. So *doinst* calls *mkinst* finitely often. The nontrivial case in the code of *doinst* is the branch for $C'.\pi_\beta$. In this case $fold\ mkinst(, ', S'')\pi_\beta$ is called where $S'' = [(S'\alpha \downarrow p(i))/\beta]S'$. We have $S''\beta = \tau \downarrow p(i)$ that is a part of the type tree $S'\alpha = \tau$ we had before. So the second component of the typing state measure is decreased in the raised *mkinst* calls and we can use the induction hypotheses to get the termination.

## 3.2 Properties of *unify*

A configuration $(, , S)$ is more general than a configuration $(, ', S')$ if $S \leq S'$ and $S', ' \vdash S', $ .

**Definition.** A *unifying problem* is given by the tuples $(\tau_1, \tau_2)(, , S)$ where $\tau_1, \tau_2$ are types and $(, , S)$ is a configuration. Then $(, ', S')$ is a solution for the unifying problem iff $S'\tau_1 = S'\tau_2$ and $(, ', S') \leq (, , S)$. A solution $(, ', S')$ which gives $(, '', S'') \leq (, ', S')$ for every other solution $(, '', S'')$ of the same problem is called *most general unifying configuration* (short: mgu configuration).

**Definition.** Let $(, , S)$ be a configuration and $r = o : \alpha \to \tau$ be a restriction with $\alpha \in dom(S)$ then $(r)(, , S)$ is called a *restriction problem*. A configuration $(, ', S')$ with $S', ' \vdash o : S'(\alpha \to \tau)$ and $(, ', S') \leq (, , S)$ is called a solution of the restriction problem.

**Definition.** Let $\mathcal{T} = (\tau_1, \tau_2, , , S, o : \alpha \to \tau)$ be a typing state then $up = (\tau_1, \tau_2)(, , S)$ is the unifying problem and $(o : \alpha \to \tau)(, , S)$ is the restriction problem to the given typing state .

All configurations $(, , S)$ used in the computation of the type inference have the following properties.

**Lemma 3.3** (structure of configurations) There are three kinds of elements $x : \sigma \in S,$

**(unique)** $x$ is a unique variable $u \in \mathcal{U}$.

**(overload)** $x = o \in \mathcal{O}$ and $\sigma = \sigma_T$ with $fv(\sigma_T) = \emptyset$. Further for the type constructor $T$ there is no more $o : \sigma'_T \in S,$ .

**(restriction)** $x = o \in \mathcal{O}$ and $S\sigma = \alpha \to \tau$. This case is called variable restriction (short: restriction named $r$). There is no more $o : \alpha \to \tau' \in S,$ with $\tau \neq \tau'$.

So in the following we call a typotheses , which can be partitioned like that a *structured* typotheses in short , $= A.D.C.$ Where the set $A$ is the typotheses for unique names and has elements of form $u : \sigma$. $D$ is the set of declarations, that is of non conflicting instance declarations $o : \sigma_T$ for overloaded variables. $C$ is the set of non conflicting variable constraints of form $o : \alpha \to \tau$. We use letter $T$ to denote the union $A.D$ of the unique or overloaded variable declarations and have partition $,\ = T.C$

**Proof:** Simple verification by looking on all computation steps of the algorithms.

The case **(overload)** is a conclusion when we observe overloaded variables are introduced only in the inst case in function *tp*. In this case *gen* is always applied so that all type variables in $\sigma_T$ are quantified.

For the case **(restriction)** we see the only place to introduce restrictions is the $\beta$ branch in function *mkinst*. In this branch the restriction is included only if the if statement checks the uniqness of the restriction.

**Lemma 3.4** (characterization of satisfied restriction) Given a configuration $(,\ ,S)$ and a variable restriction $r = o : \alpha \to \tau$ with $\alpha \in dom(S)$ and judgment $S,\ \vdash\ o : S(\alpha \to \tau)$ then there are two cases for the derivation of the judgment :

$$\begin{array}{lll}
\textbf{(var)} & S\alpha = \beta & \Longrightarrow\ !\exists o : \beta \to S\tau \in S, \\
\textbf{(constructor)} & S\alpha = T\overline{\tau} & \Longrightarrow\ !\exists o : \sigma_T \in S, \\
 & \text{where} & \sigma_T = \forall\alpha_1.\pi_{\alpha_1} \Rightarrow \ldots \forall\alpha_n.\pi_{\alpha_n} \Rightarrow T\alpha_{p(1)} \ldots \alpha_{p(n)} \to \tau' \\
 & \text{and} & \overline{\tau} = \tau_1 \ldots \tau_n \\
 & \text{and} & S \geq \{\tau_{p^{-1}(1)}/\alpha_1, \ldots, \tau_{p^{-1}(n)}/\alpha_n\} \\
 & \text{and} & S,\ \vdash\ S\pi_{\alpha_i}\ \text{for}\ i \in \{1, \ldots, n\} \\
 & \text{and} & S\tau = S\tau'
\end{array}$$

**Proof:** Simple induction on the derivation of the judgment $S,\ \vdash\ o : S(\alpha \to \tau)$ where we have to look only at the rules $(TAUT), (\forall I)$ and $(\forall E)$.

For the **(var)** case we can only apply the $(TAUT)$ rule.

For the **(constructor)** case we must apply the $(TAUT)$ rule on the given $\sigma_T \in S,\ .$ This is a consequence of the structure of configuration $(,\ ,S)$ Lemma 3.3. To derive $S,\ \vdash\ Sr$ the rule $(\forall E)$ must be applied for every $\pi_{\alpha_i}$ which gives the technical side conditions.

**Lemma 3.5** (characterization of more general configuration) We have $(,\ ,S) \geq (,\ ', S')$ iff $S \leq^R S'$ and we consider the cases for $x : \sigma \in S,\ $ :

- for non-restriction $u : \sigma \in S,\ \Longrightarrow\ u : R\sigma \in S',\ '.$
- For a restriction $r = o : \alpha \to \tau \in S,\ $ and a type variable set $\mathcal{V} = dom(S') \setminus dom(S)$ there are three cases :

$$\begin{array}{llll}
(\textbf{unchange} \Leftrightarrow\!\textbf{r}) & \alpha \notin \mathcal{V} & \Longrightarrow & Rr \in S',\ ' \\
(\textbf{rename} \Leftrightarrow\!\textbf{r}) & \alpha \in \mathcal{V}\ R\alpha = \beta & \Longrightarrow & Rr \in S',\ ' \\
(\textbf{bind} \Leftrightarrow\!\textbf{r}) & \alpha \in \mathcal{V}\ R\alpha = T\overline{\tau} & \Longrightarrow & Rr \notin S',\ '\ \text{and}\ S',\ '\ \vdash\ S'r
\end{array}$$

**Proof:** We use Lemma 3.3 the characterization of configurations the cases **(unique)** and **(overload)** explain the non restriction cases.

For the restrictions we use Lemma 3.4 so that case **(var)** gives **(rename-r)** and case **(constructor)** gives **(bind-r)**. Case **(unchange-r)** is a conclusion that $S',\ '\ \vdash\ S'r$ can be deduced only with the use of rule $(TAUT)$ so we get $Rr \in S',\ '.$

**Lemma 3.6** The "more general" relation on configurations is transitive.

**Proof:** The more general relation on $S \leq S'$ substitutions is transitive by use of composition on substitutions.

For the relation on configurations we have to show $(,\ ,S) \geq (,\ ', S') \wedge (,\ ', S') \geq (,\ '', S'') \Rightarrow (,\ ,S) \geq (,\ '', S'')$. So we have $S',\ '\ \vdash\ S',\ $ (1) and $S'',\ ''\ \vdash\ S'',\ '$ (2) and $S' \leq^R S''$. So clearly $RS',\ '\ \vdash\ RS',\ \ =\ S'',\ '\ \vdash\ S'',\ $ (3) is a correct derivation by applying $R$ on derivation for (1). Now we substitute every application of $(TAUT)$ for $x : \sigma \in S'',\ '$ in derivation (3) by a derivation of $S'',\ ''\ \vdash\ x : \sigma$ which is given by (2) we get $S'',\ ''\ \vdash\ S'',\ \Box$.

**Lemma 3.7** (soundness and completeness of unification) Let $\mathcal{T} = (\tau_1, \tau_2, , , S, r)$ be a typing state then :

1. **(a)** if $unify\ \mathcal{T} \leadsto (, ', S')$ then $(, ', S')$ is a solution to the unifying problem of $\mathcal{T}$.

   **(b)** if $(, '', S'')$ is a unify solution for *up* then $(, '', S'') \leq (, ', S')$. So $(, ', S')$ is a mgu configuration for *up*.

2. **(a)** if $mkinst(r)(, , S) \leadsto (, ', S')$ then $(, ', S')$ is a solution for the restriction problem.

   **(b)** if $(, '', S'')$ is a solution to the restriction problem then $(, '', S'') \leq (, ', S')$.

3. **(a)** if $S\alpha = T\overline{\tau}$ and $doinst\,\mathcal{T} \leadsto (, ', S')$ then $(, ', S')$ is a solution to the restriction problem $(r)(, , S)$.

   **(b)** if $(, '', S'')$ is a solution to the restriction problem then $(, '', S'') \leq (, ', S')$.

**Proof:**

We use induction on the measure of the typing state $\mathcal{T} = (\tau_1, \tau_2, , , S, r)$. We have to show stepwise that the algorithm computes the needed solution $(, ', S')$ which is the most general solution.

Look first on the cases for $S\tau_1$ and $S\tau_2$ in the function *unify*.

$\boldsymbol{S\tau_1 = \alpha, S\tau_2 = \beta}$ Trivial case we get 1(a) & (b) if we set $(, ', S') \overset{\text{def}}{=} (, , S)$.

$\boldsymbol{S\tau_1 = T\overline{\tau}_1, S\tau_2 = T\overline{\tau}_2}$

**1(a)** For $\overline{\tau}_1 = \tau_1' \ldots \tau_n'$ and $\overline{\tau}_2 = \tilde{\tau}_1 \ldots \tilde{\tau}_n$ we get for $i \in \{1, \ldots, n\}$

$$|S\tau_i'| < |S\tau_1| \text{ and} |S\tilde{\tau}_i| < |S\tau_2| \text{ (1)}$$

We have to prove $unify\,\mathcal{T}_{i-1} \leadsto \mathcal{T}_i$ (2) where 1(a) is valid for $i \in \{1, \ldots, n\}$ where $\mathcal{T}_i = (\tau_i', \tilde{\tau}_1, , {}_{i-1}, S_{i-1}, r)$ and $(, {}_0, S_0) = (, , S)$ and $unify\,\mathcal{T}_i \leadsto (, {}_i, S_i)$. Using Lemma 3.6 the transitivity of $\geq$ on configurations and the definition of solutions for unifying problems gives $S_n\tau_i' = S_n\tilde{\tau}_i$ for $i \in \{1, \ldots, n\}$ so $S_n\tau_1 = S_n\tau_2$. For $(, ', S') \overset{\text{def}}{=} (, {}_n, S_n)$ we have 1 (a).

We prove (2) by using induction hypotheses on *unify*, so we have to show $|\mathcal{T}_i| < |\mathcal{T}|$ (3) .

**i=1** Because $S_0 = S$ (1) gives that $3^{rd}$ and $4^{th}$ of the measure on typing states are reduced. That is (3) .

**i** We have to consider two cases. First if $S_i = S$ then we use the same argument as above and get (3) . If $S_i \neq S$ then $dom(S_i) \supset dom(S)$ and therefore $Var(\mathcal{T}_i) \subset Var(\mathcal{T})$ so the $1^{st}$ component of the measure on typing states decreases □.

**1(b)** For 1(b) we have that $(, '', S'')$ is a solution of the unifying problem so we get $S''\tau_i' = S''\tilde{\tau}_i$ for $i \in \{1, \ldots, n\}$ the argument above allows to use induction hypotheses and we get $(, {}_i, S_i) \geq (, '', S'')$ so $(, {}_n, S_n) \geq (, '', S'')$ so $(, ', S') \geq (, '', S'')$ □.

$\boldsymbol{S\tau_1 = \alpha, S\tau_1 = \tau}$ We take a configuration $(, {}_1, S_1) = (, \setminus , {}_\alpha, [\tau/\alpha] \circ S)$ which gives $S_1\tau_1 = S_1\tau_2$ and $S \leq S_1$. Further we define a new typing state for the following computation $\mathcal{T}' \overset{\text{def}}{=} (, {}_1, S_1) \searrow \mathcal{T}$.

For 1(b) remember $(, '', S'')$ is a solution for the unifying problem so $S''\alpha = S''\tau$ because $S \leq S''$ we get $S_1 \leq S''$ and with $, {}_1 \subseteq ,$ we get $S'', '' \vdash S'', \Rightarrow S'', '' \vdash S'', {}_1$ so we have $(, '', S'') \leq (, {}_1 S_1)$ (4) .

We have to consider the cases for $, {}_\alpha$

$\boldsymbol{\Gamma\alpha = \emptyset}$ So $, {}_1 = ,$ and for $(, ', S') \overset{\text{def}}{=} (, {}_1, S_1)$ we get 1(a) and (4) validates 1(b).

$\boldsymbol{\Gamma_\alpha \neq \emptyset}$ We have $\alpha \notin Var(\mathcal{T}')$ because $\alpha \in dom(S_1) = dom([\tau/\alpha]\circ S)$. So we get $Var(\mathcal{T}') \subset var(\mathcal{T}) \Rightarrow |\mathcal{T}'| < |\mathcal{T}|$ (5) by decreasing the first component of the measure on typing states .

Next we have to consider the cases of type $\tau$ bound to $\alpha$ and the restriction $r' = o : \alpha \rightarrow \tau' \in {,}_\alpha$ as argument of *mkinst*. We have to show that *mkinst* computes a solution to the restriction problem $(r')(, {}_1, S_1)$ that is 2(a) :

$\boldsymbol{\tau = \beta}$ We have to define a solution $(, {}', S')$ for the restriction problem $(r')(, {}_1, S_1)$ and apply the case **(var)** of lemma Lemma 3.4 to show $S', {}' \vdash S'r'$. There are two cases to consider :

$\boldsymbol{\nexists o : \beta \rightarrow \tau'' \in \mathbf{S_1\Gamma_1}}$

**2(a)** Let $, {}' \overset{\text{def}}{=} (, {}_1, o : \beta \rightarrow S_1\tau')$ and $S' \overset{\text{def}}{=} S_1$ then we get

$$o : S'(\alpha \rightarrow \tau') \;=\; S'r' \;\in S', {}' \;\Longrightarrow\; S', {}' \vdash S'r' \text{ that's 2(a)}$$

**2(b)** Because $r' \in ,$ and $S'', {}'' \vdash S''$, further $S' \leq S''$ by using (4) we get $S'', {}'' \vdash S'', {}'$ so $(, {}'', S'') \leq (, {}', S')$.

$\boldsymbol{\exists o : \beta \rightarrow \tau'' \in \mathbf{S_1\Gamma_1}}$

**2(a)** Let $\mathcal{T}'' = (\tau', \tau'', S_1, , {}_1, r')$ for the computation $unify\mathcal{T}'' \rightsquigarrow (, {}', S')$ because of (5) we can use the induction hypotheses for the $unify$ call. So we get $o : S'(\alpha \rightarrow \tau') = o : S'(\beta \rightarrow \tau'') \in S', {}'$ and naturally $S', {}' \vdash S'r'$.

**2(b)** We get $(, {}'', S'')$ a solution for $(r')(, {}_1, S_1)$ so we have by definition $S'', {}'' \vdash S''r'$ and $(, {}'', S'') \leq (, {}_1, S_1)$. Applying lemma Lemma 3.4 with case **(var)** gives $S''\alpha \rightarrow \tau' = S''\beta \rightarrow \tau''$. We also have $(, {}', S')$ is the most general unifying configuration of problem $(\alpha \rightarrow \tau', \beta \rightarrow \tau'')(, {}_1, S_1)$ so 1(b) gives $(, {}'', S'') \leq (, {}', S')$.

$\boldsymbol{\tau = T\overline{\tau}}$

**3(a)** We have to prove *doinst* $\mathcal{T}' \rightsquigarrow (, {}', S')$ with $S', {}' \vdash S'r'$ and $(, {}', S') \leq (, {}_1, S_1)$. This is done by observing that computation of *doinst* ensures the case **(constructor)** of lemma Lemma 3.4. For $\overline{\tau} = \tau'_1 \ldots \tau'_n$ and $o : \sigma_T \in {,}_1$ with $\sigma_T = \forall\alpha_1.\pi_{\alpha_1} \Rightarrow \ldots \forall\alpha_n.\pi_{\alpha_n} \Rightarrow T\alpha_{p(1)} \ldots \alpha_{p(n)} \rightarrow \tau''$. By a look at the code we see that *doinst* $\mathcal{T}'$ raises the following calls.
So we have to prove for $i \in \{1, \ldots, n\}$

$$\begin{array}{l} doinst\,(i, p, C_i, \alpha)(, {}'_{i-1}, S'_{i-1}) \rightsquigarrow (, {}'_i, S'_i) \\ \text{where} \quad fold\, mkinst\,(, {}'_{i-1}, S''_i)\pi_{\alpha_i} \rightsquigarrow (, {}'_i, S'_i) \;\; (6) \\ \quad C_i = \pi_{\alpha_{n-i-1}} \ldots \pi_{\alpha_n} \\ \quad S''_i = [S'_{i-1}\alpha \downarrow p(i)/\alpha_i] \circ S'_{i-1} \;\; (7) \\ \quad (, {}'_0, S'_0) = (, {}_1, S_1) \end{array}$$

We take as assumptions the soundness of the *mkinst* calls (6) so we get that $S'_i, {}'_i \vdash S'_i\pi_{\alpha_1} \ldots \pi_{\alpha_i}$. For $i = n$ we get $S'_n, {}'_n \vdash S'_nC_0$. The condition (7) gives $S'_n \geq \{\tau'_{p^{-1}(1)}/\alpha_1, \ldots, \tau'_{p^{-1}(n)}/\alpha_n\}$. At least to prove $S', {}' \vdash S'(o : \alpha \rightarrow \tau')$ we need further $(, {}', S')$ to be a solution for the unifying problem $(\tau', \tau'')(, {}'_n, S'_n)$.
For $\mathcal{T}'' = (\tau', \tau'', , {}'_n, S'_n, r')$ we get $Var(\mathcal{T}'') \subseteq Var(\mathcal{T}') \subset Var(\mathcal{T})$ so $|\mathcal{T}''| < |\mathcal{T}'|$; (8) and we can use the induction hypotheses to get 3(a) with $unify\,\mathcal{T}'' \rightsquigarrow (, {}', S')$.

For (6) (the only open assumption yet) we have to validate 2(a). We are doing this by defining $\mathcal{T}_i = (\tau_1, \tau_2, , {}'_{i-1}, S'_i, r'')$ with $r'' \in \pi_{\alpha_i}$ for $i \in \{1, \ldots, n\}$ and then proving $|\mathcal{T}_i| < |\mathcal{T}'|$ (9) so that induction hypotheses on *mkinst* is applicable.

Compare $|S_i''\alpha_i|$ with $|S_1\alpha|$ to see that the second component of the measure on typing states is decreased to get (9) . By a straight inductive argument we have $S_i'' < S_1$. So we have to consider two cases:

- $\mathbf{S_i''}\alpha = \mathbf{S_1}\alpha$ Then definition of measure $|.|$ on types gives $|S_{i-1}''\alpha \downarrow p(i)| = |\tau_i| < |T\overline{\tau}| = |S_1\alpha|$. By decreasing the second component of measure on typing states we get (9) .
- $\mathbf{S_i''}\alpha \neq \mathbf{S_1}\alpha$ Then we have $dom(S_{i-1}'') \supset dom(S_1)$ and by decreasing the first component of measure on typing states we get (9) □

**3(b)** We have to prove for another solution $(, '', S'')$ of the restriction problem $(r')(, _1, S_1)$ that $(, '', S'') \leq (, ', S')$. Lemma 3.4 gives $S'', '' \vdash S''\pi_{\alpha_i}$, by use of inequality (9) (t.i. $|\mathcal{T}_i| < |\mathcal{T}'|$) we can apply induction hypotheses 2(b) on *mkinst* to get $(, '', S'') \leq (, _i', S_i')$. Lemma 3.4 gives $S''\tau' = S''\tau''$, by use of inequality (8) (t.i. $|\mathcal{T}''| < |\mathcal{T}'|$) we can apply induction hypotheses 1(b) on $unify$ to get $(, '', S'') \leq (, ', S')$ □.

# 4 Soundness and Completeness of the type inference

This is the section which explains and constructs the main result. First we start with the observation that we have to focus on transformations of derivations. This leads in 4.1 to the result that logical derivations can be normalized. In the following a closer look to the entailment $\vdash$ on typotheses leads to the notion of generic instances. In 4.2 we characterize the notion of more general typotheses this lead to definition of constrained substitution. So in 4.3 we are able to state that *unify* computes the most general constrained substitution for a unification problem. The technical lemmas with proofs can be found in 4.4. The deepest insight in the overloading theory gives the right extension lemma which is a commutation inequality between constrained substitutions and the generalization function. After the discussion of the full results in 4.5 the soundness proof can be found in 4.6 and the completeness proof is detailed in 4.7

## 4.1 Transformation of Derivations

Let us motivate the classical soundness and completeness results. We construct a algorithmical type system (see figure 7) which corresponds to the logical type system. Then we want to give a good relation between those systems. The best would be the equivalence :

$$, \vdash e : \sigma \Leftrightarrow , \vdash^W e : \sigma$$

The conscious reader will realize a flaw in the formula above. Given a typing problem $(, , e)$ the algorithm W can only compute a type $\tau$ not a type scheme $\sigma$ as result. Further in the logical system there are several possibilities to derive types for a given problem $(, , e)$ opposed to the deterministic algorithm.

To get a theoretical satisfying connection between the logic and the algorithmic type system we split the equivalence in its implications.

The simple direction is *soundness* $, \vdash^W e : \tau \Rightarrow , \vdash e : \tau$ which tells us the correctness of the algorithm. This is because the soundness property means that algorithmic type results can be derived logical. The soundness proof gives a concrete meaning to this statement in constructing a derivation for $, \vdash e : \tau$ in the logical system from the given computation $, \vdash^W e : \tau$ . So we are talking about derivations, that is proof trees with rules marking nodes and judgments marking edges and the root is the resulting judgment. Let's denote a derivation as $d : , \vdash e : \sigma$ opposed to the judgment $, \vdash e : \sigma$. If we take such a derivation $d : , \vdash e : \sigma$ and transform it by structural induction on the proof tree to a new derivation $d' : , ' \vdash e : \sigma'$ we will denote this in short form as $, \vdash e : \sigma \rightsquigarrow , ' \vdash e : \sigma'$.

So here is what the soundness result really gives $, \vdash^W e : \tau \rightsquigarrow , \vdash e : \tau$ that is a derivation transformation from the *typing computations to a logical typing derivations.*

The complicated direction of the equivalence is completeness which should look like $, \vdash e : \sigma \;\Rightarrow\; , \vdash^{W} e : \sigma$. But this can't work, so we need the notion of the generic instance of a type scheme. This notion is a relation between type schemes $\sigma$ and $\sigma'$ and can be stated like $\forall e., \vdash e : \sigma \rightsquigarrow , \vdash e : \sigma'$. This means given a type scheme $\sigma$ as result of a typing, then all its generic instances $\sigma'$ can also be logically derived for the same problem; that is $\sigma$ is more general than its generic instances. We need a function *gen* to formulate the completeness result.

> Given a derivation $d : , \vdash e : \sigma$ we can construct a computation $, \vdash^{W} e : \tau$ so that $\sigma$ is a generic instance of $\sigma' = gen(, , \tau)$.

Now result $\tau$ of the algorithm W does depend only on problem $, , e$ not on type scheme $\sigma$ so the result implies $\sigma'$ is the*most general* type for the problem. The point is that we have now a derivation which can be used to construct all other possible logical solutions for the problem. We restate this as

> **Completeness:** If there is a logical solution to the problem $(, , e)$ then we construct a derivation $d : , \vdash e : \sigma'$ so for every other solution holds $, \vdash e : \sigma' \rightsquigarrow , \vdash e : \sigma$

At least let us look at some derivation transformations which are used in the following proofs. Let's assume we are given a derivation $d : , \vdash e : \sigma$.

If we can apply rules (that are $(\forall I)$ or $(\forall E)$) of the logical system on the derivation then we get a transformation $, \vdash e : \sigma \rightsquigarrow , \vdash e : \sigma'$

If we have some constraint set $C$ then we get the transformation $, \vdash e : \sigma \rightsquigarrow , .C \vdash e : \sigma$ because side condition of $(\forall I)$ can't be violated. The side conditions are stated for type variables that are bound or consumed in the judgment $, \vdash e : \sigma$ so they can't interfere with the type variables of $C$.

### 4.1.1 Normalized derivations

Given the following derivation $d$ for the judgment $, \vdash e : \sigma$ so that $\sigma = [\tau/\alpha]\sigma'$

$$(\forall E)\quad \dfrac{(\forall I)\quad \dfrac{\dfrac{\vdots\, d'}{, .\pi_\alpha \vdash e : \sigma' \quad \alpha \notin tv(, )}}{, \vdash e : \forall\alpha.\pi_\alpha \Rightarrow \sigma'} \qquad , \vdash [\tau/\alpha]\pi_\alpha}{, \vdash e : [\tau/\alpha]\sigma'}$$

We want to reduce the derivation to one without the last use of the $(\forall I)(\forall E)$ rule pair . In order to do that, we apply substitution $[\tau/\alpha]$ on derivation $d'$. So we get a new derivation $d_1 : [\tau/\alpha], .\pi_a \vdash e[\tau/\alpha]\sigma'$ that is $d_1 : , .[\tau/\alpha]\pi_\alpha \vdash e : \sigma$ because of the side condition $\alpha \notin tv(, )$. If we substitute every $(TAUT)$ for a $o : \tau' \in [\tau/\alpha]\pi_\alpha$ in derivation $d_1$ by the derivation $, \vdash o : \tau'$ given through the right premise $, \vdash [\tau/\alpha]\pi_\alpha$ of the $(\forall E)$ rule application, then we get the new derivation $d_2 : , \vdash e : \sigma$. This kind of derivation transformation is called $(\forall I)(\forall E)$-*elimination*.

Given the compound expression $e$ (that is $e \neq x$ for a term variable $x$) a type $\tau$ and a derivation $d : , \vdash e : \tau$ then we can transform the derivation, so that the last applied rule corresponds to the expression ( that is for $e = \lambda x.e'$ last applied rule should be $(\rightarrow I)$; or $e = e'e''$ corresponds to $(\rightarrow E)$ etc.). We call the new derivation *root normalized* derivation for judgment $, \vdash e : \tau$. If this condition holds for every node $,' \vdash e' : \tau' \quad e'$*compound* in the derivation, then we call this a *normalized* derivation.

We can normalize derivation $d$ by the use of $(\forall I)(\forall E)$-elimination. Because in every derivation of judgment $, \vdash e : \tau$ there must be a root normalized derivation $d' : ,' \vdash e : \tau'$ followed by applications of $(\forall I)$ and $(\forall E)$ rules. By $(\forall I)(\forall E)$-elimination we get a root normalized derivation for $, \vdash e : \tau$ . By structural induction on the derivation we get a normalized derivation.

## 4.2 More general relation on typotheses and type schemes

We want to understand first the entailment $\vdash$ on typotheses. We define $, \le ,'$ iff $(\forall x : \sigma \in ,' \Rightarrow , \vdash x : \sigma)$ that is $, \vdash ,'$. If we have $, \le ,'$ then $(\forall e.,' \vdash e : \sigma \rightsquigarrow , \vdash e : \sigma)$ if further $(\forall o : \sigma_T \in , .\exists o : \sigma'_T \in ,'$(invariance of instance declarations)). Because in a given derivation $,' \vdash e : \sigma$ we can substitute every application of $(TAUT)$ for a type variable $x : \sigma \in ,'$ by a derivation $, \vdash x : \sigma$. The side condition of $(INST)$ is not violated, and the side condition of $(\forall I)$ can be satisfied by renaming of bound type variables.

What is a derivation of $, \vdash x : \sigma$ like ? The rules $(\rightarrow I)(\rightarrow E)(LET)(INST)$ can't be applied because $x$ has no term structure. $(TAUT)$ must be used for term variable $x$ so that $x : \sigma' \in ,$ . After using $(TAUT)$ all rules applicable on the structure of type scheme $\sigma'$ can be used. The rule $(SET)$ will be only applied if its conclusio is used as right premise $S\pi_\alpha$ of rule $(\forall E)$. Let $S\pi_\alpha = \{o_i : S\alpha \rightarrow \tau_i\}$ and typotheses , have structure $, = T.C$ (where $T$ is the set of all declarations $x : \sigma \in ,$ and $C$ is the set of all constraints $o : \alpha \rightarrow \tau \in ,$ ) then premises of $(SET)$ can only be derived with use of $(TAUT)$ on $C$ or as conclusio of $(\forall E)$ rule this is the same result as lemma Lemma 3.4

This observation is put together by the notion of generic instance on type schemes.

**Definition.** (generic instances)

$$\sigma' \le^g_\Gamma \sigma \text{ iff new } u \qquad , ,u : \sigma' \vdash u : \sigma \text{ iff new } u \qquad , ,u : \sigma' \vdash^g u : \sigma$$

where $\vdash^g$ is defined by the system of rules $(TAUT)(\forall I)(\forall E)$ and

$$(SET)^{SC}\ \frac{, \vdash^g o_1 : \tau_1 \ldots , \vdash^g o_n : \tau_n}{, \vdash^g o_1 : \tau_1 \ldots o_n : \tau_n}$$

Now we characterize the more general relation on typotheses.

**Definition.** We define to be , more general than $,'$ in short $, \le ,'$ iff the following equivalent properties hold:

- $\forall x : \sigma \in ,' \Rightarrow , \vdash x : \sigma$ that is $, \vdash ,'$.
- $\forall x : \sigma \in ,' \Rightarrow x : \sigma' \in , ; \wedge \sigma' \le^g_\Gamma \sigma$ that is $, \vdash^g ,'$.

if we have property $\forall o : \sigma_T \in , .\exists o : \sigma'_T \in ,'$ then

- $\forall e.,' \vdash e : \sigma \Rightarrow , \vdash e : \sigma$

Let's see in concrete what does the structure of the type schemes $\sigma, \sigma'$ tell us about the structure of a derivation for $\sigma \le^g_\Gamma \sigma'$. We assume as structure of type schemes $\sigma = \forall.\overline{\alpha}.C \Rightarrow \tau$ and $\sigma' = \forall.\overline{\beta}.C' \Rightarrow \tau'$ from this follow the data; a substitution $S$ with $dom(S) \subset \{\overline{\alpha}\}$ and $S\tau = \tau'$ and $, ,C' \vdash SC$. So the derivation $, ,u : \sigma \vdash u : \sigma'$ has following normal form. First apply $(TAUT)$ then for every $\alpha \in dom(S)$ apply $(\forall E)$ where we use the judgment $, ,C' \vdash SC$ for the left premise so we get $, ,C', u : \sigma \vdash u : \tau'$, on this apply $(\forall I)$ until we reach $, ,u : \sigma \vdash u : \sigma'$ as natural side condition we need $\{\overline{\beta}\} \cap (tv(,) \cup fv(\sigma)) = \emptyset$ which is no problem by alpha-renaming of the $\overline{\beta}$.

Here are some special cases for the generic instance relation . $\tau \le^g_\Gamma \tau'$ iff $\tau = \tau'$ because $\tau$ has no bound variables so the only usable substitution to match $\tau'$ is $id$. We have $\tau \le^g_\Gamma \sigma$ iff $\sigma = \forall\overline{\alpha}.C \Rightarrow \tau$ where $\{\overline{\alpha}\} \cap tv(\tau) = \emptyset$ with the same argument as above.

### 4.2.1 Constrained type schemes

**Definition.** (weak generic instances of constrained type schemes)

We call a tuple $(C, \sigma)$ a constrained type scheme where $C$ is a constrained set and $\sigma$ is a type scheme. We define the weak instance relation $(C, \sigma) \le^w_T (C', \sigma')$ by the reflexive transitive relation on constrained type schemes syntactical induced by the rules :

$$(\forall L) \quad \frac{\pi_\alpha \subseteq \pi'_\alpha \qquad \alpha \notin \mathit{tv}(T.C,\sigma)}{(C.\pi_\alpha,\sigma) \leq^w_T (C,\forall\alpha.\pi'_\alpha \Rightarrow \sigma)}$$

$$(\forall R) \quad \frac{T.C \ \vdash\ [\tau/\alpha]\pi_\alpha}{(C,\forall\alpha.\pi_\alpha \Rightarrow \sigma) \leq^w_T (C,[\tau/\alpha]\sigma)}$$

Note that the notion of this relation leads to a transformation of typing derivations

$$\boxed{(C,\sigma) \leq^w_T (C',\sigma') \implies T.C \ \vdash\ e:\sigma \leadsto T.C' \ \vdash\ e:\sigma'}$$

This can easy be seen by taking the derivation of $(C,\sigma) \leq^w_T (C',\sigma')$ and applying $(\forall I)$ on the judgment when $(\forall L)$ was used; and applying $(\forall E)$ when $(\forall R)$ was used. This construction is possible because of the side condition $\alpha \notin \mathit{tv}(T.C,\sigma)$ in rule $(\forall L)$. So the derivation of the instance relation is used to construct the proof transformation of typings.

Further we can construct another typing transformation

$$\boxed{(C,\sigma) \leq^w_T (C',\sigma') \implies T.C'.u:\sigma' \ \vdash\ e:\sigma'' \leadsto T.C.u:\sigma \ \vdash\ e:\sigma''}$$

Notice the contravariance in the usage of the constrained type schemes

**Proof:** The instance relation gives by using the covariant lemma a transformation of proof $(TAUT)$ $T.C.u:\sigma \ \vdash\ u:\sigma$ to a proof $T.C'.u:\sigma \ \vdash\ u:\sigma'$ (1) . Now we substitute every use of $(TAUT)$ $T.C'.u:\sigma' \ \vdash\ u:\sigma'$ by (1) and get the proof of $T.C'.u:\sigma \ \vdash\ e:\sigma''$ (2) .

By a simple inductive argument on the use on $(\forall L)$ rules in the proof of $(C,\sigma) \leq^w_T (C',\sigma')$ we get $C' \subseteq C$ and for $\mathcal{V} \overset{\text{def}}{=} \mathit{tv}(C) \setminus \mathit{tv}(T.C')$ we get $C' = C \setminus C_{\mathcal{V}}$. So if we take proof (2) and substitute constrained set $C'$ by $C$ the side condition of rule $(\forall I)$ stays invariant :

$$\forall\alpha \in \mathit{tv}(T.C'). \qquad \alpha \notin \mathit{tv}(C' \setminus C'_\alpha) \implies \alpha \notin \mathit{tv}(C \setminus C_\alpha)$$

So we transform (2) to a proof of $T.C.u:\sigma \ \vdash\ e:\sigma''$ □

We have type invariance under weak instance relation that is $(C,\tau) \leq^w_T (C',\tau') \ \Rightarrow\ \tau = \tau'$. The only applicable rule on left constrained type scheme is $(\forall L)$ but side condition permits binding a type variable $\alpha \in \mathrm{tv}(\tau)$. So applying rule $(\forall R)$ afterwards does not change $\tau$ so only identity is possible.

### 4.2.2 The generalization function

**Definition.** (generalization function *gen*)

$$\mathit{gen}(T.C,\sigma) \ \overset{\text{def}}{=}\ (C \setminus C_{\overline{\alpha}}, \forall\overline{\alpha}.C_{\overline{\alpha}} \Rightarrow \sigma)$$

where $\overline{\alpha}$ is the maximal sequence $\overline{\alpha} = \alpha_1 \cdots \alpha_n$ such that $\alpha_i \notin \mathit{tv}(T.C \setminus \{\alpha_1 \cdots \alpha_i\})$ and $\alpha_i \in \mathit{tv}(\sigma)$ Further $g(\,,\sigma)$ is defined by applying function *gen* on the argument and taking only the type scheme of the result. By this definition we get the following properties:

**(gen1)** If $\mathit{gen}(\,,\sigma) = (\,,{}',\sigma')$ then $\forall e.,\ \vdash\ e:\sigma \leadsto\ ,{}'\ \vdash\ e:\sigma'$ by using $(\forall I)$ rules.

**(gen2)** If $\mathit{gn}(\,,\sigma) = \sigma'$ then $\forall e.,\ \vdash\ e:\sigma \leadsto\ ,\ \vdash\ e:\sigma'$. This can be seen by using first above transformation $,\ \vdash\ e:\sigma \leadsto\ ,{}'\ \vdash\ e:\sigma'$ so that $C = \,,\ \setminus\ ,{}'$ are constraints consumed by $(\forall I)$ rule but we can extend a proof by $C$ so $,{}'\ \vdash\ e:\sigma' \leadsto\ ,{}'.C = \,,\ \vdash\ e:\sigma'$.

**(gen3)** If $\mathit{gen}(\,,\sigma) = (\,,{}',\sigma')$ then $\forall\alpha \notin \mathit{tv}(\sigma).,_\alpha = \,,{}'_\alpha$ by the side condition $\alpha_i \in \mathit{tv}(\sigma)$.

**(gen4)** $\mathit{gen}(T.C.\pi_\alpha,\sigma) = \mathit{gen}(T.C,\forall\alpha.\pi_\alpha \Rightarrow \sigma)$ for $\alpha \notin \mathit{tv}(T.C)$ and $\alpha \in \mathit{tv}(\sigma)$.

**(gen5)** By maximality we get for $gen(T.C,\sigma) = (T.C',\sigma')$ then $\forall\alpha \in \mathrm{tv}(\sigma).\alpha \notin tv(T.C' \setminus \alpha) \Rightarrow C'_\alpha = \emptyset$.

**(gen6)** $gen(\,,\sigma) = (\,',\sigma') \Rightarrow gen(\,.\pi_\alpha,\sigma) = (\,'.\pi_\alpha,\sigma')$ if $\alpha \notin tv(\sigma)$.

By using property **(gen6)** we get a typing transformation principle for type systems defined with use of function *gen*.

$$\boxed{T.C \vdash^{W,d} e : \tau \rightsquigarrow T.C.C' \vdash^{W,d} e : \tau}$$

We can assume that the type variables of $C'$ can't conflict with the type variables bound by *gen* in the derivation of $T.C \vdash^{W,d} e : \tau$. So we can apply property **(gen6)** that rule applications of $(LET)^{W,d}$ stay correct if extending the constraint set by $C'$. No side conditions of other rules in the type systems $W, d$ are hurt by extending the constrained set, so we get the result.

## 4.3 Contradictions and constrained substitutions

Let's see what is the substance of the results on *unify*. In the HM-system (ours without overloading) substitutions are applied on type derivations to prove soundness of the inference algorithm. It is an application of the implication $(\, \vdash^{HM} e : \sigma \Rightarrow S\, \vdash^{HM} e : S\sigma)$. A look at system O convinces that this holds also for overloading. But in our system with constraints, a substitution makes only sense if the substituted constraints can be satisfied. So the inference mechanism has to check satisfiability for all instantiations appearing in the program.

Given a typotheses , with restriction $o : \alpha \to \tau \in \,$, and for a type constructor $T$ there is no type scheme $\sigma_T$ with $o : \sigma_T \in \,$, then given a substitution $S$ with $S\alpha = T\overline{\tau}$ typotheses $S\,$ makes no sense because $o : T\overline{\tau} \to S\tau$ can't be satisfied by a declaration $o : \sigma_T$. So implication $(\, \vdash^{O} e : \sigma \Rightarrow S\, \vdash^{O} e : S\sigma)$ is correct but may be senseless.

We define the set of restriction variables for a typotheses , as $rv(\,) \stackrel{\text{def}}{=} \{\alpha | o : \alpha \to \tau \in \,\}$. A substitution $S$ is called *conflict free* for typotheses , iff $dom(S) \cap rv(\,) = \emptyset$, so every configuration $(\,,S)$ is a tuple of a typotheses , with a conflict free substitution belonging to it. The set of determined type variables for a substitution is defined as $dv(S) \stackrel{\text{def}}{=} \{\alpha | S\alpha = T\overline{\tau}\}$.

Next we want to understand how conflicting substitutions operate on typotheses. In the following we will talk about *structured* typotheses. We define the application of a substitution on a typotheses , converges if the resulting constraints can be "restructured", that is if they don't contradict. We want that all determined type variables of a substitution must be satisfied if they are constrained. In short for a typotheses , $= T.C$ :

$$S\, \downarrow \text{ iff } \exists C' \qquad ST.C \setminus dv(S).C' \vdash SC_{|dv(S)}$$

the data can be denoted as $S\, \downarrow C'$ or $S\, \downarrow \,'$ or $\, \leq^{(S,C')} \,'$ where $\,' = ST.C \setminus dv(S).C'$ is the right side of the judgment. The idea is that $\,'$ is a conservative extension of $S\,$, so property $\,' \vdash S\,$ should hold, which is given by the definition. So the constraint set $C'$ are all the variable restrictions which are used in $(TAUT)$ rules as premises for applying $(SET)^{SC}$ in derivation of the judgment. The dual notion of divergence is defined as $S\, \uparrow$ iff a constrained set with property above does not exist, that is there exists $o : T\overline{\tau} \to \tau' \in S\,$ and there is no instance declaration $o : \sigma_T \in \,$.

So we define a tuple $(S,C')$ of a substitution $S$ and a constrained set $C'$ with $SC' = C'$ to be *constrained* substitution and use letters $U, V, W$ to denote them. Further

$$(S,C')\, \downarrow \text{ iff } T.C \setminus dv(S).C' \vdash SC_{|dv(S)} \qquad \text{for } \, = T.C$$

dually the notion of diverging application of a constrained substitution $(S,C')$ on a typotheses , denoted as $(S,C')\, \uparrow$ iff $S(\,.C') \uparrow$. The characterization of set $C'$ gives the existence of a minimal set if $S\, \downarrow$, where minimality means every $r \in C'$ is used in a $(TAUT)$ for deriving the "restructuring" judgment. We define this as the minimal conservative extension of $T.C$ if applying $S$.

$$C' = mce(S, T.C) \qquad \text{iff} \qquad C' \text{ minimal an } ST.C' \vdash SC$$

**Lemma 4.1** (constrained substitution on derivations) If $, \ \vdash e : \sigma$ and $(S, C'), \downarrow , '$ then $, ' \vdash e : S\sigma$. If $(S, C'), \uparrow$ and $, = T, C$ then there is no constrained set $C''$ such that $ST, C \setminus dv(S), C'' \vdash e : S\sigma$.
**Proof:** Clearly $S, \ \vdash e : S\sigma$ is a derivation in System O. Substitute in this derivation every use of $(TAUT)$ for a $r \in SC_{|dv(S)}$ by a derivation of $, ' \vdash r$, which is possible by definition of $S, \downarrow , '$. So we get a derivation of $, ' \vdash e : S\sigma$.

The proof for the dual uses the similar argument as above.

**Definition.** (variable sets for constrained substitutions)
Let $U = (S, C)$ be a constrained substitution then we define the following type variable sets

$$\begin{array}{rcc} codom(S) & \overset{\text{def}}{=} & \bigcup\{tv(S\alpha) \mid \alpha \in dom(S)\} \\ dom(U) & \overset{\text{def}}{=} & dom(S) \\ dv(U) & \overset{\text{def}}{=} & dv(S) \\ rv(U) & \overset{\text{def}}{=} & rv(C) \\ tv(U) & \overset{\text{def}}{=} & tv(C) \\ v(U) & \overset{\text{def}}{=} & dom(U) \cup codom(U) \cup tv(U) \end{array}$$

### 4.3.1 The connection between configurations and constrained substitutions

Now here is the connection to configurations

**Lemma 4.2** Characterization of $\le$ on configurations with $\le^U$ on typotheses.

$$(, , S) \le (, ', S') \Leftrightarrow S, \le^{(R,C'')} S', '$$
$$\text{where } S \le^R S' \text{and } S, = T.C \text{and } S, ' = T.C' \text{and } C'' = R(C' \setminus C)$$

**Proof:** Definition of $\le$ on configurations gives the existence of $R$ with $S \le^R S'$. Definition of configurations give that $S,$ is structured so we can define $S, = T.C$. The same argument gives the structure of $S', ' = RS, '$ so $S, '$ is structured and we can define $S, ' = T'.C'$. *The only condition needed for the equivalence is $T = T'$*

Let's look on the structure of $S', '$ we get $S', ' = RS, ' = RT.C' = RT.RC'$. So $C'$ is a set of constraints which is $C'_{|dv(R)} = \emptyset$. So we have $C' \setminus C = C' \setminus (C \setminus dv(R))$ This leads to $RC' = R(C \setminus dv(R).C' \setminus C) = R(C \setminus dv(R)).R(C' \setminus C) = R(C \setminus dv(R)).C''$ (1) . With this material we can complete the proof.

$$\begin{array}{rrl} & & S', ' \vdash S', \\ \Leftrightarrow & S', ' = & RS, ' \vdash RS, \\ \Leftrightarrow & S', ' = & RT.C' \vdash RT.C \\ \Leftrightarrow & S', ' = & RT.C' \vdash RC \\ \Leftrightarrow & S', ' = & RT.C \setminus dv(R).C'' \vdash RC \\ \Leftrightarrow & S', ' = & RT.C \setminus dv(R).C'' \vdash R(C_{|dv(R)}) \\ \Leftrightarrow & & (R, C'')T.C \downarrow S', ' \\ \Leftrightarrow & & (R, C'')S, \downarrow S', ' \\ \Leftrightarrow & & S, \le^{(R,C'')} S', ' \end{array}$$

The proof tells us that *if declarations of , and , ' coincide then the notion of more general configurations and structured typotheses related by constrained unification coincide.* A look on the unification algorithm in 3 gives that in the computation the declarations stay unchanged.

Now we define $U \le_\Gamma^V W$ iff $U, \downarrow , ' \wedge V, ' \downarrow , '' \wedge W, \downarrow , ''$. Transitivity of $\le_\Gamma$ on constrained substitution for typotheses , is easy to see, further we define composition as $VU \overset{\text{def}}{=} W$ iff $U \le_\Gamma^V W$.

We define a constrained unifier $U$ of an unifying problem $(, )(\tau_1, \tau_2)$ as $\tau_1 \overset{U}{\approx}_\Gamma \tau_2$ iff $U$ is a unifier of $\tau_1, \tau_2$ and $U, \downarrow$. Further the most general constrained unifier is defined as $\tau_1 \overset{U}{\sim}_\Gamma \tau_2$ iff $\tau_1 \overset{U}{\approx}_\Gamma \tau_2$

and $\forall V.\tau_1 \overset{V}{\approx}_\Gamma \tau_2 \;\Rightarrow\; U \leq_\Gamma V$. The minimality property gives the following characterization;if $(S,C)$ is the most general constrained unifier of the problem $(,\ )(\tau_1,\tau_2)$ then $C = mce(S,,\ )$ that means $C$ is the minimal conservative extension of , if applying $S$ on , .

Now theorem Lemma 3.7 tells that *unify* computes the most general unifying configuration to a unifying problem $(S,\ )(\tau_1,\tau_2)$. Using the characterization of $\leq$ on configurations this gives the result :

$$unify(\tau_1,\tau_2)(,\ ,S) = (,\ ',S') \text{ with } S \leq^R S' \text{ and } C'' = R(S,\ \setminus S,\ ') \;\Leftrightarrow\; \tau_1 \overset{(R,C'')}{\sim}_{S\Gamma} \tau_2$$

So if we take $S$ as $id$ we use *unify* to define algorithm *mgcu* and have result :

$$mgcu(,\ )(\tau_1,\tau_2) = (R,C'') \;\Leftrightarrow\; \tau_1 \overset{(R,C'')}{\sim}_{\Gamma} \tau_2$$

Recall the syntactic characterization of the general instance relation on type schemes $\sigma \leq^g_\Gamma \sigma'$. Let us translate this notion in terms of constrained substitutions. We have a simple substitution $S$ with $dom(S) \subseteq bv(\sigma)$. So we can assume $dom(S) \cap tv(,\ ) = \emptyset$ that gives $S,\ \downarrow,$ altogether side condition $,\ .C' \;\vdash\; SC$ gives a constrained substitution $U = (S,C')$ and side condition becomes $U,\ .C\downarrow$.

So $\forall.\overline{\alpha}.C \Rightarrow \tau \leq^g_\Gamma \forall.\overline{\beta}.C' \Rightarrow \tau'$ iff $\exists U$ and $U = (S,C')$ and $dom(U) \subseteq \{\overline{\alpha}\}$ and $U\tau = \tau'$ and $U,\ .C\downarrow$.

## 4.4 Relations between judgments

We did define the more general relation and constrained substitutions on typotheses and type schemes. Now we extend this naturally on derivations, so we get the notion of relating two derivations by a substitution or the generality relation. We use this to present by diagrams what our type theoretic questions are about.

First we define $(\sigma,,\ ) \xrightarrow{\leq} (\sigma',,\ ')$ (or $(\sigma,,\ ) \leq (\sigma',,\ ')$ ) iff $,\ = T.C$ and $,\ ' = T.C'$ and $(C,\sigma) \leq^w_T (C',\sigma')$. As described in section 4.2 this can be extended on derivation, so given derivation $d :,\ \vdash e:\sigma$ we get $d' :,\ ' \vdash e:\sigma'$ in short $,\ \vdash e:\sigma \xrightarrow{\leq} ,\ ' \vdash e:\sigma'$ .

Let's do the analog with substitutions $(\sigma,,\ ) \xrightarrow{U} (\sigma',,\ ')$ iff $U,\ \downarrow,\ ' \wedge U\sigma = \sigma'$, look back to section 4.3 to trust the extension on derivations so given derivation $d :,\ \vdash e:\sigma$ we get $d' :,\ ' \vdash e:\sigma'$ in short $,\ \vdash e:\sigma \xrightarrow{U} ,\ ' \vdash e:\sigma'$ .

Remember the definition of function *gen* in section 4.2.2. We restate one of its property given $(\sigma,,\ ) \xrightarrow{gen} (\sigma',,\ ')$ iff $gen(,\ ,\sigma) = (,\ ',\sigma')$ then $,\ \vdash e:\sigma \xrightarrow{gen} ,\ ' \vdash e:\sigma'$ . Further we are able to denote the function application of $g$ as an arrow $(\sigma,,\ ) \xrightarrow{g} \sigma'$

Given $(\tau,,\ ) \xrightarrow{gen} (\sigma,,\ ')$ and $U,\ '\downarrow$ if $\sigma$ has structure $\sigma = \forall\overline{\alpha}.C' \Rightarrow \tau$ we assume $v(U) \cap \{\overline{\alpha}\} = \emptyset$ which can be achieved by alpha-renaming. Because of $,\ _{\overline{\alpha}} = C' \;\wedge\; ,\ ' = ,\ \setminus C'$ we get $U,\ \downarrow$. Let constrained substitution $U$ have structure $U = (S,C)$ then $(U,\ )_{\overline{\alpha}} = S(,\ _{\overline{\alpha}}) = SC'$, further $U\sigma = \forall\overline{\alpha}.SC' \Rightarrow S\tau$ altogether we get a lemma.

**Lemma 4.3** (left extension) Given $gen(\tau,,\ ) = (\sigma,,\ ')$ and $U,\ '\downarrow$ with $v(U) \cap bv(\sigma) = \emptyset$ then $U \circ gen(\tau,,\ ) = gen \circ U(\tau,,\ )$. If $U = (S,C)$ we get graphically

$$\begin{tikzcd}
(\tau,,\ ) \arrow[r, "gen"] \arrow[d, dotted, "U"] & (\sigma,,\ ') \arrow[d, "U"] \\
(U\tau,U,\ ) \arrow[r, "gen"] & (U\sigma,U,\ ')
\end{tikzcd}$$

**Lemma 4.4** (substitution on generality relation) Given $\sigma \leq^g_\Gamma \sigma'$ and $U, \downarrow$ then $U\sigma \leq^g_{U\Gamma} U\sigma'$.
**Proof:** We assume $\sigma = \forall\overline{\alpha}.C \Rightarrow \tau$ and $\sigma' = \forall\overline{\beta}C' \Rightarrow \tau'$. Because $\sigma$ is more general than $\sigma'$ we have substitution $S$ with $dom(S) \subseteq \{\overline{\alpha}\}$ and $S\tau = \tau'$ and $, , C' \ \vdash \ SC$.

We can assume $U = (S', C'')$ and $v(U) \cap \{\overline{\alpha}, \overline{\beta}\} = \emptyset$ so $U, , C' \downarrow$ leads to $U, , C' \ \vdash \ S'SC$ clearly

$S'S\tau = S'\tau'$ altogether we have the result $U\sigma \leq^g_{U\Gamma} U\sigma'$.

**Corollary 4.5** (operation of constrained substitutions on deterministic derivations)

Given $, \vdash^d e : \tau$ and $U, \downarrow$ then $U, \vdash^d e : U\tau$

**Proof:** By induction on the structure of the derivation. The cases for rules $(\rightarrow I)^d, (\rightarrow E)^d$ are trivial if using the induction hypotheses on the premises.

For $(LET)^d$ use the "left extension" for the middle premise. If last rule application of derivation has scheme below

$$(LET)^d \quad \frac{, \vdash^d e : \tau' \qquad g(\tau', , ) = \sigma \qquad , , u : \sigma \vdash^d e' : \tau}{, \vdash^d \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

Then application of $U$ on the left and right premises is allowed by induction hypotheses. Application of $U$ on the middle premise gives the equations : $g(U\tau', U, ) = g \circ U(\tau', , ) \overset{Lemma\ 4.3}{=} U \circ g(\tau', , ) = U\sigma$

For $(TAUT)^d$ see that $U$ operates naturally on the premises.

$$(TAUT)^d \quad \frac{x : \sigma \in , \qquad \sigma \leq^g_\Gamma \tau}{, \vdash^d x : \tau}$$

We get surely $x : U\sigma \in U, $. The lemma above gives $U\sigma \leq^g_{U\Gamma} U\tau$. So we can apply again $(TAUT)^d$ to get $U, \vdash^d x : U\tau$.

Now let's see if we are able to construct the dual to Lemma 4.3; the right extension lemma. We start with $(\tau, , )$ and $U$ with $U, \downarrow$ and ask if it is possible to commute application of $g$ and $U$ on tuple $(\tau, , )$. We start with the simple situation

$$\begin{tikzcd}
(\tau, , ) \arrow[r, "g"] \arrow[d, "U"] & \sigma \arrow[d, dotted, "?"] \\
(U\tau, U, ) \arrow[r, "g"] & \sigma'
\end{tikzcd}$$

For the upper arrow we get the data $, = , ''.C''$ , $\sigma = \forall\overline{\alpha}.C'' \Rightarrow \tau$ , $\{\overline{\alpha}\} \cap tv(, '') = \emptyset$. The lower arrow gives data $U, = U, '.C'$ and $\sigma' = \forall\overline{\beta}.C' \Rightarrow U\tau$ and $\{\overline{\beta}\} \cap tv(U, ') = \emptyset$. Let $, = T.C$ and $U = (S, C_1)$ we define $S_1 = S \setminus \{\overline{\alpha}\}$ , $S_2 = S_{|\{\overline{\alpha}\}}$. So we get

$$\begin{array}{cccc}
 & U\sigma & \leq^g_{U\Gamma} & \sigma' \\
\Leftrightarrow & \forall\overline{\alpha}.S_1C'' \Rightarrow S_1\tau & \leq^g_{U\Gamma} & \forall\overline{\beta}.C' \Rightarrow S\tau \\
\Leftrightarrow & S_1\tau \leq^{S_2} S\tau & \wedge & U, , C' \ \vdash \ S_2S_1C'' \\
\Leftrightarrow & \text{true by definition} & \wedge & U, \ \vdash \ SC'' \\
\Leftrightarrow & \text{true because} & & U, \downarrow \text{implies } U, \ \vdash \ S, \ \text{ and } SC'' \subseteq S,
\end{array}$$

This is the following lemma.

**Lemma 4.6** (simple right extension) Given a tuple $(\tau, , )$ and a constrained substitution $U$ with $U, \downarrow$ then

$$U \circ g(\tau, , ) \leq g \circ U(\tau, , )$$

or graphical

$$\begin{tikzcd}
(\tau, , ) \arrow[r, "g"] \arrow[dd, "U"] & \sigma \arrow[d, dotted, "U"] \\
 & U\sigma \arrow[d, dotted, "\leq^g_{U\Gamma}"] \\
(U\tau, U, ) \arrow[r, "g"] & \sigma'
\end{tikzcd}$$

Now we want to extend this result to include observations how generalization behaves if consuming constraints in the typotheses. For this closer look we invent some new notions. We define the dependence relation between type variable for a given typothes , as a relation $\rightarrow\subseteq rv(, ) \times rv(, )$; and $\alpha \rightarrow \beta$ iff $o : \beta \rightarrow \tau \in$ , and $\alpha \in tv(\tau)$. So $\alpha \rightarrow \beta$ tells that $\alpha$ can be $\forall$-quantified with $(\forall I)$ only if $\beta$ will be quantified and consumed before so that restriction $o : \beta \rightarrow \tau$ doesn't hurt the side condition $\alpha \in tv(, \setminus \alpha)$ anymore. We call type variable $\alpha$ *based* if $\alpha \in rv(, )$ and $\alpha \in tv(T)$ where , $= T.C$ so $\alpha$ can never be quantified. This leads to the notions

, **non cyclic** iff $\rightarrow$-relation has no cycles

$\beta$ **non bindable in** , $\beta$ is based or can reach through a $\rightarrow$-path a based type variable.

$C'$ **non bindable in** , iff there is a set $\mathcal{V}$ of non bindable type variables in , and $C' = \, _{\mathcal{V}}$.

$(, , \tau)$ **is fully used** iff for every $\alpha \in rv(, )$ either $\alpha \rightarrow \beta$ or $\alpha$ is based or $\alpha \in tv(\tau)$

$(, , \tau)$ **is clean** iff , is non cyclic and $(, , \tau)$ is fully used.

We want to find out the connection between the bound type variables if applying generalization before and after application of a constraint substitution. Through the new characterization of the non bindable type variables we get the result, that non bind-ability stays invariant under application of constrained substitution. This results by a look at the behavior of $\rightarrow$-paths under substitution. Technically we do this step by the following two lemmas.

**Lemma 4.7** Given typotheses $T.C$ and a terminating substitution on it that is $ST.C\downarrow$. If $r_\beta \in C$ and $\alpha' \in tv(Sr_\beta)$ and $\beta \in dv(S)$ and $\beta \in tv(\sigma)$ for a $x : \sigma \in T.C$ then $\alpha' \in tv(S\sigma)$
**Proof:** We have $r_\beta = o : \beta \rightarrow \tau \in C$. Because of $\beta \in dv(S)$ we can assume $S\beta = K\overline{\tau}$. So termination $ST.C\downarrow$ gives existence of $o : \sigma_K \in T$ so we can assume the structure $\sigma_K = \forall\overline{\alpha}.C' \Rightarrow K\overline{\alpha} \rightarrow \tau'$ and $S$ is a unifier of $K\overline{\alpha} \rightarrow \tau'$ and $\beta \rightarrow \tau$.

So we have $S\tau' = S\tau$ and structure of $\sigma_K$ gives $tv(\tau') \subseteq \{\overline{\alpha}\}$ (1) . As result of $\alpha' \in tv(S\tau)$ there is a path $p$ such that $S\tau|p = \alpha' = S\tau'|p$. Use of (1) gives a subpath $p' \triangleleft p$ with $\tau'|p' = \alpha_i$ so we get $\alpha' \in tv(S\alpha_i)$ (2) .

Now unifying property gives $S\beta = SK\overline{\alpha}$. So from $\beta \in tv(\sigma)$ we get

$$tv(S\beta) \subseteq tv(S\sigma) \;\Leftrightarrow\; tv(SK\overline{\alpha}) \subseteq tv(S\sigma) \;\Rightarrow\; tv(S\alpha_i) \subseteq tv(S\sigma)$$

together with (2) we get $\alpha' \in tv(S\sigma)$ □.

**Remark:** So if $x : \sigma$ is a restriction $o' : \beta \rightarrow \tau''$ the sentence above tells that in case of a chain $\alpha \rightarrow \beta \rightarrow \gamma$ in $T.C$. We get a new dependence $\alpha' \rightarrow \gamma'$ in $ST.C$ if $\beta$ is determined by $S$ and if $S$ renames like $S\alpha = \alpha'$ and $S\gamma = \gamma'$. We call this behavior inductively defined the collapsing of a chain under substitution $S$ to a chain of renamed type variables.

In the case that $x : \sigma$ is $u : \sigma$ or $o : \sigma_K$ we had $\beta$ is a based type variable in $T.C$. Then the sentence above tells us for the dependence $\alpha \rightarrow \beta$ then $\alpha'$ is a based type variable in $ST.C$ if $S$ renames like $S\alpha = \alpha'$.

**Lemma 4.8** (invariance of non bind-ability under substitution) Given $(T.C, \tau)$ and a terminating substitution $S$ on it. If $C$ is non bindable in $T.C$ then $C' \stackrel{\text{def}}{=} mce(S, T.C)$ and $C'' \stackrel{\text{def}}{=} S(C \setminus dv(S))$ is also non bindable in $(ST.C'.C'', S\tau)$

**Proof:**

$\boldsymbol{r_{\alpha'} = o : \alpha' \to \tau \in C'}$ Then there is by definition of $C'$ a $r_\alpha = o : \alpha \to \tau' \in C$ with $\alpha \in dv(S)$ and $r_{\alpha'} \in mce(S, T.r_\alpha)$ such that $\alpha' \in tv(S\alpha)$. Because $\alpha$ is non bindable either $\alpha$ is based or there is a base chain $\alpha \to \ldots \to \gamma$. The remark above implies that $\alpha'$ is either based or has a collapsed base chain. So $\alpha'$ is non bindable.

$\boldsymbol{r_{\alpha'} = o : \alpha' \to \tau \in C''}$ Then we get by definition of $C'$ a constraint $r_\alpha = o : \alpha \to \tau' \in C$ with $Sr_\alpha = r_{\alpha'}$. The same argument as in above case leads from $\alpha$ non bindable to $\alpha'$ non bindable.

Note that the definition of the sets $C'$ and $C''$ are the sets necessary for termination $S(T.C) \downarrow ST.C'.C''$. Further the proof above never used the fact that the full constraint set $C$ is non bindable. So if we have the situation that the subset $C_1 \subseteq C$ is non bindable in $T.C$ then the extension $C_2$ with $S(T.C_1) \downarrow ST.C_2$ is non bindable in $(ST.C, S\tau)$.

Now we are able to prove the full right extension lemma

**Lemma 4.9** (right extension) Given two typings related by a constrained substitution that is $(T.C_1.C_1', \tau') \leq^{(S,C_2)} (ST.C.C', \tau)$ then we can relate its results if generalization is applied. So if $gen(T.C_1.C_1', \tau') = (C_1, \sigma_1)$ and $gen(ST.C.C', \tau) = (C, \sigma)$ then the results are related as

$$T.C_1 \leq^{(S,C_2')} ST.C \;\wedge\; C_2' \subseteq C_2 \;\wedge\; S\sigma_1 \leq^g_{ST.C} \sigma$$

We represent this graphical as

$$\begin{tikzcd}
(T.C_1.C_1', \tau') \arrow[r, "gen"] \arrow[dd, "V=(S,C_2)"] & (T.C_1, \sigma_1) \arrow[d, dotted, "V'=(S,C_2') \wedge C_2' \subseteq C_2"] \\
 & (ST.C, S\sigma_1) \arrow[d, dotted, "S\sigma_1 \leq^g_{ST.C} \sigma"] \\
(ST.C.C', \tau) \arrow[r, "gen"] & (ST.C, \sigma)
\end{tikzcd}$$

**Proof:** We take the structures of the type schemes as $\sigma_1 = \forall\overline{\alpha}.C_1' \Rightarrow \tau'$ and $\sigma = \forall\overline{\beta}.C' \Rightarrow \tau$. First we separate $S$ into two parts $S_1 \stackrel{\text{def}}{=} S_{|\{\overline{\alpha}\}}$ and $S_2 = S_{|dom(S)\setminus\{\overline{\alpha}\}}$ which clearly gives $S = S_1 \circ S_2$. Now we get $S\sigma_1 = \forall\overline{\alpha}.S_2C_1' \Rightarrow S_2\tau'$. For the proof of $S\sigma_1 \leq^g_{ST.C} \sigma$ we have to find a substitution $S'$ such that $S_2\tau' \leq^{S'} \tau \quad (1)$ and $ST.C.C' \;\vdash\; S'S_2C_1' \quad (2)$ . If we take $S' \stackrel{\text{def}}{=} S_1$ then $(1)$ is trivial because of $\tau = S\tau' = S_1S_2\tau'$. Further we had $VT.C_1.C_1' \downarrow ST.C.C'$ so typotheses $ST.C.C'$ is a conservative extension of typotheses $T.C_1.C_1'$ under application of $S$ so clearly $ST.C.C' \;\vdash\; S_1S_2C_1' = SC_1'$ that is $(2)$ .

Now we use invariance of bind-ability under constrained substitution. So we have a relation between non bindable constraint sets $C_1$ and $C$. Lemma 4.8 tells us that the extension $C_2''$ of termination situation $S(T.C_1) \downarrow ST.C_2''$ is non bindable in $(ST.C.C', \tau)$ so we get $C_2'' \subseteq C$. So for the constrained substitution $(S, C)$ we get $T.C_1 \leq^{(S,C)} ST.C$. Because of $ST.C \subseteq ST.C.C'$ we can assume that a subset $C_2' \subseteq C_2$ is enough to represent the constrained substitution necessary for the relation $T.C_1 \leq^{(S,C_2')} ST.C$ □

**Lemma 4.10** (non-consumption of constraints) Given a derivation $d : T.C.C' \vdash^d e : \tau$ such that $gen(T.C.C', \tau) = (C, \sigma)$ that is constraint set $C'$ is consumed in the generalization function. Then we can transform derivation $d$ so that precedent is result of generalization. Formally

$$d' : T.C.C'.C'' \vdash^d e : \tau' \;\wedge\; gen(T.C.C'.C'', \tau') = (C.C', \sigma)$$

**Proof:** We assume structure $\sigma = \forall\overline{\alpha}.C' \Rightarrow \tau$ so we have $\{\overline{\alpha}\} \cap tv(T.C) = \emptyset$. We take some fresh $\overline{\beta}$ so that $S = [\overline{\beta}/\overline{\alpha}]$ is a renaming with fresh type variables. Now we get a new derivation by applying $S$ on $d$ where we use $ST.C.C' = T.C.SC'$ so $d' : T.C.SC' \vdash^d e : S\tau$ but the result of applying the generalization function is invariant $gen(T.C.SC', S\tau) = (C, \sigma)$ because $\sigma$ is invariant under renaming of bound type variables. We define $C'' \overset{\text{def}}{=} SC' \ \wedge \ \tau' \overset{\text{def}}{=} S\tau$. Generalization did consume only $C''$ so if we extend $d'$ by $C'$ we get the result.

$$d'' : T.C.C'.C'' \vdash^d e : \tau' \ \wedge \ gen(T.C.C'.C'', \tau') = (C.C', \sigma)$$

## 4.5 The full results

The completeness and soundness results are achieved in two steps. This is done by defining a intermediate system (system $d$) between the logical system O and the algorithmical system $W$. System $d$ has the property, that for a given typing problem $(,\ , e)$ there is at most one derivation possible for judgment $, \vdash^d e : \tau$, opposed to system O where by use of $(\forall I)$ and $(\forall E)$ rules derivation of $, \ \vdash \ e : \tau$ is not unique if it is derivable. So relating of O judgments and $W$ computation results is always done via a $d$ judgment.

We illustrate that for the soundness result. In section 4.6 we prove the two lemmas :
**Lemma 4.13** (soundness of $\vdash^d$) If $, \vdash^d e : \tau$ then $, \ \vdash \ e : \tau$.
**Lemma 4.14** (soundness of $\vdash^W$) If $U, \vdash^W e : \tau$ then $U, \ \downarrow$ and $U, \vdash^d e : \tau$.
As a easy corollary we get

**Corollary 4.11** (full soundness of $wjud$) If $U, \vdash^W e : \tau$ then $U, \ \downarrow$ and $U, \ \ \vdash \ e : \tau$.

The full completeness result is achieved the same way. In 4.7 we prove the two lemmas :
**Lemma 4.15** (completeness of $\vdash^d$) If $, \ \vdash \ e : \sigma$ then $\exists C$ with $rv(C) \cap tv(,\ , \sigma) = \emptyset$ :

$$, .C \vdash^d e : \tau \ \wedge \ g(, .C, \tau) = \sigma' \ \wedge \ \sigma' \leq^g_{\Gamma.C} \sigma$$

**Lemma 4.16**(completeness of $\vdash^W$) If $UT.C \vdash^d e : \tau$ then

$$U'T.C' \vdash^W e : \tau' \ \wedge \ U' \leq^V U \ \wedge \ (U'T.C', \tau') \leq^V (UT.C, \tau)$$

We get as a corollary the full completeness result.

**Corollary 4.12** (full completeness of $\vdash^W$) If $U, \ \vdash \ e : \sigma$ then $\exists C$ with $rv(C) \cap tv(,\ , \sigma) = \emptyset$ and

$$U',\ ' \vdash^W e : \tau' \ \wedge \ U' \leq^V U \ \wedge \ U',\ ' \leq^V U,\ .C$$

so for $\sigma' \overset{\text{def}}{=} g(U',\ ', \tau')$ we have $V\sigma' \leq^g_{U\Gamma.C} \sigma$.
**Proof:** We apply Lemma 4.15 on the given derivation $U, \ \vdash \ e : \sigma$. We get the constrained set $C$ with the desired property and

$$U,\ .C \vdash^d e : \tau \ \wedge \ g(U,\ .C, \tau) = \sigma'' \ \ (1) \ \ \wedge \ \sigma'' \leq^g_{U\Gamma.C} \sigma \ \ (2)$$

We apply Lemma 4.16 on the derivation $U,\ .C \vdash^d e : \tau$ and get

$$U'T.C' \vdash^W e : \tau' \ \wedge \ U' \leq^V U \ \wedge \ (U'T.C', \tau') \leq^V (UT.C, \tau) \ \ (3)$$

So use of (1) and (3) leads to $g \circ V(U'T.C', \tau') = \sigma''$. We can use simple right extension lemma to get for $\sigma' \overset{\text{def}}{=} g(U'T.C', \tau')$ the relation $V\sigma' \leq^g_{U\Gamma.C} \sigma''$. Together with (2) and use of transitivity of the generic instance relation we get $V\sigma' \leq^g_{U\Gamma.C} \sigma$ □.

$$(TAUT)^d \qquad \frac{x : \sigma \in ,\qquad \sigma \leq^g_\Gamma \tau}{, \vdash^d x : \tau}$$

$$(\to I)^d \qquad \frac{, , u : \tau \vdash^d e : \tau'}{, \vdash^d \lambda u.e : \tau \to \tau'}$$

$$(\to E)^d \qquad \frac{, \vdash^d e : \tau' \to \tau \qquad , \vdash^d e' : \tau'}{, \vdash^d e\, e' : \tau}$$

$$(LET)^d \qquad \frac{,' \vdash^d e : \tau' \qquad gen(\tau', , ') = (\sigma, , ) \qquad , , u : \sigma \vdash^d e' : \tau}{, \vdash^d \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

Figure 6: The deterministic system

## 4.6 The soundness property

**Lemma 4.13** (soundness of $\vdash^d$) If $, \vdash^d e : \tau$ then $, \ \vdash\ e : \tau$.
**Proof:** The proof uses induction on the derivation structure of $\vdash^d$ judgments, and translates such derivations into a derivation for system O. We analyze the last applied rule of the deterministic derivation and use induction hypotheses on the premises of the rule. So there is a case to consider for every rule of the deterministic system.

$\boldsymbol{(TAUT)^d}$ last rule was

$$(TAUT)^d \ \frac{x : \sigma \in ,\qquad \sigma \leq^g_\Gamma \tau}{, \vdash^d x : \tau}$$

Definition of generic instances applied on $\sigma \leq^g_\Gamma \tau$ gives for the case $x : \sigma \in \ ,$ a proof $d : , \ \vdash\ x : \tau$

$\boldsymbol{(\to I)^d, (\to E)^d}$ The rules of system O $(\to I), (\to E)$ are syntactical equal so using induction hypotheses on the premises of the $d$-rules we get the premises of the O-rules. Application of the corresponding O-rules gives soundness.

$\boldsymbol{(LET)^d}$ last application of rule gave

$$(LET)^d \ \frac{,' \vdash^d e : \tau' \qquad gen(\tau', , ') = (\sigma, , ) \qquad , , u : \sigma \vdash^d e' : \tau}{, \vdash^d \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

Induction hypotheses on the left side gives $,' \ \vdash\ e : \tau'$ applying $gen$ gives $, \ \vdash\ e : \sigma$ (1) (see section 4.2.2). Induction applied on right side gives $, , u : \sigma \ \vdash\ e : \tau$ (2) .

So we can apply $(LET)$ rule where we use (1) for left premise and (2) for the right premise to get $\qquad , \ \vdash\ \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau$ □.

**Lemma 4.14** (soundness of $\vdash^W$) If $U, \vdash^W e : \tau$ then $U, \ \downarrow ,'$ and $,' \vdash^d e : \tau$.
prstrt

$\boldsymbol{(TAUT)^W}$ last rule was

$$(TAUT)^W \ \frac{x : \forall\overline{\alpha}.C' \Rightarrow \tau \in ,\qquad \overline{\beta}\ new}{[\overline{\beta}/\overline{\alpha}], , C' \vdash^W x : [\overline{\beta}/\overline{\alpha}]\tau}$$

$$(TAUT)^W \qquad \frac{x : \forall\overline{\alpha}.C \Rightarrow \tau \in\ , \qquad \overline{\beta}\ new}{[\overline{\beta}/\overline{\alpha}],\ ,C\vdash^W x : [\overline{\beta}/\overline{\alpha}]\tau}$$

$$(\rightarrow I)^W \qquad \frac{U,\ ,u : \alpha\vdash^W e : \tau \qquad \alpha\ new}{U,\ \vdash^W \lambda u.e : U\alpha \rightarrow \tau}$$

$$(\rightarrow E)^W \quad \frac{U,\vdash^W e : \tau \qquad U'U,\vdash^W e' : \tau' \qquad U'\tau \overset{V}{\sim}_{U'U\Gamma} \tau' \rightarrow \alpha \qquad \alpha\ new}{VU'U,\vdash^W e\,e' : V\alpha}$$

$$(LET)^W \quad \frac{U,\ '\vdash^W e : \tau' \qquad gen(\tau', U,\ ') = (\sigma, U,\ ) \qquad U'U,\ ,u : \sigma\vdash^W e' : \tau}{U'U,\ \vdash^W \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

Figure 7: Type inference algorithm W

We define $,' = [\overline{\beta}/\overline{\alpha}], .C'$ so clearly $x : \sigma \in\ ,'$ and of course $\sigma \leq^g_{\Gamma'} [\overline{\beta}/\overline{\alpha}]\tau$ because of $,\ \vdash [\overline{\beta}/\overline{\alpha}]C'$. This are the data needed to apply $(TAUT)^d$ to get $,'\vdash^d x : [\overline{\beta}/\overline{\alpha}]\tau$.

$\boldsymbol{(\rightarrow I)^W}$ Given is a derivation with last rule

$$(\rightarrow I)^W \ \frac{U,\ .u : \alpha\vdash^W e : \tau \qquad \alpha\ new}{U,\ \vdash^W \lambda u.e : U\alpha \rightarrow \tau}$$

We apply induction hypotheses on the premise to get $U(,\ .u : \alpha) \downarrow\ ,'$ and $,'\vdash^d e : \tau$. So section 4.3 gives $u : U\alpha \in\ ,'$ that is we can apply $(\rightarrow I)^d$ on $,'_u, u : U\alpha\vdash^d e : \tau$ to get $,'\vdash^d \lambda u.e : U\alpha \rightarrow \tau$ □

$\boldsymbol{(\rightarrow E)^W}$

$$(\rightarrow E)^W \ \frac{U,\vdash^W e : \tau \qquad U'U,\vdash^W e' : \tau' \qquad U'\tau \overset{V}{\sim}_{U'U\Gamma} \tau' \rightarrow \alpha \qquad \alpha\ new}{VU'U,\vdash^W e\,e' : V\alpha}$$

**left premise** We get $U,\ \downarrow\ ,_1$ and $,_1\vdash^d e : \tau$ (3)

**middle premise** We get $U'U,\ \downarrow\ ,_2$ that is $U',_1 \downarrow\ ,_2$ (4) and $,_2\vdash^d e' : \tau'$ (5) . We use (3) and (4) to get $,_2\vdash^d e : U'\tau$ (6) by using Lemma 4.4.

**right premise** Definition of the most general constrained unifier gives $VU'U,\ \downarrow\ ,_3$ that is $V,_2 \downarrow\ ,_3$ so we can extend (5) to $,_3\vdash^d e' : V\tau'$ (7) and (6) to $,_3\vdash^d e : VU'\tau$ (8) . The unifying property gives us $VU'\tau = V\tau' \rightarrow V\alpha$ so that judgment (8) becomes $,_3\vdash^d e : V\tau' \rightarrow V\alpha$ (9) .

**conclusio** Using the derived data we apply $(\rightarrow E)^d$ for $,_3 = VU'U,$ .

$$(\rightarrow E)^d \ \frac{(9), _3\vdash^d e : V\tau' \rightarrow V\alpha \qquad ,_3\vdash^d e' : V\tau'\ (7)}{,_3\vdash^d e\,e' : V\alpha}$$

$\boldsymbol{(LET)^W}$

$$(LET)^W \ \frac{U,\ '\vdash^W e : \tau' \qquad gen(\tau', U,\ ') = (\sigma, U,\ ) \qquad U'U,\ .u : \sigma\vdash^W e' : \tau}{U'U,\ \vdash^W \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

**left premise** We apply induction hypotheses to get $U, '\vdash^d e : \tau'$ (10) .

**right premise** We apply induction hypotheses to get $U'U, \vdash^d e' : \tau$ (11) .

**middle premise** Because of $U'U, \ \downarrow$ we can apply left extension Lemma 4.3 and get $(U'\sigma, U'U, \ ) = gen(U'\tau', U'U, \ ')$ (12) . Lemma Lemma 4.4 allows for $U'U, \ ' \downarrow$ application of constrained substitution $U'$ on derivation (10) so we get $U'U, '\vdash^d e : U'\tau$ (13) .

**conclusio** Now we collect the data to apply rule $(LET)^d$

$$(LET)^d \frac{(13) U'U, '\vdash^d e : U'\tau' \qquad (12) g(U'\tau', U'U, \ ') = (U'\sigma, U'U, \ ) \qquad U'U, \ .u : \sigma \vdash^d e' : \tau \ (11)}{U'U, \vdash^d \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

## 4.7 The completeness property

Here are the proofs for the completeness property of the type inference algorithm for system O. We state first verbal the result, and then second formally the induction hypotheses which is proved.

Completeness of system d means, if there is a O solution for a typing problem, then there is a d solution where the precedent uses same declarations and a new constraint set, such that the generalization of the d solution is more general than the given O solution. We can restate this as if we are given a O derivation then we can derive the needed constraints to construct a d derivation.

**Lemma 4.15** (completeness of $\vdash^d$) Given $, \ \vdash \ e : \sigma$ then $\exists C$ with $rv(C) \cap tv(, \ , \sigma) = \emptyset$ :

$$\begin{array}{ll} & , \ \vdash \ e : \sigma \rightsquigarrow \ , \ .C \vdash^d e : \tau \ \wedge \ g(, \ .C, \tau) = \sigma' \ \wedge \ \sigma' \leq^g_{\Gamma.C} \sigma \\ \wedge & , \ \vdash \ e : \tau \rightsquigarrow \ , \ .C \vdash^d e : \tau \end{array}$$

**Proof:** We assume w.l.o.g. that the given derivation of judgment $, \ \vdash \ e : \sigma$ is normalized, which is possible because all logical derivations can be normalized. The proof is done by induction on the structure of the derivation. There is a case to consider for every rule of system O, where we assume that this was the last applied rule in the derivation.

***(TAUT)***

$$(TAUT) \qquad , \ \vdash \ x : \sigma \ (x : \sigma \in \ , \ )$$

We have $x : \sigma \in \ , \ $. We assume $\sigma = \forall \overline{\alpha}.C \Rightarrow \tau$. So we clearly get $\sigma \leq^g_{\Gamma.C} \tau$ which allows to derive $, \ .C \vdash^d x : \tau$ with use of rule $(TAUT)^d$. Of course we get also the side conditions

$$g(, \ .C, \tau) = \sigma' \ \wedge \ \sigma' \leq^g_{\Gamma.C} \sigma$$

**($\forall$*I*)** Last applied rule was

$$(\forall I) \frac{, \ .\pi_\alpha \ \vdash \ e : \sigma'' \qquad (\alpha \notin \mathrm{tv}(, \ ))}{, \ \vdash \ e : \forall \alpha.\pi_\alpha \Rightarrow \sigma''}$$

We use definition $\sigma \overset{\text{def}}{=} \forall \alpha.\pi_\alpha \Rightarrow \sigma''$. We apply induction hypotheses on the premise and get the data

$$, \ .\pi_\alpha.C' \vdash^d e : \tau \ \wedge \ g(, \ .\pi_\alpha.C', \tau) = \sigma' \ \wedge \ \sigma' \leq^g_{\Gamma.\pi_\alpha.C'} \sigma''$$

We define $C \overset{\text{def}}{=} C'.\pi_\alpha$. So condition $rv(C') \cap tv(, \ .\pi_\alpha, \sigma'') = \emptyset$ gives with $\alpha \notin tv(, \ )$ that $\alpha \notin tv(, \ .C')$. The syntactical characterization of the generic instance relation gives $\sigma' \leq^g_{\Gamma.C} \forall \alpha.\pi_\alpha \Rightarrow \sigma'' = \sigma$.

**($\forall$*E*)** Last applied rule

$$(\forall E) \frac{, \ \vdash \ e : \forall \alpha.\pi_\alpha \Rightarrow \sigma'' \qquad , \ \vdash \ [\tau/\alpha]\pi_\alpha}{, \ \vdash \ e : [\tau/\alpha]\sigma''}$$

We define $\sigma \overset{\text{def}}{=} [\tau/\alpha]\sigma''$. There are two cases to consider for $\sigma$ to see which part of the induction hypotheses must be shown :

- ***σ* is a type scheme** We are given $, .C\vdash^d e : \tau$ so that for $\sigma' \overset{\text{def}}{=} g(, .C, \tau)$ we have $\sigma' \leq^g_\Gamma \forall\alpha.\pi_\alpha \Rightarrow \sigma''$. All we we have to prove is $\sigma' \leq^g_{\Gamma.C} [\tau/\alpha]\sigma'' = \sigma$ but this is the basic property of the generic instance relation.
- ***σ* is a type** We define $\tau \overset{\text{def}}{=} \sigma$. We have to construct a derivation $, .C\vdash^d e : \tau$. Because our assumption that the logical derivation is normalized we get that $e$ is not compound, it is a term variable $e = x$. So the only possible normalized derivation is to start with $(TAUT)$ on a $x : \sigma_1 \in \,,$ and applying $(\forall E)$ rules afterwards. This case is already handled in the (TAUT) case.

**$(\rightarrow I)$**

$$(\rightarrow I)\,\frac{, .u : \tau' \;\vdash\; e : \tau}{, \;\vdash\; \lambda u.e : \tau' \rightarrow \tau}$$

Induction hypotheses applied on the premise gives $, .C.u : \tau'\vdash^d e : \tau$. So we can apply rule $(\rightarrow I)^d$ and get $, .C\vdash^d \lambda u.e : \tau' \rightarrow \tau$.

**$(\rightarrow E)$**

$$(\rightarrow E)\,\frac{, \;\vdash\; e : \tau' \rightarrow \tau \qquad , \;\vdash\; e' : \tau'}{, \;\vdash\; ee' : \tau}$$

**left premise** Induction hypotheses gives $, .C_1\vdash^d e : \tau' \rightarrow \tau$.

**right premise** Induction hypotheses gives $, .C_2\vdash^d e' : \tau'$.

**conclusio** We define $C = C_1 \cup C_2$ and extend the given judgments to $T.C\vdash^d e : \tau' \rightarrow \tau$ and $T.C\vdash^d e' : \tau'$. So we can apply $(\rightarrow E)^d$ and get the result.

**$(LET)$** Last applied rule had following form

$$(LET)\,\frac{, \;\vdash\; e' : \sigma \qquad , , u : \sigma \;\vdash\; e : \tau}{, \;\vdash\; \mathsf{let}\ u = e'\ \mathsf{in}\ e : \tau}$$

**left premise** Induction hypotheses gives

$$, .C_1\vdash^d e' : \tau' \;\; (1) \;\;\wedge\;\; g(, .C_1, \tau') = \sigma' \;\;\wedge\;\; \sigma' \leq^g_{\Gamma.C_1} \sigma \;(2)$$

**right premise** Induction hypotheses gives $, .C_2.u : \sigma\vdash^d e : \tau$. We define $C = C_1 \cup C_2$ then we extend above judgment to $, .C.u : \sigma\vdash^d e : \tau$. We use (2) because of $C_1 \subseteq C$ to get $, .C.u : \sigma'\vdash^d e : \tau \quad (3)$ .

**conclusio** Now we extend also judgment (1) to use constraint set $C$ we get $, .C\vdash^d e : \tau'$. We use non-consumption Lemma 4.10 to get

$$, .C.C'\vdash^d e : \tau'' \;\; (4) \;\;\wedge\;\; gen(, .C.C', \tau'') = (, .C, \sigma') \;\; (5)$$

At least we put all the data together to apply rule $(LET)^d$.

$$(LET)^d\,\frac{(4)\;, .C.C'\vdash^d e : \tau'' \qquad gen(, .C.C', \tau'') = (, .C, \sigma')\;(5) \qquad , .C\vdash^d e' : \tau\;(3)}{, .C\vdash^d \mathsf{let}\ u = e\ \mathsf{in}\ e' : \tau}$$

**Lemma 4.16** (completeness of $\vdash^W$)

$$UT.C\vdash^d e : \tau \leadsto U'T.C'\vdash^W e : \tau' \;\wedge\; U' \leq^V U \;\wedge\; (U'T.C', \tau') \leq^V (UT.C, \tau)$$

**Proof:**

### $(TAUT)^d$

$$(TAUT)^d \frac{x : \sigma \in U''T.C \qquad \sigma \leq^g_{U''T.C} \tau}{U''T.C {\vdash}^d x : \tau}$$

We assume $\sigma = \forall\overline{\alpha}.C'' \Rightarrow \tau''$. So left premise gives a substitution $S$ with $S\tau'' = \tau$ (1) and $U''T.C \ \vdash \ SC''$ (2) . If we assume $\overline{\beta}$ fresh then we can define $\tau' \overset{\text{def}}{=} [\overline{\beta}/\overline{\alpha}]\tau''$and $C' \overset{\text{def}}{=} C.[\overline{\beta}/\overline{\alpha}]C''$ and $U' \overset{\text{def}}{=} U''$. The application of rule $(TAUT)^W$ on this data gives $U'T.C'{\vdash}^W e : \tau'$. Because of (1) and (2) we get for $V \overset{\text{def}}{=} S \circ [\overline{\beta}/\overline{\alpha}]$ and $U \overset{\text{def}}{=} V \circ U'$ clearly $U' \leq^V U$ and $(U'T.C', \tau') \leq^V (UT.C, \tau)$.

### $(\rightarrow I)^d$

$$(\rightarrow I)^d \frac{U'',\ ,u : \tau_1{\vdash}^d e : \tau_2}{U'',{\vdash}^d \lambda u.e : \tau_1 \rightarrow \tau_2}$$

For a new type variable $\alpha$ we get $U \overset{\text{def}}{=} [\tau_1/\alpha]U''$ and clearly $U,\ ,u : \alpha{\vdash}^d e : \tau_2$. Applying induction hypotheses on this gives $U',\ ',u : \alpha{\vdash}^W e : \tau_2'$ and $U' \leq^V_\Gamma U$ and $(U',\ '.u : \alpha, \tau_2') \leq^V (U,\ .u : \alpha, \tau_2)$. Applying rule $(\rightarrow I)^W$ gives $U',\ '{\vdash}^W e : U'\alpha \rightarrow \tau_2'$ and clearly $(U',\ ',U'\alpha \rightarrow \tau_2') \leq^V (U,\ ,\tau_1 \rightarrow \tau_2)$.

### $(\rightarrow E)^d$

$$(\rightarrow E)^d \frac{U,{\vdash}^d e : \tau' \rightarrow \tau \qquad U,{\vdash}^d e' : \tau'}{U,{\vdash}^d e\, e' : \tau}$$

**left premise** Induction hypotheses gives $U_1T.C_1{\vdash}^W e \ : \ \tau_1$ and $U_1 \ \leq^{V_1} \ U$ and $(U_1T.C_1, \tau_1) \leq^{V_1} (U,\ ,\tau' \rightarrow \tau)$ (3) .

**right premise** We restate left premise as $V_1U_1T.C_1{\vdash}^d e' : \tau'$. Induction hypotheses applied on this gives us $U_2U_1T.C_2{\vdash}^W e' : \tau_2$ and $U_2 \leq^{V_2} V_1$ and $(U_2U_1T.C_2, \tau_2) \leq^{V_2} (V_1U_1T.C_1, \tau') = (U,\ ,\tau')$. Now we use extend-ability of $W$ derivations by constraint sets. We define $C' \overset{\text{def}}{=} C_1.C_2$ and $,\,' \overset{\text{def}}{=} T.C'$ so we get $U_1,\ '{\vdash}^W e : \tau_1$ (4) and $U_2U_1,\ '{\vdash}^W e' : \tau_2$ (5) and of course still $U_2U_1,\ ' \leq^{V_2} U,\ $.

**conclusio** Let $\alpha$ be a new type variable and let $U'' \overset{\text{def}}{=} [\tau/\alpha]V_2$ then we get

$$U''\tau_2 \rightarrow \alpha = \tau' \rightarrow \tau = V_1\tau_1 = U''U_2\tau_1$$

So $U''$ is a unifier of $\tau_3 \rightarrow \alpha$ and $U_2\tau_1$ this implies the existence of the most general constraint unifier $\tau_2 \rightarrow \alpha \overset{U_3}{\sim}_{U_2U_1\Gamma'} U_2\tau_1$ (6) . The most general property of $U_3$ implies $U_3 \leq^{V_3} V_2$ and $(U_3U_2U_1,\ ', U_3\alpha) \leq^{V_3} (V_2U_2U_1,\ ', \tau) = (U,\ , \tau)$
All this data allow to apply $(\rightarrow E)^W$

$$(\rightarrow E)^W \frac{U_1,\ '{\vdash}^W e : \tau_1 \ (4) \qquad U_2U_1,\ '{\vdash}^W e' : \tau_2 \ (5) \qquad \tau_2 \rightarrow \alpha \overset{U_3}{\sim}_{U_2U_1\Gamma'} U_2\tau_1 \ (6) \qquad \alpha\ new}{U_3U_2U_1,\ '{\vdash}^W e\, e' : U_3\alpha}$$

Last applied rule was

### $(LET)^d$

$$(LET)^d \frac{U,\ ''{\vdash}^d e' : \tau'' \qquad gen(U,\ '', \tau'') = (U,\ ,\sigma) \qquad U,\ ,u : \sigma{\vdash}^d e : \tau}{U,{\vdash}^d \mathsf{let}\ u = e'\ \mathsf{in}\ e : \tau}$$

**left premise** Applying induction hypotheses on left premise gives following data;

$$U_1T.C_1{\vdash}^W e' : \tau_1 \ (7) \ \wedge \ U_1 \leq^{V_1} U \ \wedge \ (U_1T.C_1, \tau_1) \leq^{V_1} (U,\ '', \tau'')$$

**middle premise** We can apply the generalization function on the data above and get $gen(U_1T.C_1,\tau_1) = (U_1T.C_1',\sigma_1)$. With the data of middle premise we can apply the left extension Lemma 4.9to get for $V_1 = (S, C_2)$

$$UT.C_1' \leq^{(S,C_2')} U, \;\; (8) \;\; \wedge \; C_2' \subseteq C_2 \; \wedge \; S\sigma_1 \leq^g_{U\Gamma} \sigma \; (9)$$

We define $V_1' \overset{\text{def}}{=} (S, C_2')$.

**right premise** With use of (8) we can rephrase the right premise as $V_1'U_1T.C_1'.u : \sigma\vdash^d e : \tau$. We can substitute $\sigma$ because of (9) and get $V_1'U_1T.C_1'.u : \sigma_1\vdash^d e : \tau$. Applying induction hypotheses on this give us the data

$$\begin{array}{ll} & U_2U_1T.C_2''.u : \sigma_1\vdash^W e : \tau' \;\; (10) \;\; \wedge \; U_2 \leq^{V_2} V_1' \\ \wedge & (U_2U_1T.C_2''.u : \sigma_1,\tau') \leq^{V_2} (V_1'U_1T.C_1'.u : \sigma_1,\tau) = (U, \; .u : S\sigma_1.\tau) \; (11) \end{array}$$

**conclusio** We define $C' \overset{\text{def}}{=} C_1' \cup C_2''$ and get by extension of (10) a derivation for $U_2U_1T.C'.u : \sigma_1\vdash^W e : \tau'$ (12) . The restricted variables of $rv(C_2'')$ can't conflict with the bound type variables of $\sigma_1$ and application of function *gen* tells us that $\sigma_1$ is maximal bound so $C_2''$ is non bindable in $(U_1T.C_1',\sigma_1)$ so

$$gen(U_1T.(C_1 \cup C_2''),\tau_1) = (U_1T.(C_1' \cup C_2''),\sigma_1) = (U_1T.C',\sigma_1) \; (13)$$

By extension of (7) we can derive $U_1T.(C_1 \cup C_2)\vdash^W e' : \tau_1$ (14) Because of $V_1'U_1T.C_1' = U, \; = V_2U_2U_1T.C_2''$ we can clearly extend $V_2$ with using $V_1'$ to a new constrained substitution $V$ such that $VU_2U_1T.C' = U, .$ If we define $U' = U_2U_1$ we get by use of (11) the needed relations

$$U' \leq^V U \; \wedge \; (U'T.C',\tau') \leq^V (U, ,\tau)$$

At least we have the data to apply rule $(LET)^W$ and derive the necessary judgment in system $W$.

$$(LET)^W \quad \frac{\begin{array}{ll} (14) & U_1T.(C_1 \cup C_2'')\vdash^W e' : \tau_1 \\ (13) & gen(U_1T.(C_1 \cup C_2''),\tau_1) = (U_1T.C',\sigma_1) \\ (14) & U_2U_1T.C', u : \sigma_1\vdash^W e : \tau' \end{array}}{U'T.C'\vdash^W \mathsf{let}\; u = e' \;\mathsf{in}\; e : \tau'}$$

## 5 Conclusion

We have shown in [OWW95] that a rather modest extension to the Hindley/Milner system is enough to support both overloading and polymorphic records with a limited form of F-bounded polymorphism. The resulting system stays firmly in the tradition of ML typing, with type soundness and principal type properties completely analogous to the Hindley/Milner system.

The needed properties of the algorithm are proved here. A look at the termination for the constrained unification gives the idea, that extending the type formation rules to regular trees would extend the expressiveness without loss of the nice properties of System O.

The encoding of a polymorphic record calculus in System O indicates that there might be some deeper relationships between F-bounded polymorphism and overloading. This is also suggested by the similarities between the dictionary transform for type classes and the Penn translation for bounded polymorphism [BTCGS91]. A study of these relationships remains a topic for future work.

# References

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[Che94] Kung Chen. *A Parametric Extension of Haskell's Type Classes.* PhD thesis, Yale University, New Haven, Connecticut, December 1994. YALEU/DCS/RR-1057.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, January 1982.

[Jon92] Mark Philip Jones. *Qualified Types: Theory and Practice.* PhD thesis, Oxford University, July 1992.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 193–204, June 1992.

[LMM87] J-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. *Foundations of Deductive Databases and Logic Programming*, pages 587 – 625, 1987.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–1469, June 1995.