Programming Language Foundations

05 Polymorphic Type Inference

Prof. Dr. David Sabel
Wintersemester 2024/25

Last update: December 18, 2024

Motivation and Outline

- Why should we care about type inference?
- Type inference algorithms for KFPTS+seq for parametric polymorphic types
- Typing recursive supercombinators
  - Iterative type inference
  - Hindley-Damas-Milner type inference

Motivation

Why should we use a type system?

- for untyped programs, dynamic type errors can occur
- runtime errors are programming errors
- strong and static typing $\longrightarrow$ no type errors during runtime
- types as documentation
- types usually lead to a better program structure
- types as specification in the design phase

Motivation (Cont'd)

Minimal requirements:

- typing should be decided during compile time
- well-typed programs have no type errors during runtime

Desirable properties

- the type system does not restrict the programmer
- the compiler can compute types = type inference

Motivation (Cont'd)

Hochschule RheinMain

Not all type systems satisfy all the properties:

- Simply typed lambda calculus:
  typed language is no longer Turing-complete, since all well-typed programs converge
- Type system extensions in Haskell:
  typing / type inference is undecidable
  in some cases the compiler does not terminate!
  requires effort / precaution of the programmer

Naive Approach

Hochschule RheinMain

Naive definition:

> *A* KFPTSP+seq*-program is well-typed, if it cannot lead to a dynamic type error during runtime.*

But, this does not work well, since:

Dynamic typing in KFPTS+seq is undecidable!

Undecidability of Dynamic Typing

Hochschule RheinMain

Let `tmEncode` be a KFPTS+seq-supercombinator that simulates a universal Turing machine:

- Input: an encoding of a Turing machine M and an input w
- Output: `True`, if the TM M halts on w

`tmEncode` is programmable:

- in the lecture notes, there is a Haskell-program that performs this simulation
- the program is not dynamically untyped (since it is Haskell-typeable)
- thus we can assume `tmEncode` exists in KFPTS+seq and it is not dynamically untyped

Undecidability of Dynamic Typing (Cont'd)

Hochschule RheinMain

For TM encoding enc and input inp, let the expression s be defined as

$$\begin{array}{ll} s := & \texttt{if tmEncode}\ enc\ inp \\ & \quad \texttt{then case}_{\mathsf{Bool}}\ \texttt{Nil of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\} \\ & \quad \texttt{else case}_{\mathsf{Bool}}\ \texttt{Nil of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\} \end{array}$$

Then the following holds:

s is dynamically untyped $\iff$ the evaluation of $(\texttt{tmEncode}\ enc\ inp)$ ends with `True`

This shows:

if we can decide whether s is dynamically untyped, then we can decide the halting problem

Thus:

Proposition
The dynamic typing of KFPTS+seq-programs is undecidable.

UNIFICATION

Types

Syntax of polymorphic Types:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

where TV is a type variable, TC type constructor

- A base type is a type of the form TC, where TC is of arity 0.
- A monomorphic type is a type without type variables

Examples

- `Int`, `Bool` and `Char` are base types.
- `[Int]` und `Char` $\to$ `Int` are monomorphic types, but no base types,
- `[a]` und `a` $\to$ `a` are neither base nor monomorphic types (but polymorphic types)

Quantified Types

For polymorphic types, we use the universal quantifier::

- If τ is a polymorphic type with occurrences of type variables $\alpha_1, \ldots, \alpha_n$, then $\forall \alpha_1, \ldots, \alpha_n.\tau$ is the universally quantified type for τ
- Since the order is irrelevant, we also use $\forall \mathcal{X}.\tau$ where $\mathcal{X}$ is a set of type variables

Later:

- universally quantified types can be copied and renamed, while types without quantifiers cannot be renamed

Type Substitutions

Type substitution = a mapping $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ of a finite set of type variables to types.

Written as $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.

Formally, extension to types: σ_E mapping from types to types

$$\begin{array}{rcl}
\sigma_E(TV) & := & \sigma(TV),\ \text{if } \sigma \text{ maps } TV \\
\sigma_E(TV) & := & TV,\ \text{if } \sigma \text{ does not map } TV \\
\sigma_E(TC\ T_1\ \ldots\ T_n) & := & TC\ \sigma_E(T_1)\ \ldots\ \sigma_E(T_n) \\
\sigma_E(T_1 \to T_2) & := & \sigma_E(T_1) \to \sigma_E(T_2)
\end{array}$$

In the following, we do not distinguish between σ and its extension σ_E!

Semantics of Polymorphic Types

Hochschule RheinMain

Semantics

Type substitution σ is ground for a type τ iff

- $\sigma(X)$ is a monomorphic type for all X mapped by σ
- $\sigma(X)$ is defined for all $X \in \mathit{Vars}(\tau)$

Semantics of type τ:

$$\mathtt{sem}(\tau) := \{\sigma(\tau) \mid \sigma \text{ is a ground substitution for } \tau\}$$

This corresponds to the intuition of schematic types:

a polymorphic type describes the schema of a set of monomorphic types

Typing Rules

Hochschule RheinMain

Rule for Application:

$$\frac{s :: T_1 \to T_2, \quad t :: T_1}{(s\ t) :: T_2}$$

Problem: Guess the right instance, e.g.

```
map :: (a -> b)    -> [a] -> [b]
not :: Bool -> Bool
```

Typing of `map not`:

Before applying the rule, the type of `map` must be instantiated:

$$\sigma = \{\mathtt{a} \mapsto \mathtt{Bool}, \mathtt{b} \mapsto \mathtt{Bool}\}$$

Instead of guessing σ, σ can be computed: Using Unification

Unification

Hochschule RheinMain

Definition

- A unification problem on types is a set E of equations of the form $\tau_1 \doteq \tau_2$ where τ_1 and τ_2 are polymorphic types.
- A solution to a unification problem on types is a substitution σ (called unifier), such that $\sigma(\tau_1) = \sigma(\tau_2)$ for all equations $\tau_1 \doteq \tau_2$ of E.
- A most general solution (most general unifier, mgu) of E is a unifier σ such that for every unifier ρ of E there is a substitution γ such that $\rho(x) = \gamma \circ \sigma(x)$ for all $x \in \mathit{Vars}(E)$.

Unification Algorithm

Hochschule RheinMain

- data structure: E = multiset of equations
- let $E \cup E'$ be the disjoint union of multisets
- $E[\tau/\alpha]$ is defined as $\{s[\tau/\alpha] \doteq t[\tau/\alpha] \mid (s \doteq t) \in E\}$.

Algorithm: Apply the following inference rules until

- a fail occurs, or
- no more rule is applicable

Unification Algorithm: Inference Rules

Hochschule RheinMain

Fail-rules:

$$\textsc{Fail1}\ \frac{E \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (TC_2\ \tau'_1\ \ldots\ \tau'_m)\}}{\mathsf{Fail}}$$

if $TC_1 \neq TC_2$

$$\textsc{Fail2}\ \frac{E \cup \{(TC_1\ \tau_1\ \ldots\ \tau_n) \doteq (\tau'_1 \to \tau'_2)\}}{\mathsf{Fail}}$$

$$\textsc{Fail3}\ \frac{E \cup \{(\tau'_1 \to \tau'_2) \doteq (TC_1\ \tau_1\ \ldots\ \tau_n)\}}{\mathsf{Fail}}$$

Unification Algorithm: Inference Rules (2)

Hochschule RheinMain

Decomposition:

$$\textsc{Decompose1}\ \frac{E \cup \{TC\ \tau_1\ \ldots\ \tau_n \doteq TC\ \tau'_1\ \ldots\ \tau'_n\}}{E \cup \{\tau_1 \doteq \tau'_1, \ldots, \tau_n \doteq \tau'_n\}}$$

$$\textsc{Decompose2}\ \frac{E \cup \{\tau_1 \to \tau_2 \doteq \tau'_1 \to \tau'_2\}}{E \cup \{\tau_1 \doteq \tau'_1, \tau_2 \doteq \tau'_2\}}$$

Unification Algorithm: Inference Rules (3)

Hochschule RheinMain

Orientation and Elimination:

$$\textsc{Orient}\ \frac{E \cup \{\tau_1 \doteq \alpha\}}{E \cup \{\alpha \doteq \tau_1\}}$$

if τ_1 is not a type variable and α is a type variable

$$\textsc{Elim}\ \frac{E \cup \{\alpha \doteq \alpha\}}{E}$$

where α is a type variable

Unification Algorithm: Inference Rules (4)

Hochschule RheinMain

Solve and Occurs-Check

$$\textsc{Solve}\ \frac{E \cup \{\alpha \doteq \tau\}}{E[\tau/\alpha] \cup \{\alpha \doteq \tau\}}$$

if type variable α does not occur in τ,
but α occurs in E

$$\textsc{OccursCheck}\ \frac{E \cup \{\alpha \doteq \tau\}}{\mathsf{Fail}}$$

if $\tau \neq \alpha$ and type variable α occurs in τ

Examples

Example 1: $\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}$:

$$\textsc{Decompose2}\ \frac{\{(a \to b) \doteq \texttt{Bool} \to \texttt{Bool}\}}{\{a \doteq \texttt{Bool}, b \doteq \texttt{Bool}\}}$$

The unifier is $\{a \mapsto \texttt{Bool}, b \mapsto \texttt{Bool}\}$

Examples

Example 2: $\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}$:

$$\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}$$

$$\textsc{Decompose2}\ \frac{\{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\}}{\{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\}}$$

$$\begin{array}{rc}
\textsc{Decompose2} & \{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\} \\ \cline{2-2}
\textsc{Orient} & \{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\} \\ \cline{2-2}
 & \{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\}
\end{array}$$

$$\begin{array}{rc}
\textsc{Decompose2} & \{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\} \\ \cline{2-2}
\textsc{Orient} & \{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\} \\ \cline{2-2}
\textsc{Solve} & \{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\} \\ \cline{2-2}
 & \{[d] \doteq [a], a \doteq \texttt{Bool}, c \doteq [a]\}
\end{array}$$

$$\begin{array}{rc}
\textsc{Decompose2} & \{[d] \doteq c, a \to [a] \doteq \texttt{Bool} \to c\} \\ \cline{2-2}
\textsc{Orient} & \{[d] \doteq c, a \doteq \texttt{Bool}, [a] \doteq c\} \\ \cline{2-2}
\textsc{Solve} & \{[d] \doteq c, a \doteq \texttt{Bool}, c \doteq [a]\} \\ \cline{2-2}
\end{array}$$

Examples

Example 3: $\{a \doteq [b], b \doteq [a]\}$

$$\begin{array}{rc}
\textsc{Solve} & \{a \doteq [b], b \doteq [a]\} \\ \cline{2-2}
\textsc{OccursCheck} & \{a \doteq [[a]], b \doteq [a]\} \\ \cline{2-2}
 & \text{Fail}
\end{array}$$

Example 4: $\{a \to [b] \doteq a \to c \to d\}$

$$\begin{array}{rc}
\textsc{Decompose2} & \{a \to [b] \doteq a \to c \to d\} \\ \cline{2-2}
\textsc{Elim} & \{a \doteq a, [b] \doteq c \to d\} \\ \cline{2-2}
\textsc{Fail2} & \{[b] \doteq c \to d\} \\ \cline{2-2}
 & \text{Fail}
\end{array}$$

Properties of the Unification Algorithm

- The algorithm stops with Fail iff the input has no unifier
- The algorithm stops successfully if the input has a unifier
  The equation system E then is of the form $\{\alpha_1 \doteq \tau_1, \ldots, \alpha_n \doteq \tau_n\}$, where α_i are pairwise distinct and α_i does not occur in any τ_j.
  The unifier is $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$.
- if the algorithm returns a unifier, then it is a most general unifier
- The order of rule application is irrelevant, no branching is necessary.
  The algorithm can be implemented in a deterministic way.
- The algorithm terminates for every unification problem

Properties of the Unification Algorithm (Cont'd)

Hochschule RheinMain

- Types in the result can be of exponential size

  E.g. $\{\alpha_n \doteq \alpha_{n-1} \to \alpha_{n-1}, \alpha_{n-1} \doteq \alpha_{n-2} \to \alpha_{n-2}, \dots \alpha_1 \doteq \alpha_0 \to \alpha_0\}$
  The unifier maps α_i to a type that contains $2^i - 1$ type arrows. E.g.
  $\sigma(\alpha_1) = \alpha_0 \to \alpha_0$,
  $\sigma(\alpha_2) = (\alpha_0 \to \alpha_0) \to (\alpha_0 \to \alpha_0)$,
  $\sigma(\alpha_3) = ((\alpha_0 \to \alpha_0) \to (\alpha_0 \to \alpha_0)) \to ((\alpha_0 \to \alpha_0) \to (\alpha_0 \to \alpha_0))$
- Using sharing and an adapted Solve-rule, the unification algorithm can be implemented such that the runtime is $O(n \log n)$
  The shared representation of the result types is $O(n)$.
- The unification problem is P-complete. I.e.
  - All PTIME-problems can be presented as unification problem
  - Unification is not efficiently parallelizable.

Sketch of the Termination Proof

Hochschule RheinMain

Let E be a unification problem and

- $\mathit{Var}(E)$ = number of unsolved type variables in E
  a variable α is solved iff it occurs once in E as the left hand side of an equation (i.e. $E = E' \cup \{\alpha \doteq \tau\}$ where $\alpha \notin \mathit{Vars}(E') \cup \mathit{Vars}(\tau)$).
- $\mathit{Size}(E)$ = sum of all sizes of types on right-hand and left sides of equations in E
  the size of a type is tsize defined as: $\mathtt{tsize}(TV) = 1$,
  $\mathtt{tsize}(TC\ T_1\ \dots\ T_n) = 1 + \sum_{i=1}^{n} \mathtt{tsize}(T_i)$ and
  $\mathtt{tsize}(T_1 \to T_2) = 1 + \mathtt{tsize}(T_1) + \mathtt{tsize}(T_2)$
- $\mathit{OEq}(E)$ = number of not oriented equations in E
  an equation is oriented, if it is of the form $\alpha \doteq \tau$ where α is a type variable.
- $M(E) = (\mathit{Var}(E), \mathit{Size}(E), \mathit{OEq}(E))$, where $M(\mathrm{Fail}) := (-1, -1, -1)$.

Sketch of the Termination Proof (Cont'd)

Hochschule RheinMain

Change of the measure per rule

	$\mathit{Var}(E)$	$\mathit{Size}(E)$	$\mathit{OEq}(E)$
Fail-rules	$<$	$<$	$<$
OccursCheck	$<$	$<$	$<$
Decompose	$\leq$	$<$	
Orient	$\leq$	$=$	$<$
Elim	$\leq$	$<$	
Solve	$<$		

Thus: for each rule $\frac{E}{E'}$ we have $M(E') <_{lex} M(E)$, where $<_{lex}$ is the lexicographic order on triples.

Hochschule RheinMain

TYPING OF KFPTS+seq-EXPRESSIONS

Typing

We now consider the

polymorphic typing of KFPTS+seq-expressions

For now, we ignore the typing of supercombinators

Rule for Application with Unification

$$\frac{s :: \tau_1, \quad t :: \tau_2}{(s\ t) :: \sigma(\alpha)}$$

if σ is an mgu for $\tau_1 \doteq \tau_2 \to \alpha$ and α is a fresh type variable

Example:

$$\frac{\texttt{map} :: (a \to b) \to [a] \to [b], \quad \texttt{not} :: \texttt{Bool} \to \texttt{Bool}}{(\texttt{map}\ \texttt{not}) :: \sigma(\alpha)}$$

if σ is an mgu for $(a \to b) \to [a] \to [b] \doteq (\texttt{Bool} \to \texttt{Bool}) \to \alpha$
and α is a fresh type variable

Unification results in $\{a \mapsto \texttt{Bool}, b \mapsto \texttt{Bool}, \alpha \mapsto [\texttt{Bool}] \to [\texttt{Bool}]\}$

Thus: $\sigma(\alpha) = [\texttt{Bool}] \to [\texttt{Bool}]$

Typing with Binders

How to type an abstraction $\lambda x.s$?

- Type the body s
- Let $s :: \tau$
- Then $\lambda x.s$ has a function type $\tau_1 \to \tau$
- How corresponds τ_1 with τ?
- τ_1 is the type of x
- If x occurs in s, then we need τ_1 for typing τ!

Typing with Binders (Cont'd)

Informal rule for abstractions:

$$\frac{\text{Typing } s \text{ with assumption “} x \text{ is of type } \tau_1\text{” results in } s :: \tau}{\lambda x.s :: \tau_1 \to \tau}$$

How do we get τ_1?

Start with the most general type for x, and restrict it by the type inference

Example:

- $\lambda x.(x\ \texttt{True})$
- Typing $(x\ \texttt{True})$ starts with $x :: \alpha$
- Since x is applied, the typing has to result in $\alpha = \texttt{Bool} \to \alpha'$
- Type of the abstraction: $\lambda x.(x\ \texttt{True}) :: (\texttt{Bool} \to \alpha') \to \alpha'$.

Typing of Expressions

Hochschule RheinMain

Typing judgement:

$$\Gamma \vdash s :: \tau, E$$

Meaning:

Given a set Γ of type assumptions, for expression s the type τ and the type equations E can be derived

- Γ contains type assumptions for constructors, supercombinators, and variables
- In E type equations are collected, they will be unified later

Typing of Expressions (Cont'd)

Hochschule RheinMain

Type derivation rules are written as

$$\frac{\text{Premise(s)}}{\text{Conclusion}}$$

or more concrete:

$$\frac{\Gamma_1 \vdash s_1 :: \tau_1, E_1 \quad \ldots \quad \Gamma_k \vdash s_k :: \tau_k, E_k}{\Gamma \vdash s :: \tau, E}$$

Typing of Expressions (Cont'd)

Hochschule RheinMain

As a simplification:

for typing constructor applications $(c\ s_1\ \ldots\ s_n)$ they are treated like nested applications $(((c\ s_1)\ \ldots)\ s_n))$

Typing Rules for KFPTS+seq-Expressions (1)

Hochschule RheinMain

Axiom for variables:

$$(\text{AxV})\ \frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau, \emptyset}$$

Axiom for constructors:

$$(\text{AxC})\ \frac{}{\Gamma \cup \{c :: \forall \alpha_1 \ldots \alpha_n . \tau\} \vdash c :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}$$

where β_i are fresh type variables

- Note that each time a freshly renamed copy of the type is used!

Typing Rules for KFPTS+seq-Expressions (2)

Axiom for supercombinators (with already know type):

$$(\mathrm{AxSC})\ \frac{}{\Gamma \cup \{SC :: \forall \alpha_1 \ldots \alpha_n.\tau\} \vdash SC :: \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n], \emptyset}$$
where β_i are fresh type variables

- Note that each time a freshly renamed copy of the type is used!

Typing Rules for KFPTS+seq-Expressions (3)

Rule for applications:

$$(\mathrm{RApp})\ \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s\ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}$$
where α is a fresh type variable

Rule for `seq`:

$$(\mathrm{RSeq})\ \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (\texttt{seq}\ s\ t) :: \tau_2, E_1 \cup E_2}$$

Typing Rules for KFPTS+seq-Expressions (4)

Rule for abstractions:

$$(\mathrm{RAbs})\ \frac{\Gamma \cup \{x :: \alpha\} \vdash s :: \tau, E}{\Gamma \vdash \lambda x.s :: \alpha \to \tau, E}$$
where α is a fresh type variable

Typing Rules for KFPTS+seq-Expressions (5)

Typing of `case`: ideas

$$\begin{pmatrix} \texttt{case}_T\ s\ \texttt{of}\ \{ \\ \quad (c_1\ x_{1,1}\ \ldots\ x_{1,ar(c_1)}) \to t_1; \\ \quad \ldots; \\ \quad (c_m\ x_{m,1}\ \ldots\ x_{m,ar(c_m)}) \to t_m\} \end{pmatrix}$$

- The patterns and the expression s are of the same type. This type matches the type index T of $\texttt{case}_T$ (due to the patterns)
- The expressions $t_1, \ldots, t_n$ are of the same type. This type is the type of the case-expression

Typing Rules for KFPTS+seq-Expressions (5)

Rule for `case`:

$$\text{(RCase)}\frac{\begin{array}{l}\Gamma \vdash s :: \tau, E \\ \text{for } i = 1,\ldots,m\text{: } \Gamma \cup \{x_{i,1} :: \alpha_{i,1},\ldots,x_{i,ar(c_i)} :: \alpha_{i,ar(c_i)}\} \vdash (c_i\ x_{i,1}\ \ldots\ x_{i,ar(ci)}) :: \tau_i, E_i \\ \text{for } i = 1,\ldots,m\text{: } \Gamma \cup \{x_{i,1} :: \alpha_{i,1},\ldots,x_{i,ar(c_i)} :: \alpha_{i,ar(c_i)}\} \vdash t_i :: \tau'_i, E'_i\end{array}}{\Gamma \vdash \begin{pmatrix}\texttt{case}_T\ s\ \texttt{of}\ \{ \\ \quad (c_1\ x_{1,1}\ \ldots\ x_{1,ar(c_1)}) \to t_1; \\ \quad \ldots; \\ \quad (c_m\ x_{m,1}\ \ldots\ x_{m,ar(c_m)}) \to t_m\}\end{pmatrix} :: \alpha, E'}$$

where $E' = E \cup \bigcup_{i=1}^{m} E_i \cup \bigcup_{i=1}^{m} E'_i \cup \bigcup_{i=1}^{m} \{\tau \doteq \tau_i\} \cup \bigcup_{i=1}^{m} \{\alpha \doteq \tau'_i\}$
and $\alpha_{i,j}, \alpha$ are fresh type variables

Case-Rule for Bool

$$\text{(RCase)}\frac{\Gamma \vdash s :: \tau, E \quad \Gamma \vdash \texttt{True} :: \tau_1, E_1 \quad \Gamma \vdash \texttt{False} :: \tau_2, E_2 \quad \Gamma \vdash t_1 :: \tau'_1, E'_1 \quad \Gamma \vdash t_2 :: \tau'_2, E'_2}{\Gamma \vdash (\texttt{case}_{\texttt{Bool}}\ s\ \texttt{of}\ \{\texttt{True} \to t_1; \texttt{False} \to t_2\}) :: \alpha, E'}$$

where $E' = E \cup E_1 \cup E_2 \cup E'_1 \cup E'_2 \cup \{\tau \doteq \tau_1, \tau \doteq \tau_2\} \cup \{\alpha \doteq \tau'_1, \alpha \doteq \tau'_2\}$
and $\alpha_{i,j}, \alpha$ are fresh type variables

Case-Rule for Lists

$$\text{(RCase)}\frac{\begin{array}{l}\Gamma \vdash s :: \tau, E \\ \Gamma \vdash \texttt{Nil} :: \tau_1, E_1 \\ \Gamma \cup \{x_1 :: \alpha_1, x_2 :: \alpha_2\} \vdash \texttt{Cons}\ x_1\ x_2 :: \tau_2, E_2 \\ \Gamma \vdash t_1 :: \tau'_1, E'_1 \\ \Gamma \cup \{x_1 :: \alpha_1, x_2 :: \alpha_2\} \vdash t_2 :: \tau'_2, E'_2\end{array}}{\Gamma \vdash (\texttt{case}_{\texttt{List}}\ s\ \texttt{of}\ \{\texttt{Nil} \to t_1; (\texttt{Cons}\ x_1\ x_2) \to t_2\}) :: \alpha, E'}$$

where $E' = E \cup E_1 \cup E_2 \cup E'_1 \cup E'_2 \cup \{\tau \doteq \tau_1, \tau \doteq \tau_2\} \cup \{\alpha \doteq \tau'_1, \alpha \doteq \tau'_2\}$
and $\alpha_{i,j}, \alpha$ are fresh type variables

Algorithm: Type Inference of KFPTS+seq-Expressions

Let s be a closed KFPTS+seq-expression, where the types of all supercombinators and all constructors occurring in s are known

1. Start with assumption Γ containing the types of the constructors and supercombinators
2. Derive $\Gamma \vdash s :: \tau, E$ using the typing rules
3. Solve E with unification
4. If unification ends with Fail, then s is not typeable; otherwise let σ be an mgu of E. Then the type of s is $s :: \sigma(\tau)$.

Optimization

Hochschule RheinMain

Additional rule to unify inbetween:

$$(\text{RUnif})\ \frac{\Gamma \vdash s :: \tau, E}{\Gamma \vdash s :: \sigma(\tau), E_\sigma}$$

where E_σ is the solved equation system of E and σ is the corresponding unifier

Well-Typedness

Hochschule RheinMain

Definition

A KFPTSP+seq-expression s is well-typed iff it can be typed by given algorithm.

Example: Typing of (Cons True Nil)

Hochschule RheinMain

Start with:

Type assumption: $\Gamma_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a], \texttt{True} :: \texttt{Bool}\}$

$$(\text{RApp})\ \frac{\Gamma_0 \vdash (\texttt{Cons True}) :: \tau_1, E_1, \quad \Gamma_0 \vdash \texttt{Nil} :: \tau_2, E_2}{\Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha_4\}}$$

$$(\text{RApp})\ \frac{\Gamma_0 \vdash (\texttt{Cons True}) :: \tau_1, E_1, \quad (\text{AxC})\ \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{\Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, E_1 \cup \emptyset \cup \{\tau_1 \doteq [\alpha_3] \to \alpha_4\}}$$

$$(\text{RApp})\ \frac{(\text{RApp})\ \dfrac{\Gamma_0 \vdash \texttt{Cons} :: \tau_3, E_3, \Gamma_0 \vdash \texttt{True} :: \tau_4, E_4}{\Gamma_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\tau_3 \doteq \tau_4 \to \alpha_2\} \cup E_3 \cup E_4}, \quad (\text{AxC})\ \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{\Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\tau_3 \doteq \tau_4 \to \alpha_2\} \cup E_3 \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

$$(\text{RApp})\ \frac{(\text{RApp})\ \dfrac{(\text{AxC})\ \dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset}, \Gamma_0 \vdash \texttt{True} :: \tau_4, E_4}{\Gamma_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \tau_4 \to \alpha_2\} \cup E_4}, \quad (\text{AxC})\ \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{\Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \tau_4 \to \alpha_2\} \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

$$(\text{RApp})\ \frac{(\text{RApp})\ \dfrac{\Gamma_0 \vdash \texttt{Cons} :: \alpha_1 \to [\alpha_1] \to [\alpha_1], \emptyset, \quad \Gamma_0 \vdash \texttt{True} :: \texttt{Bool}, \emptyset}{\Gamma_0 \vdash (\texttt{Cons True}) :: \alpha_2, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2\}}, \quad (\text{AxC})\ \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_3], \emptyset}}{\Gamma_0 \vdash (\texttt{Cons True Nil}) :: \alpha_4, \{\alpha_1 \to [\alpha_1] \to [\alpha_1] \doteq \texttt{Bool} \to \alpha_2\} \cup \{\alpha_2 \doteq [\alpha_3] \to \alpha_4\}}$$

Example: Typing $\lambda x.x$

Hochschule RheinMain

Start with: Type assumption: $\Gamma_0 = \emptyset$

$$(\text{RAbs})\ \frac{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \tau, E}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \to \tau, E}$$

$$(\text{RAbs})\ \frac{(\text{AxV})\ \dfrac{}{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset}}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \to \alpha, \emptyset}$$

Nothing to unify, thus $(\lambda x.x) :: \alpha \to \alpha$

Example: Typing of Ω

Typing of $(\lambda x.(x\ x))\ (\lambda y.(y\ y))$

Start with: Type assumption: $\Gamma_0 = \emptyset$

$$\text{(RApp)}\ \frac{\emptyset \vdash (\lambda x.(x\ x)) :: \tau_1, E_1, \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha_1\}}$$

$$\text{(RApp)}\ \frac{\text{(RAbs)}\ \dfrac{\{x :: \alpha_2\} \vdash (x\ x) :: \tau_1, E_1}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \tau_1, E_1}\ , \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha_1\}}$$

$$\text{(RApp)}\ \frac{\text{(RAbs)}\ \dfrac{\text{(RApp)}\ \dfrac{\{x :: \alpha_2\} \vdash x :: \tau_3, E_3, \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4}}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_3, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4}\ , \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\tau_3 \doteq \tau_4 \to \alpha_3\} \cup E_3 \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \to \alpha_1\}}$$

$$\text{(RApp)}\ \frac{\text{(AxV)}\ \dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}\ , \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\alpha_2 \doteq \tau_4 \to \alpha_3\} \cup E_4}$$

$$\text{(RApp)}\ \frac{}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \to \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \to \alpha_1\}}$$

$$\text{(RApp)}\ \frac{\text{(RAbs)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxV)}\ \dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}\ , \text{(AxV)}\ \dfrac{}{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset}\ ,}{\{x :: \alpha_2\} \vdash (x\ x) :: \alpha_3, \{\alpha_2 \doteq \alpha_2 \to \alpha_3\}}}{\emptyset \vdash (\lambda x.(x\ x)) :: \alpha_2 \to \alpha_3, \{\alpha_2 \doteq \alpha_2 \to \alpha_3\}}\ , \emptyset \vdash (\lambda y.(y\ y)) :: \tau_2, E_2}{\emptyset \vdash (\lambda x.(x\ x))\ (\lambda y.(y\ y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \to \alpha_3\} \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \to \alpha_1\}}$$

Example: Expression with Supercombinators (1)

Assumption: `map` and `length` are already typed.
We type:

$$t := \lambda xs.\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \mathtt{Nil}; (\mathtt{Cons}\ y\ ys) \to \mathtt{map}\ \mathtt{length}\ ys\}$$

We use the start assumption:

$$\begin{aligned}\Gamma_0 = \{&\mathtt{map} :: \forall a,b.(a \to b) \to [a] \to [b],\\ &\mathtt{length} :: \forall a.[a] \to \mathtt{Int},\\ &\mathtt{Nil} :: \forall a.[a]\\ &\mathtt{Cons} :: \forall a.a \to [a] \to [a]\\ \}&\end{aligned}$$

Example: Expression with Supercombinators (2)

Derivation tree:

$$\text{(RAbs)}\ \frac{\text{(RCase)}\ \dfrac{\text{(AxV)}\ \dfrac{}{B_3}\ , \text{(AxC)}\ \dfrac{}{B_4}\ , \text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{B_8}\ , \text{(AxV)}\ \dfrac{}{B_9}}{B_6}\ , \text{(AxV)}\ \dfrac{}{B_7}}{B_5}\ , \text{(AxC)}\ \dfrac{}{B_{10}}\ , \text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxSC)}\ \dfrac{}{B_{14}}\ , \text{(AxSC)}\ \dfrac{}{B_{15}}}{B_{12}}\ , \text{(AxV)}\ \dfrac{}{B_{13}}}{B_{11}}}{B_2}}{B_1}$$

Labels:

$$\begin{aligned}B_1 = \quad &\Gamma_0 \vdash t :: \alpha_1 \to \alpha_{13},\\ &\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\ &(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \mathtt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\\ &\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}\\ B_2 = \quad &\Gamma_0 \cup \{xs :: \alpha_1\} \vdash\\ &\quad \mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \mathtt{Nil}; (\mathtt{Cons}\ y\ ys) \to \mathtt{map}\ \mathtt{length}\ ys\} :: \alpha_{13},\\ &\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7,\\ &(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \mathtt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12},\end{aligned}$$

Example: Expression with Supercombinators (3)

$$\text{(RAbs)}\ \frac{\text{(RCase)}\ \dfrac{\text{(AxV)}\ \dfrac{}{B_3}\ , \text{(AxC)}\ \dfrac{}{B_4}\ , \text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{B_8}\ , \text{(AxV)}\ \dfrac{}{B_9}}{B_6}\ , \text{(AxV)}\ \dfrac{}{B_7}}{B_5}\ , \text{(AxC)}\ \dfrac{}{B_{10}}\ , \text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxSC)}\ \dfrac{}{B_{14}}\ , \text{(AxSC)}\ \dfrac{}{B_{15}}}{B_{12}}\ , \text{(AxV)}\ \dfrac{}{B_{13}}}{B_{11}}}{B_2}}{B_1}$$

Labels:

$$\begin{aligned}B_3 = \quad &\Gamma_0 \cup \{xs :: \alpha_1\} \vdash\ xs :: \alpha_1, \emptyset\\ B_4 = \quad &\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \mathtt{Nil} :: [\alpha_2], \emptyset\\ B_5 = \quad &\Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\mathtt{Cons}\ y\ ys) :: \alpha_7,\\ &\quad \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7\}\\ B_6 = \quad &\Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\mathtt{Cons}\ y) :: \alpha_6,\\ &\quad \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6\}\\ B_7 = \quad &\Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset\\ B_8 = \quad &\Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \mathtt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset\\ B_9 = \quad &\Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash y :: \alpha_3, \emptyset\\ B_{10} = \quad &\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \mathtt{Nil} :: [\alpha_{14}], \emptyset\end{aligned}$$

Example: Expression with Supercombinators (4)

$$\text{(RAbs)}\dfrac{\text{(RCase)}\dfrac{\text{(AxV)}\dfrac{}{B_3},\ \text{(AxC)}\dfrac{}{B_4},\ \text{(RApp)}\dfrac{\text{(RApp)}\dfrac{\text{(AxC)}\dfrac{}{B_8},\ \text{(AxV)}\dfrac{}{B_9}}{B_6},\ \text{(AxV)}\dfrac{}{B_7}}{B_5},\ \text{(AxC)}\dfrac{}{B_{10}},\ \text{(RApp)}\dfrac{\text{(RApp)}\dfrac{\text{(AxSC)}\dfrac{}{B_{14}},\ \text{(AxSC)}\dfrac{}{B_{15}}}{B_{12}},\ \text{(AxV)}\dfrac{}{B_{13}}}{B_{11}}}{B_2}}{B_1}$$

Labels:

$$\begin{array}{rl}
B_{11} = & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map length})\ ys :: \alpha_{12}, \\
& \quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}\} \\
B_{12} = & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\texttt{map length}) :: \alpha_{11}, \\
& \quad \{(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}\} \\
B_{13} = & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset \\
B_{14} = & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{map} :: (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9], \emptyset \\
B_{15} = & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \texttt{length} :: [\alpha_{10}] \to \texttt{Int}, \emptyset
\end{array}$$

Example: Expression with Supercombinators (5)

Labels:

$$\begin{array}{rl}
B_1 = & \Gamma_0 \vdash t :: \alpha_1 \to \alpha_{13}, \\
& \{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7, \\
& (\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}, \\
& \alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}, \}
\end{array}$$

Solve using unification:

$$\begin{array}{l}
\{\alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_3 \to \alpha_6, \alpha_6 \doteq \alpha_4 \to \alpha_7, \\
(\alpha_8 \to \alpha_9) \to [\alpha_8] \to [\alpha_9] \doteq ([\alpha_{10}] \to \texttt{Int}) \to \alpha_{11}, \alpha_{11} \doteq \alpha_4 \to \alpha_{12}, \\
\alpha_1 \doteq [\alpha_2], \alpha_1 = \alpha_7, \alpha_{13} \doteq [\alpha_{14}], \alpha_{13} = \alpha_{12}\}
\end{array}$$

Results in:

$$\begin{array}{rcl}
\sigma & = & \{\alpha_1 \mapsto [[\alpha_{10}]], \alpha_2 \mapsto [\alpha_{10}], \alpha_3 \mapsto [\alpha_{10}], \alpha_4 \mapsto [[\alpha_{10}]], \alpha_5 \mapsto [\alpha_{10}], \\
& & \alpha_6 \mapsto [[\alpha_{10}]] \to [[\alpha_{10}]], \alpha_7 \mapsto [[\alpha_{10}]], \alpha_8 \mapsto [\alpha_{10}], \alpha_9 \mapsto \texttt{Int}, \\
& & \alpha_{11} \mapsto [[\alpha_{10}]] \to [\texttt{Int}], \alpha_{12} \mapsto [\texttt{Int}], \alpha_{13} \mapsto [\texttt{Int}], \alpha_{14} \mapsto \texttt{Int}\}
\end{array}$$

Thus $t :: \sigma(\alpha_1 \to \alpha_{13}) = [[\alpha_{10}]] \to [\texttt{Int}]$.

Example: Typing of Lambda-Bound Variables (1)

`const` is defined as

```
const :: a -> b -> a
const x y = x
```

Typing of $\lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'})$

Type assumption:

$\Gamma_0 = \{\texttt{const} :: \forall a, b.a \to b \to a, \texttt{True} :: \texttt{Bool}, \texttt{'A'} :: \texttt{Char}\}$.

Example: Typing of Lambda-Bound Variables (2)

$$\text{(RAbs)}\dfrac{\text{(RApp)}\dfrac{\text{(RApp)}\dfrac{\text{(AxSC)}\dfrac{}{\Gamma_1 \vdash \texttt{const} :: \alpha_2 \to \alpha_3 \to \alpha_2, \emptyset},\ \text{(RApp)}\dfrac{\text{(AxV)}\dfrac{}{\Gamma_1 \vdash x :: \alpha_1},\ \text{(AxC)}\dfrac{}{\Gamma_1 \vdash \texttt{True} :: \texttt{Bool}}}{\Gamma_1 \vdash (x\ \texttt{True}) :: \alpha_4, E_1}}{\Gamma_1 \vdash \texttt{const}\ (x\ \texttt{True}) :: \alpha_5, E_2},\ \text{(RApp)}\dfrac{\text{(AxV)}\dfrac{}{\Gamma_1 \vdash x :: \alpha_1},\ \text{(AxC)}\dfrac{}{\Gamma_1 \vdash \texttt{'A'} :: \texttt{Char}}}{\Gamma_1 \vdash (x\ \texttt{'A'}) :: \alpha_6, E_3}}{\Gamma_1 \vdash \texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_7, E_4}}{\Gamma_0 \vdash \lambda x.\texttt{const}\ (x\ \texttt{True})\ (x\ \texttt{'A'}) :: \alpha_1 \to \alpha_7, E_4}$$

where $\Gamma_1 = \Gamma_0 \cup \{x :: \alpha_1\}$ and:

$$\begin{array}{rcl}
E_1 & = & \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4\} \\
E_2 & = & \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5\} \\
E_3 & = & \{\alpha_1 \doteq \texttt{Char} \to \alpha_6\} \\
E_4 & = & \{\alpha_1 \doteq \texttt{Bool} \to \alpha_4, \alpha_2 \to \alpha_3 \to \alpha_2 \doteq \alpha_4 \to \alpha_5, \alpha_1 \doteq \texttt{Char} \to \alpha_6, \\
& & \alpha_5 \doteq \alpha_6 \to \alpha_7\}
\end{array}$$

Unification fails, since $\texttt{Char} \neq \texttt{Bool}$

Example: Typing of Lambda-Bound Variables (3)

In Haskell-interpreter:

```
Main> \x -> const (x True)  (x 'A')

<interactive>:1:23:
Couldn't match expected type 'Char' against inferred type 'Bool'
      Expected type: Char -> b
      Inferred type: Bool -> a
 In the second argument of 'const', namely '(x 'A')'
 In the expression: const (x True) (x 'A')
```

- Example shows: Lambda-bound variables are monomorphically typed!
- The same applies to variables bound by `case`-patterns
- Hence, one speaks of `let`-polymorphism, since only `let`-bound variables are typed polymorphically.
- In KFPTS+seq, there is no `let`, but supercombinators which are similar to `let`

TYPING SUPERCOMBINATORS

Recursive Supercombinators

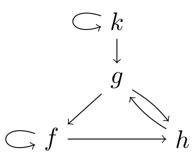
Definition

Let $\mathcal{SC}$ be a set of supercombinators, $SC_i, SC_j \in \mathcal{SC}$

- $SC_i \preceq SC_j$ iff the rhs of the definition of SC_j uses the supercombinator SC_i.
- $\preceq^+$ is the transitive closure of $\preceq$ (and $\preceq^*$ is the reflexive-transitive closure)
- SC_i is directly recursive iff $SC_i \preceq SC_i$ and recursive iff $SC_i \preceq^+ SC_i$
- $SC_1, \ldots, SC_m$ are mutually recursive if $SC_i \preceq^+ SC_j$ for all $i, j \in \{1, \ldots, m\}$.

Example

```
f x y = if x≤1 then y else f (x-y) (y + h x)
g x   = if x==0 then (f 1 x) + (h 2) else 10
h x   = if x==1 then 0 else g (x-1)
k x y = if x==1 then y else k (x-1) (y+(g x))
```



f and k are directly recursive, f, g, h are mutually recursive, f, g, h, k are recursive

Typing Non-Recursive Supercombinators

- Non-recursive Supercombinators can be typed like abstractions
- Notation: $\Gamma \vdash_T SC :: \tau$ means:
  With assumption Γ, SC can be typed with type τ

Typing rule for (closed) non-recursive supercombinators:

$$(\text{RSC1})\ \frac{\Gamma \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \forall \mathcal{X}.\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

if σ is the solution of E,

$SC\ x_1\ \ldots\ x_n = s$ is the definition of SC

and SC is non-recursive,

and $\mathcal{X} = \mathit{Vars}(\sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau))$

Example: Typing of `(.)`

```
(.) f g x = f (g x)
```

Γ_0 is empty, since no constructors or supercombinators occur

$$\text{(RSC1)}\dfrac{\text{(RApp)}\dfrac{\text{(AxV)}\dfrac{}{\Gamma_1 \vdash f :: \alpha_1, \emptyset} \;,\; \text{(RApp)}\dfrac{\text{(AxV)}\dfrac{}{\Gamma_1 \vdash g :: \alpha_2, \emptyset} \;,\; \text{(AxV)}\dfrac{}{\Gamma_1 \vdash x :: \alpha_3, \emptyset}}{\Gamma_1 \vdash (g\ x) :: \alpha_5, \{\alpha_2 \doteq \alpha_3 \to \alpha_5\}}}{\Gamma_1 \vdash (f\ (g\ x)) :: \alpha_4, \{\alpha_2 \doteq \alpha_3 \to \alpha_5, \alpha_1 = \alpha_5 \to \alpha_4\}}}{\emptyset \vdash_T (.) :: \forall \mathcal{X}.\sigma(\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4)}$$

where $\Gamma_1 = \{f :: \alpha_1, g :: \alpha_2, x :: \alpha_3\}$

Unification results in $\sigma = \{\alpha_2 \mapsto \alpha_3 \to \alpha_5, \alpha_1 \mapsto \alpha_5 \to \alpha_4\}$.
Thus: $\sigma(\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4) = (\alpha_5 \to \alpha_4) \to (\alpha_3 \to \alpha_5) \to \alpha_3 \to \alpha_4$

Now $\mathcal{X} = \{\alpha_3, \alpha_4, \alpha_5\}$ and we may rename this to:

$$(.) :: \forall a, b, c.(a \to b) \to (c \to a) \to c \to b$$

Typing of Recursive Supercombinators

- Assume $SC\ x_1\ \dots\ x_n = e$ and SC occurs in e (SC is recursive)
- What is the problem when typing SC?
- To type the body e, the type of SC must be known!

ITERATIVE TYPE INFERENCE ALGORITHM

Idea of the Iterative Type Inference

- Start with the most general type for SC (i.e. a type variable)
- Type the body using this assumption
- This results in a newly derived type for SC
- Continue (iterate) with this type
- Stop if new type = old type:
  Then we found a consistent type assumption

Most general type: Type T, such that $\mathrm{sem}(T) = \{$all monomorphic types$\}$.

The type α satisfies this (as quantified type $\forall \alpha.\alpha$)

Iterative Type Inference

Rule to compute new assumptions:

$$(\text{SCRec})\ \frac{\Gamma \cup \{x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \sigma(\alpha_1 \to \ldots \alpha_n \to \tau)}$$

if $SC\ x_1\ \ldots\ x_n = s$ is the definition of SC, σ the solution of E

The same as RSC1, but Γ has to contain an assumption for SC

Iterative Typing: Preparatory Work

Because of mutual recursion:

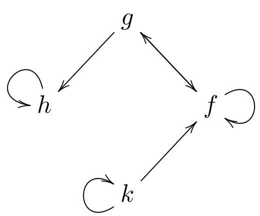
- Dependency analysis of the supercombinators
- Compute the strongly connected components in the call graph
- Let $\simeq$ be the equivalence relation of $\preceq^*$. The strongly connected components are the equivalence classes of $\simeq$
- Each equivalence class is typed together

The order of the typing is according to $\preceq^*$ modulo $\simeq$.

Example


```
f x y = if x<=1 then y else f (x-y) (y + g x)
g x   = if x==0 then (f 1 x) + (h 2) else 10
h x   = if x==1 then 0 else h (x-1)
k x y = if x==1 then y else k (x-1) (y+(f x y))
```

The call graph is:



The equivalence classes (ordered) are $\{h\} \preceq^+ \{f, g\} \preceq^+ \{k\}$.

Iterative Type Inference Algorithm

Iterative Type Inference Algorithm

Input: Mutually recursive supercombinators $SC_1, \ldots, SC_m$

1. Start assumption Γ contains types of the constructors and the already typed SCs
2. $\Gamma_0 := \Gamma \cup \{SC_1 :: \forall\alpha_1.\alpha_1, \ldots, SC_m :: \forall\alpha_m.\alpha_m\}$ and $j = 0$.
3. For each SC_i $(i = 1, \ldots, m)$ apply rule (SCRec) for Γ_j, to infer the type of SC_i.
4. If the m type derivations are successful (for all i: $\Gamma_j \vdash_T SC_i :: \tau_i$)
   Then quantify: $SC_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SC_m :: \forall\mathcal{X}_m.\tau_m$
   Set $\Gamma_{j+1} := \Gamma \cup \{SC_1 :: \forall\mathcal{X}_1.\tau_1, \ldots, SC_m :: \forall\mathcal{X}_m.\tau_m\}$
5. If $\Gamma_j \neq \Gamma_{j+1}$, then set $j := j + 1$ and go to step (3).
   Otherwise, $\Gamma_j = \Gamma_{j+1}$, and thus Γ_j is **consistent**.

Output: quantified polymorphic types of the SC_i of the consistent type assumption.
If a single unification fails, then $SC_1, \ldots, SC_m$ are not typeable.

Properties of the Algorithm

- The computed types are unique up to renaming for each iteration and thus: if the algorithm terminates, then the types of the supercombinators are unique.
- In each step: newly computed types are more specific or remain the same (computation is monotonic w.r.t. `sem`: "$\mathtt{sem}(T_{j+1}) \subseteq \mathtt{sem}(T_j)$")
- If the algorithm does not terminate, then no polymorphic type for the supercombinators exists (since computation is monotonic w.r.t. `sem` and starts with the largest set)
- The algorithm computes the greatest fixpoint w.r.t. `sem`: Suppose that F is the operator that performs one iteration of the algorithm on the set of monomorphic types. If the algorithm stops with set S, then $F(S) = S$ (so S is a fixpoint) and S is the largest set M such that $F(M) = M$.
- This shows, that the iterative type inference algorithm computes the most general polymorphic type (w.r.t. `sem`)

Example: length (1)

$$\mathtt{length}\ xs = \mathtt{case_{List}}\ xs\ \mathtt{of}\{\mathtt{Nil} \to 0; (y : ys) \to 1 + \mathtt{length}\ ys\}$$

Assumption:

$\Gamma = \{\mathtt{Nil} :: \forall a.[a], (:) :: \forall a.a \to [a] \to [a], 0, 1 :: \mathtt{Int}, (+) :: \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}\}$

1.Iteration: $\Gamma_0 = \Gamma \cup \{\mathtt{length} :: \forall \alpha.\alpha\}$

$$\text{(SCRec)}\ \dfrac{\text{(RCase)}\ \dfrac{\begin{array}{ll}(a) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1\\ (b) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \mathtt{Nil} :: \tau_2, E_2\\ (c) & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3\\ (d) & \Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4\\ (e) & \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}\} \vdash (1 + \mathtt{length}\ ys) :: \tau_5, E_5\end{array}}{\begin{array}{c}\Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\mathtt{case_{List}}\ xs\ \mathtt{of}\{\mathtt{Nil} \to 0; (y : ys) \to 1 + \mathtt{length}\ xs\}) :: \alpha_3,\\ E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\}\end{array}}}{\Gamma_0 \vdash_T \mathtt{length} :: \sigma(\alpha_1 \to \alpha_3)}$$

where σ is the solution of

$$E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \dot{=} \tau_2, \tau_1 \dot{=} \tau_3, \alpha_3 \dot{=} \tau_4, \alpha_3 \dot{=} \tau_5\}$$

Example: length (2)

(a): $\text{(AxV)}\ \dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset}$

I.e. $\tau_1 = \alpha_1$ and $E_1 = \emptyset$

(b): $\text{(AxC)}\ \dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash \mathtt{Nil} :: [\alpha_6], \emptyset}$

I.e. $\tau_2 = [\alpha_6]$ and $E_2 = \emptyset$

(c) $$\text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{\Gamma'_0 \vdash (:) :: \alpha_9 \to [\alpha_9] \to [\alpha_9], \emptyset}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma'_0 \vdash y :: \alpha_4, \emptyset}}{\Gamma'_0 \vdash ((:)\ y) :: \alpha_8, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \dot{=} \alpha_4 \to \alpha_8\}}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma'_0 \vdash ys :: \alpha_5, \emptyset}}{\Gamma'_0 \vdash (y : ys) :: \alpha_7, \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \dot{=} \alpha_4 \to \alpha_8, \alpha_8 \dot{=} \alpha_5 \to \alpha_7\}}$$

where $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

I.e.,. $\tau_3 = \alpha_7$ and $E_3 = \{\alpha_9 \to [\alpha_9] \to [\alpha_9] \dot{=} \alpha_4 \to \alpha_8, \alpha_8 \dot{=} \alpha_5 \to \alpha_7\}$

Example: length (3)

(d) $\text{(AxC)}\ \dfrac{}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \mathtt{Int}, \emptyset}$

I.e. $\tau_4 = \mathtt{Int}$ und $E_4 = \emptyset$

(e) $$\text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{\Gamma'_0 \vdash (+) :: \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}, \emptyset}\ ,\ \text{(AxC)}\ \dfrac{}{\Gamma'_0 \vdash 1 :: \mathtt{Int}, \emptyset}}{\Gamma'_0 \vdash ((+)\ 1) :: \alpha_{11}, \{\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \dot{=} \mathtt{Int} \to \alpha_{11}\}}\ ,\ \text{(RApp)}\ \dfrac{\text{(AxSC)}\ \dfrac{}{\Gamma'_0 \vdash \mathtt{length} :: \alpha_{13}, \emptyset}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma'_0 \vdash (ys) :: \alpha_5, \emptyset}}{\Gamma'_0 \vdash (\mathtt{length}\ ys) :: \alpha_{12}, \{\alpha_{13} \dot{=} \alpha_5 \to \alpha_{12}\}}}{\Gamma'_0 \vdash (1 + \mathtt{length}\ ys) :: \alpha_{10}, \{\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \dot{=} \mathtt{Int} \to \alpha_{11}, \alpha_{13} \dot{=} \alpha_5 \to \alpha_{12}, \alpha_{11} \dot{=} \alpha_{12} \to \alpha_{10}\}}$$

where $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$

I.e., $\tau_5 = \alpha_{10}$ and

$E_5 = \{\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int} \dot{=} \mathtt{Int} \to \alpha_{11}, \alpha_{13} \dot{=} \alpha_5 \to \alpha_{12}, \alpha_{11} \dot{=} \alpha_{12} \to \alpha_{10}\}$

Example: length (4)

In summary: $\Gamma_0 \vdash_T \texttt{length} :: \sigma(\alpha_1 \to \alpha_3)$
where σ is the solution of

$$\begin{aligned}&\{\alpha_9 \to [\alpha_9] \to [\alpha_9] \doteq \alpha_4 \to \alpha_8, \alpha_8 \doteq \alpha_5 \to \alpha_7,\\ &\texttt{Int} \to \texttt{Int} \to \texttt{Int} \doteq \texttt{Int} \to \alpha_{11}, \alpha_{13} \doteq \alpha_5 \to \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \to \alpha_{10},\\ &\alpha_1 \doteq [\alpha_6], \alpha_1 \doteq \alpha_7, \alpha_3 \doteq \texttt{Int}, \alpha_3 \doteq \alpha_{10}\}\end{aligned}$$

Unification results in the unifier:

$$\begin{aligned}&\{\alpha_1 \mapsto [\alpha_9], \alpha_3 \mapsto \texttt{Int}, \alpha_4 \mapsto \alpha_9, \alpha_5 \mapsto [\alpha_9], \alpha_6 \mapsto \alpha_9, \alpha_7 \mapsto [\alpha_9], \alpha_8 \mapsto [\alpha_9] \to [\alpha_9],\\ &\alpha_{10} \mapsto \texttt{Int}, \alpha_{11} \mapsto \texttt{Int} \to \texttt{Int}, \alpha_{12} \mapsto \texttt{Int}, \alpha_{13} \mapsto [\alpha_9] \to \texttt{Int}\}\end{aligned}$$

thus $\sigma(\alpha_1 \to \alpha_3) = [\alpha_9] \to \texttt{Int}$

$\Gamma_1 = \Gamma \cup \{\texttt{length} :: \forall\alpha.[\alpha] \to \texttt{Int}\}$

Since $\Gamma_0 \neq \Gamma_1$ another iteration is required.
2. iteration: It results in the same type, hence Γ_1 is consistent.

Iterative Typing is More General than Haskell

Example

`g x = 1 : (g (g 'c'))`

$\Gamma = \{1 :: \texttt{Int}, \texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$
$\Gamma_0 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha\}$ (and $\Gamma_0' = \Gamma_0 \cup \{x :: \alpha_1\}$):

$$\scriptstyle(\text{SCRec})\dfrac{\scriptstyle(\text{RApp})\dfrac{\scriptstyle(\text{RApp})\dfrac{(\text{AxC})\dfrac{}{\Gamma_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}, \quad (\text{AxC})\dfrac{}{\Gamma_0' \vdash 1 :: \texttt{Int}, \emptyset}}{\Gamma_0' \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}, \quad (\text{RApp})\dfrac{(\text{AxSC})\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_6, \emptyset}, \quad (\text{RApp})\dfrac{(\text{AxSC})\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_8, \emptyset}, \quad (\text{AxC})\dfrac{}{\Gamma_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset},}{\Gamma_0' \vdash (\texttt{g 'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}}{\Gamma_0' \vdash (\texttt{g (g 'c')}) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}}{\Gamma_0' \vdash \texttt{Cons 1 (g (g 'c'))} :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}}{\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_1 \to [\texttt{Int}]}$$

where $\sigma = \{\alpha_2 \mapsto [\texttt{Int}], \alpha_3 \mapsto [\texttt{Int}] \to [\texttt{Int}], \alpha_4 \mapsto [\texttt{Int}], \alpha_5 \mapsto \texttt{Int}, \alpha_6 \mapsto \alpha_7 \to [\texttt{Int}], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ is the solution of
$\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

I.e. $\Gamma_1 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha \to [\texttt{Int}]\}$.

The next iteration shows that Γ_1 is consistent.

Iterative Typing is More General than Haskell (Cont'd)

Haskell cannot infer a type for g:

```
Prelude> let g x = 1:(g(g 'c'))

<interactive>:1:13:
Couldn't match expected type '[t]' against inferred type 'Char'
      Expected type: Char -> [t]
      Inferred type: Char -> Char
    In the second argument of '(:)', namely '(g (g 'c'))'
    In the expression: 1 : (g (g 'c'))
```

But: Haskell can check the type if it is given:

```
let g::a -> [Int]; g x = 1:(g(g 'c'))
Prelude> :t g
g :: a -> [Int]
```

Reason: If the type is present, Haskell performs type checking and no type inference.
Then g is treated like an already typed supercombinator.

Example: Multiple Iterations are Required (1)

`g x = x : (g (g 'c'))`

- $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$.
- $\Gamma_0 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha\}$

$$\scriptstyle(\text{SCRec})\dfrac{\scriptstyle(\text{RApp})\dfrac{\scriptstyle(\text{RApp})\dfrac{(\text{AxC})\dfrac{}{\Gamma_0' \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}, \quad (\text{AxV})\dfrac{}{\Gamma_0' \vdash x :: \alpha_1, \emptyset}}{\Gamma_0' \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}, \quad (\text{RApp})\dfrac{(\text{AxSC})\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_6, \emptyset}, \quad (\text{RApp})\dfrac{(\text{AxSC})\dfrac{}{\Gamma_0' \vdash \texttt{g} :: \alpha_8, \emptyset}, \quad (\text{AxC})\dfrac{}{\Gamma_0' \vdash \texttt{'c'} :: \texttt{Char}, \emptyset},}{\Gamma_0' \vdash (\texttt{g 'c'}) :: \alpha_7, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7\}}}{\Gamma_0' \vdash (\texttt{g (g 'c')}) :: \alpha_4, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4\}}}{\Gamma_0' \vdash \texttt{Cons}\ x\ \texttt{(g (g 'c'))} :: \alpha_2, \{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}}{\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1 \to \alpha_2) = \alpha_5 \to [\alpha_5]}$$

where $\sigma = \{\alpha_1 \mapsto \alpha_5, \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \to [\alpha_5], \alpha_8 \mapsto \texttt{Char} \to \alpha_7\}$ is the solution of
$\{\alpha_8 \doteq \texttt{Char} \to \alpha_7, \alpha_6 \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

I.e. $\Gamma_1 = \Gamma \cup \{\texttt{g} :: \forall\alpha.\alpha \to [\alpha]\}$.

Example: Multiple Iterations are Required (2)

Hochschule RheinMain

Since $\Gamma_0 \neq \Gamma_1$ another iteration is required.
Let $\Gamma_1' = \Gamma_1 \cup \{x :: \alpha_1\}$:

$$
{\scriptstyle(\textsc{SCRec})}\dfrac{
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxC})}\dfrac{}{\Gamma_1' \vdash \mathtt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}
\;,\;
{\scriptstyle(\textsc{AxV})}\dfrac{}{\Gamma_1' \vdash x :: \alpha_1, \emptyset}
}{\Gamma_1' \vdash (\mathtt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}
\;,\;
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxSC})}\dfrac{}{\Gamma_1' \vdash \mathtt{g} :: \alpha_6 \to [\alpha_6], \emptyset}
\;,\;
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxSC})}\dfrac{}{\Gamma_1' \vdash \mathtt{g} :: \alpha_8 \to [\alpha_8], \emptyset}
\;,\;
{\scriptstyle(\textsc{AxC})}\dfrac{}{\Gamma_1' \vdash \mathtt{'c'} :: \mathtt{Char}, \emptyset}\;,
}{\Gamma_1' \vdash (\mathtt{g}\ \mathtt{'c'}) :: \alpha_7, \{\alpha_8 \to [\alpha_8] \doteq \mathtt{Char} \to \alpha_7\}}
}{\Gamma_1' \vdash (\mathtt{g}\ (\mathtt{g}\ \mathtt{'c'})) :: \alpha_4, \{\alpha_8 \to [\alpha_8] \doteq \mathtt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4\}}
}{\Gamma_1' \vdash \mathtt{Cons}\ x\ (\mathtt{g}\ (\mathtt{g}\ \mathtt{'c'})) :: \alpha_2, \{\alpha_8 \to [\alpha_8] \doteq \mathtt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}
}{\Gamma_1 \vdash_T \mathtt{g} :: \sigma(\alpha_1 \to \alpha_2) = [\mathtt{Char}] \to [[\mathtt{Char}]]}
$$

where $\sigma = \{\alpha_1 \mapsto [\mathtt{Char}], \alpha_2 \mapsto [[\mathtt{Char}]], \alpha_3 \mapsto [[\mathtt{Char}]] \to [[\mathtt{Char}]], \alpha_4 \mapsto [[\mathtt{Char}]], \alpha_5 \mapsto [\mathtt{Char}], \alpha_6 \mapsto [\mathtt{Char}], \alpha_7 \mapsto [\mathtt{Char}], \alpha_8 \mapsto \mathtt{Char}\}$
is the solution of $\{\alpha_8 \to [\alpha_8] \doteq \mathtt{Char} \to \alpha_7, \alpha_6 \to [\alpha_6] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

Hence $\Gamma_2 = \Gamma \cup \{\mathtt{g} :: [\mathtt{Char}] \to [[\mathtt{Char}]]\}$.

Example: Multiple Iterations are Required (3)

Hochschule RheinMain

Since $\Gamma_1 \neq \Gamma_2$ another iteration is required:
Let $\Gamma_2' = \Gamma_2 \cup \{x :: \alpha_1\}$:

$$
{\scriptstyle(\textsc{SCRec})}\dfrac{
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxC})}\dfrac{}{\Gamma_2' \vdash \mathtt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}
\;,\;
{\scriptstyle(\textsc{AxV})}\dfrac{}{\Gamma_2' \vdash x :: \alpha_1, \emptyset}
}{\Gamma_2' \vdash (\mathtt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3}
\;,\;
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxSC})}\dfrac{}{\Gamma_2' \vdash \mathtt{g} :: [\mathtt{Char}] \to [[\mathtt{Char}]], \emptyset}
\;,\;
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxSC})}\dfrac{}{\Gamma_2' \vdash \mathtt{g} :: [\mathtt{Char}] \to [[\mathtt{Char}]], \emptyset}
\;,\;
{\scriptstyle(\textsc{AxC})}\dfrac{}{\Gamma_2' \vdash \mathtt{'c'} :: \mathtt{Char}, \emptyset}\;,
}{\Gamma_2' \vdash (\mathtt{g}\ \mathtt{'c'}) :: \alpha_7, \{[\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \mathtt{Char} \to \alpha_7\}}
}{\Gamma_2' \vdash (\mathtt{g}\ (\mathtt{g}\ \mathtt{'c'})) :: \alpha_4, \{[\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \mathtt{Char} \to \alpha_7, [\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \alpha_7 \to \alpha_4\}}
}{\Gamma_2' \vdash \mathtt{Cons}\ x\ (\mathtt{g}\ (\mathtt{g}\ \mathtt{'c'})) :: \alpha_2, \{[\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \mathtt{Char} \to \alpha_7, [\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}
}{\Gamma_2 \vdash_T \mathtt{g} :: \sigma(\alpha_1 \to \alpha_2)}
$$

where σ is the solution of
$\{[\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \mathtt{Char} \to \alpha_7, [\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha_1 \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}$

Unification:

$$
\dfrac{\dfrac{[\mathtt{Char}] \to [[\mathtt{Char}]] \doteq \mathtt{Char} \to \alpha_7, \\ \ldots}{[\mathtt{Char}] \doteq \mathtt{Char}, \\ [[\mathtt{Char}]] \doteq \alpha_7, \\ \ldots}}{\mathit{Fail}}
$$

g is not typeable.

The Example Shows ...

Hochschule RheinMain

Proposition

The iterative type inference algorithm sometimes requires multiple iterations until a result (untyped / consistent assumption) is found.

Note: There are examples where multiple iterations are required to find a consistent type assumption.

Non-Termination of the Iterative Typing (1)

Hochschule RheinMain

```
f = [g]
g = [f]
```

Since $\mathtt{f} \simeq \mathtt{g}$, the iterative typing types f and g together.
$\Gamma = \{\mathtt{Cons} :: \forall a.a \to [a] \to [a], \mathtt{Nil} : \forall a.a\}$.
$\Gamma_0 = \Gamma \cup \{\mathtt{f} :: \forall \alpha.\alpha, \mathtt{g} :: \forall \alpha.\alpha\}$

$$
{\scriptstyle(\textsc{SCRec})}\dfrac{
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{RApp})}\dfrac{
{\scriptstyle(\textsc{AxC})}\dfrac{}{\Gamma_0 \vdash \mathtt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}
\;,\;
{\scriptstyle(\textsc{AxSC})}\dfrac{}{\Gamma_0 \vdash \mathtt{g} :: \alpha_5}
}{\Gamma_0 \vdash (\mathtt{Cons}\ \mathtt{g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}
\;,\;
{\scriptstyle(\textsc{AxC})}\dfrac{}{\Gamma_0 \vdash \mathtt{Nil} :: [\alpha_2], \emptyset}
}{\Gamma_0 \vdash [\mathtt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}
}{\Gamma_0 \vdash_T \mathtt{f} :: \sigma(\alpha_1) = [\alpha_5]}
$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\}$ is
the solution of $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Non-Termination of the Iterative Typing (2)

Hochschule RheinMain

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxC}) \;,\; \dfrac{}{\Gamma_0 \vdash \texttt{f} :: \alpha_5}\,(\text{AxSC})}{\Gamma_0 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3\}}\,(\text{RApp}) \;,\; \dfrac{}{\Gamma_0 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxC})}{\Gamma_0 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_0 \vdash_T \texttt{g} :: \sigma(\alpha_1) = [\alpha_5]}\,(\text{SCRec})$$

$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \to [\alpha_5], \alpha_4 \mapsto \alpha_5\}$ is the solution of $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq \alpha_5 \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Hence, $\Gamma_1 = \Gamma \cup \{\texttt{f} :: \forall a.[a], \texttt{g} :: \forall a.[a]\}$. Since $\Gamma_1 \neq \Gamma_0$, another iteration is required.

Non-Termination of the Iterative Typing (3)

Hochschule RheinMain

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxC}) \;,\; \dfrac{}{\Gamma_1 \vdash \texttt{g} :: [\alpha_5]}\,(\text{AxSC})}{\Gamma_1 \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}\,(\text{RApp}) \;,\; \dfrac{}{\Gamma_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxC})}{\Gamma_1 \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_1 \vdash_T \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]]}\,(\text{SCRec})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ is the solution of $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxC}) \;,\; \dfrac{}{\Gamma_1 \vdash \texttt{f} :: [\alpha_5]}\,(\text{AxSC})}{\Gamma_1 \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3\}}\,(\text{RApp}) \;,\; \dfrac{}{\Gamma_1 \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxC})}{\Gamma_1 \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_1 \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]]}\,(\text{SCRec})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \to [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\}$ is the solution of $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5] \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

Hence $\Gamma_2 = \Gamma \cup \{\texttt{f} :: \forall a.[[a]], \texttt{g} :: \forall a.[[a]]\}$. Since $\Gamma_2 \neq \Gamma_1$, another iteration is required.

Non-Termination of the Iterative Typing (4)

Hochschule RheinMain

Conjecture: The iterative typing does not terminate

Proof (by induction): iteration i: $\Gamma_i = \Gamma \cup \{\texttt{f} :: \forall a.[a]^i, \texttt{g} :: \forall a.[a]^i\}$ where $[a]^i$ i-fold nested list

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxC}) \;,\; \dfrac{}{\Gamma_i \vdash \texttt{g} :: [\alpha_5]^i}\,(\text{AxSC})}{\Gamma_i \vdash (\texttt{Cons g}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}\,(\text{RApp}) \;,\; \dfrac{}{\Gamma_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxC})}{\Gamma_i \vdash [\texttt{g}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_i \vdash_T \texttt{f} :: \sigma(\alpha_1) = [[\alpha_5]^i]}\,(\text{SCRec})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ is the solution of $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_i \vdash \texttt{Cons} :: \alpha_4 \to [\alpha_4] \to [\alpha_4], \emptyset}\,(\text{AxC}) \;,\; \dfrac{}{\Gamma_i \vdash \texttt{f} :: [\alpha_5]^i}\,(\text{AxSC})}{\Gamma_i \vdash (\texttt{Cons f}) :: \alpha_3, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3\}}\,(\text{RApp}) \;,\; \dfrac{}{\Gamma_i \vdash \texttt{Nil} :: [\alpha_2], \emptyset}\,(\text{AxC})}{\Gamma_i \vdash [\texttt{f}] :: \alpha_1, \{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}}\,(\text{RApp})}{\Gamma_i \vdash_T \texttt{g} :: \sigma(\alpha_1) = [[\alpha_5]^i]}\,(\text{SCRec})$$

$\sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \to [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\}$ is the solution of $\{\alpha_4 \to [\alpha_4] \to [\alpha_4] \doteq [\alpha_5]^i \to \alpha_3, \alpha_3 \doteq [\alpha_2] \to \alpha_1\}$

I.e. $\Gamma_{i+1} = \Gamma \cup \{\texttt{f} :: \forall a.[a]^{i+1}, \texttt{g} :: \forall a.[a]^{i+1}\}$.

Thus ...

Hochschule RheinMain

Proposition

The iterative type inference algorithm may not terminate.

Moreover, the following holds (the proof can be found in the literature)

Theorem

Iterative typing is undecidable.

This follows from the undecidability of so-called semi unification of first-order terms. (works of Kfoury, Tiuryn, and Urzyczyn and Henglein)

Call Hierachy

- The iterative typing does not need the information of the call hierarchy:
  The same types are inferred independently in which order they are computed

Type Safety

A typed program calculus fulfills type safety iff

- Typing is preserved by reduction (type preservation):
  For monomorphic type τ: If $t :: \tau$ and $t \to t'$, then $t' :: \tau$
  This includes the case that a polymorphic type becomes more general.
- Typed, closed expressions are reducible if they are not a WHNF (well-typed programs don't get stuck) (progress lemma)

Type Safety (2)

Lemma

Let s be a directly dynamically untyped KFPTS+seq-expression. Then the iterative typing cannot type s.

Proof. Assume s is directly dynamically untyped:

- $s = R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$ and c is not of type T.
  iterative typing adds equations ensuring the types of $(c\ s_1\ \ldots\ s_n)$ and of the patterns in $Alts$ are equal. Since c is not of type T, unification fails.
- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$: iterative typing add ensuring the type of $\lambda x.t$ is equal to the type of the patterns in $Alts$, and that it is a function type. Unification fails, since the patterns do not have a function type.
- $R[(c\ s_1\ \ldots\ s_{ar(c)})\ t]$: $((c\ s_1\ \ldots\ s_{ar(c)})\ t)$ is typed as a nested application $(((c\ s_1)\ \ldots)\ s_{ar(c)})\ t)$. Equations are added implying that c can receive at most $ar(c)$ arguments. Since there is one more argument, unification will fail.

Type Safety (3)

Lemma (Type Preservation)

Let s be a well-typed and closed KFPTSP+seq-expression (of a well-typed KFPTSP+seq-program) and $s \xrightarrow{name} s'$. Then s' is well-typed.

Proof (Sketch): Inspect the (β)-, $(SC-\beta)$- and (case)-reduction and the typing of the expressions before and after the reduction.

Type Safety (4)

Hochschule RheinMain

The two lemmas show:

Proposition

Let s be a well-typed, closed KFPTSP+seq-expression. Then s is not dynamically untyped.

Progress Lemma

Let s be a well-typed, closed KFPTSP+seq-expression. Then

- s is a WHNF, or
- s is call-by-name-reducible, i.e. $s \xrightarrow{name} s'$ for some s'.

Proof. A closed KFPTS+seq-expression s is irreducible iff s is a WHNF or s is directly dynamically untyped (and thus not well-typed).

Type Safety (5)

Hochschule RheinMain

Theorem

Type safety holds for the iterative typing of KFPTSP+seq.

Forcing Termination of Type Inference

Hochschule RheinMain

- Let $SC_1, \ldots, SC_m$ be mutually recursive supercombinators
- Let $\Gamma_i \vdash_T SC_1 :: \tau_1, \ldots, \Gamma_i \vdash_T SC_m :: \tau_m$ be the types derived in the i^{th} iteration

Milner-Step: Type $SC_1, \ldots, SC_m$ together with the type assumption:
$\Gamma_M = \Gamma \cup \{SC_1 :: \tau_1, \ldots, SC_m :: \tau_m\}$; without quantifiers
and the following rule (SCRecM) ...

Forcing Termination (Cont'd)

Hochschule RheinMain

$$(\text{SCRecM})\ \frac{\text{for } i = 1, \ldots, m\text{: } \Gamma_M \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i', E_i}{\Gamma_M \vdash_T \text{for } i = 1, \ldots, m\ SC_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i')}$$

if σ is the solution of $E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\tau_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i'\}$

and
$$\begin{array}{lcl} SC_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = & s_1 \\ \ldots & & \\ SC_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = & s_m \end{array}$$
are the definitions of $SC_1, \ldots, SC_m$

As additional typing rule we add:

$$(\text{AxSC2})\ \frac{}{\Gamma \cup \{SC :: \tau\} \vdash SC :: \tau}$$
if τ is not universally quantified

Forcing Termination (Cont'd)

Differences to an iterative step:

- Types of to-be-typed SCs are not quantified
- No copies of these types are made
- At the end, the assumed types are unified with the derived types

This ensures: the new type assumption derived by (SCRecM) is always consistent

After a Milner-step the iterative algorithm terminates.

HINDLEY-DAMAS-MILNER-TYPING

The Hindley-Damas-Milner Typing

The algorithm is similar to iterative typing, with the differences:

- Only one iteration step is performed
- The type assumption assumes for each to-be-typed supercombinator SC_i the type α_i (without quantifier!)
- consistency is enfored by additional unification equations

Haskell uses Hindley-Damas-Milner-typing

The Hindley-Damas-Milner Type Inference Algorithm

$SC_1, \ldots, SC_m$ are mutually recursive supercombinators ofan equivalence class w.r.t. $\simeq$ supercombinators strictly less than $SC_1, \ldots, SC_m$ w.r.t. $\preceq$ are already typed

1. Assumption Γ contains types of the already typed SCs and of the constructors (all universally quantified)
2. Type $SC_1, \ldots, SC_m$ with the rule (MSCRec):

$$(\mathrm{MSCRec})\ \frac{\text{for } i = 1, \ldots, m\text{: } \Gamma \cup \{SC_1 :: \beta_1, \ldots, SC_m :: \beta_m\} \cup \{x_{i,1} :: \alpha_{i,1}, \ldots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau_i, E_i}{\Gamma \vdash_T \text{for } i = 1, \ldots, m\ SC_i :: \sigma(\alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i)}$$

$$\text{if } \sigma \text{ solution of } E_1 \cup \ldots \cup E_m \cup \bigcup_{i=1}^{m} \{\beta_i \doteq \alpha_{i,1} \to \ldots \to \alpha_{i,n_i} \to \tau_i\}$$

$$\text{and } \begin{array}{l} SC_1\ x_{1,1}\ \ldots\ x_{1,n_1} = s_1 \\ \ldots \\ SC_m\ x_{m,1}\ \ldots\ x_{m,n_m} = s_m \end{array} \text{ are the definitions of } SC_1, \ldots, SC_m$$

If unification fails, then $SC_1, \ldots, SC_m$ are not Hindley-Damas-Milner typeable

The Hindley-Damas-Milner Type Inference Algorithm

Hochschule RheinMain

Simplification: Rule for one single recursive supercombinator:

$$(\text{MSCRec1})\ \frac{\Gamma \cup \{SC :: \beta, x_1 :: \alpha_1, \ldots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \sigma(\alpha_1 \to \ldots \to \alpha_n \to \tau)}$$

if σ is the solution of $E \cup \{\beta \doteq \alpha_1 \to \ldots \to \alpha_n \to \tau\}$
and $SC\ x_1\ \ldots\ x_n = s$ is the definition of SC

Properties of the Hindley-Damas-Milner Typing

Hochschule RheinMain

- the algorithm terminates
- the algorithm computes unique types
- Hindley-Damas-Milner typing is decidable
- the decision problem whether an expression is Hindley-Damas-Milner-typeable is DEXPTIME-complete
- the types may be more restrictive than the iterative type, in particular, an expression may be iteratively typeable but not Hindley-Damas-Milner-typeable.
- The Hindley-Damas-Milner algorithm needs knowledge of the call hierarchy of the SCs:
  It may return more restrictive types if the typing is not along the hierarchy

Example

Hochschule RheinMain

Sometimes exponentially many type variables are required:

```
(let x0 = \z->z in
  (let x1 = (x0,x0) in
    (let x2 = (x1,x1) ins
      (let x3 = (x2,x2) in
         (let x4 = (x3,x3) in
            (let x5 = (x4,x4) in
               (let x6 = (x5,x5) in x6)))))))
```

Requires 2^6 type variables, the generalized example requires 2^n.

Example: map

Hochschule RheinMain

```
map f xs = case xs of {
             [] → []
             (y:ys) → (f y):(map f ys)
           }
```

$\Gamma_0 = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{Nil} :: \forall a.[a]\}$

Sei $\Gamma = \Gamma_0 \cup \{\texttt{map} :: \beta, f :: \alpha_1, xs :: \alpha_2\}$ and $\Gamma' = \Gamma \cup \{y : \alpha_3, ys :: \alpha_4\}$.

$$(\text{MSCRec1})\ \frac{(\text{RCase})\ \dfrac{\begin{array}{ll}(a) & \Gamma \vdash xs :: \tau_1, E_1\\ (b) & \Gamma \vdash \texttt{Nil} :: \tau_2, E_2\\ (c) & \Gamma' \vdash (\texttt{Cons}\ y\ ys) :: \tau_3, E_3\\ (d) & \Gamma \vdash \texttt{Nil} :: \tau_4, E_4\\ (e) & \Gamma' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \tau_5, E_5\end{array}}{\Gamma \vdash \texttt{case}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \texttt{Cons}\ y\ ys \to \texttt{Cons}\ y\ (\texttt{map}\ f\ ys)\} :: \alpha, E}}{\Gamma \vdash_T \texttt{map} :: \sigma(\alpha_1 \to \alpha_2 \to \alpha)}$$

if σ is the solution of $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$

where $E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha \doteq \tau_4, \alpha \doteq \tau_5\}$.

Example: map (2)

Hochschule RheinMain

(a) $\text{(AxV)}\ \dfrac{}{\Gamma \vdash xs :: \alpha_2, \emptyset}$

I.e. $\tau_1 = \alpha_2$ and $E_1 = \emptyset$.

(b) $\text{(AxC)}\ \dfrac{}{\Gamma \vdash \texttt{Nil} :: [\alpha_5], \emptyset}$

I.e. $\tau_2 = [\alpha_5]$ and $E_2 = \emptyset$

(c)
$$\text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_6 \to [\alpha_6] \to [\alpha_6]}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash y :: \alpha_3, \emptyset}}{\Gamma' \vdash (\texttt{Cons}\ y) :: \alpha_7, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7\}}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{\Gamma' \vdash (\texttt{Cons}\ y\ ys) :: \alpha_8, \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}}$$

I.e. $\tau_3 = \alpha_8$ and $E_3 = \{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8\}$

(d) $\text{(AxC)}\ \dfrac{}{\Gamma \vdash \texttt{Nil} :: [\alpha_9], \emptyset}$

I.e. $\tau_4 = [\alpha_9]$ and $E_4 = \emptyset$.

Example: map (3)

Hochschule RheinMain

(e)

$$\text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{\Gamma' \vdash \texttt{Cons} :: \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}], \emptyset}\ ,\ \text{(RApp)}\ \dfrac{\text{(AxV)}\ \dfrac{}{\Gamma' \vdash f :: \alpha_1, \emptyset}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash y :: \alpha_3, \emptyset}}{\Gamma' \vdash (f\ y) :: \alpha_{15}, \{\alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}}{\Gamma' \vdash (\texttt{Cons}\ (f\ y)) :: \alpha_{11}, \{\alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}\}}\ ,\ \text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxSC2)}\ \dfrac{}{\Gamma' \vdash \texttt{map} :: \beta, \emptyset}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash f :: \alpha_1, \emptyset}\ ,}{\Gamma' \vdash (\texttt{map}\ f) :: \alpha_{12}, \{\beta \doteq \alpha_1 \to \alpha_{12}\}}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{\Gamma' \vdash (\texttt{map}\ f\ ys) :: \alpha_{13}, \{\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}}}{\begin{array}{c}\Gamma' \vdash (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)) :: \alpha_{14}, \\ \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}\end{array}}$$

I.e. $\tau_5 = \alpha_{14}$ and

$E_5 = \{\alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15},$
$\beta \doteq \alpha_1 \to \alpha_{12}, \alpha_{12} \doteq \alpha_4 \to \alpha_{13}\}$

Example: map (4)

Hochschule RheinMain

Unifiy equations $E \cup \{\beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}$:

$$\begin{array}{l}\{\alpha_6 \to [\alpha_6] \to [\alpha_6] \doteq \alpha_3 \to \alpha_7, \alpha_7 \doteq \alpha_4 \to \alpha_8, \alpha_{11} \doteq \alpha_{13} \to \alpha_{14}, \\ \ \alpha_{10} \to [\alpha_{10}] \to [\alpha_{10}] \doteq \alpha_{15} \to \alpha_{11}, \alpha_1 \doteq \alpha_3 \to \alpha_{15}, \beta \doteq \alpha_1 \to \alpha_{12}, \\ \ \alpha_{12} \doteq \alpha_4 \to \alpha_{13}, \alpha_2 \doteq [\alpha_5], \alpha_2 \doteq \alpha_8, \alpha \doteq \alpha_9, \alpha \doteq \alpha_{14}, \\ \ \beta \doteq \alpha_1 \to \alpha_2 \to \alpha\}\end{array}$$

Unification results in

$$\begin{array}{l}\sigma = \{\alpha \mapsto [\alpha_{10}], \alpha_1 \mapsto \alpha_6 \to \alpha_{10}, \alpha_2 \mapsto [\alpha_6], \alpha_3 \mapsto \alpha_6, \alpha_4 \mapsto [\alpha_6], \alpha_5 \mapsto \alpha_6, \\ \quad \alpha_7 \mapsto [\alpha_6] \to [\alpha_6], \alpha_8 \mapsto [\alpha_6], \alpha_9 \mapsto [\alpha_{10}], \alpha_{11} \mapsto [\alpha_{10}] \to [\alpha_{10}], \\ \quad \alpha_{12} \mapsto [\alpha_6] \to [\alpha_{10}], \alpha_{13} \mapsto [\alpha_{10}], \alpha_{14} \mapsto [\alpha_{10}], \alpha_{15} \mapsto \alpha_{10}, \\ \quad \beta \mapsto (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}],\end{array}$$

I.e. $map :: \sigma(\alpha_1 \to \alpha_2 \to \alpha) = (\alpha_6 \to \alpha_{10}) \to [\alpha_6] \to [\alpha_{10}]$.

Examples Known from Iterative Typing

Hochschule RheinMain

```
g x = x : (g (g 'c'))
```

Iterative typing results in Fail (after multiple iterations)
Hindley-Damas-Milner: $\Gamma = \{\texttt{Cons} :: \forall a.a \to [a] \to [a], \texttt{'c'} :: \texttt{Char}\}$.
Let $\Gamma' = \Gamma \cup \{x :: \alpha, \mathsf{g} :: \beta\}$.

$$\text{(MSCRec)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(RApp)}\ \dfrac{\text{(AxC)}\ \dfrac{}{\Gamma \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\ ,\ \text{(AxV)}\ \dfrac{}{\Gamma \vdash x :: \alpha, \emptyset}}{\Gamma \vdash (\texttt{Cons}\ x) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3}\ ,\ \text{(RApp)}\ \dfrac{\text{(AxSC2)}\ \dfrac{}{\Gamma \vdash \mathsf{g} :: \beta, \emptyset}\ ,\ \text{(RApp)}\ \dfrac{\text{(AxSC2)}\ \dfrac{}{\Gamma \vdash \mathsf{g} :: \beta, \emptyset}\ ,\ \text{(AxC)}\ \dfrac{}{\Gamma \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\ ,}{\Gamma \vdash (\mathsf{g}\ \texttt{'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}}{\Gamma \vdash (\mathsf{g}\ (\mathsf{g}\ \texttt{'c'})) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}}{\Gamma \vdash \texttt{Cons}\ x\ (\mathsf{g}\ (\mathsf{g}\ \texttt{'c'})) :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4 \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}}{\Gamma \vdash_T \mathsf{g} :: \sigma(\alpha \to \alpha_2)}$$

where σ is the solution of

$$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \alpha \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$$

Unification fails, since `Char` should be made equal to a list. Thus, g ist not Hindley-Damas-Milner-typeable.

Examples Known from Iterative Typing (2)


```
g x = 1 : (g (g 'c'))
```

Iterative type: $g :: \forall\alpha.\alpha \to [\texttt{Int}]$
Hindley-Damas-Milner: Let $\Gamma' = \Gamma \cup \{x :: \alpha, \mathsf{g} :: \beta\}$.

$$\text{(SCRec)}\dfrac{\text{(RApp)}\dfrac{\text{(RApp)}\dfrac{\text{(AxC)}\dfrac{}{\Gamma \vdash \texttt{Cons} :: \alpha_5 \to [\alpha_5] \to [\alpha_5], \emptyset}\ ,\ \text{(AxC)}\dfrac{}{\Gamma \vdash 1 :: \texttt{Int}, \emptyset}}{\Gamma \vdash (\texttt{Cons}\ 1) :: \alpha_3, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3}\ ,\ \text{(RApp)}\dfrac{\text{(AxSC2)}\dfrac{}{\Gamma \vdash \mathsf{g} :: \beta, \emptyset}\ ,\ \text{(RApp)}\dfrac{\text{(AxSC2)}\dfrac{}{\Gamma \vdash \mathsf{g} :: \beta, \emptyset}\ ,\ \text{(AxC)}\dfrac{}{\Gamma \vdash \texttt{'c'} :: \texttt{Char}, \emptyset}\ ,}{\Gamma \vdash (\texttt{g 'c'}) :: \alpha_7, \{\beta \doteq \texttt{Char} \to \alpha_7\}}}{\Gamma \vdash (\texttt{g (g 'c')}) :: \alpha_4, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4\}}}{\Gamma \vdash \texttt{Cons 1 (g (g 'c'))} :: \alpha_2, \{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2\}}}{\Gamma \vdash_T \mathsf{g} :: \sigma(\alpha \to \alpha_2)}$$

where σ is the solution of

$$\{\beta \doteq \texttt{Char} \to \alpha_7, \beta \doteq \alpha_7 \to \alpha_4, \alpha_5 \to [\alpha_5] \to [\alpha_5] \doteq \texttt{Int} \to \alpha_3, \alpha_3 \doteq \alpha_4 \to \alpha_2, \beta \doteq \alpha \to \alpha_2\}$$

Unification fails since $[\alpha_5] \doteq \texttt{Char}$ should be unified.

Iterative Typing May Return More General Types


```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Types of the constructors
$\texttt{Empty} :: \forall a.\ \texttt{Tree}\ a$ and
$\texttt{Node} :: \forall a.\ a \to \texttt{Tree a} \to \texttt{Tree a} \to \texttt{Tree a}$

```
g x y = Node True (g x y) (g y x)
```

Hindley-Damas-Milner: $\mathsf{g} :: a \to a \to \texttt{Tree Bool}$
Iterative Typing:: $\mathsf{g} :: a \to b \to \texttt{Tree Bool}$
Reason:
Iterative typing uses copies of the type of g,

Hindley-Damas-Milner Typing and Type Safety

- Hindley-Damas-Milner typed programs are always iteratively typeable
- Hence Hindley-Damas-Milner typed programs are never dynamically untyped
- Also the progress lemma holds: Hindley-Damas-Milner typed (closed) programs are WHNFs or reducible

Hindley-Damas-Milner Typing and Type Safety (2)

- Type-Preservation: Does hold in KFPTSP+seq, but not in Hskell:

  ```
  let x = (let y = \u -> z in (y [], y True, seq x True))
      z = const z x
  in x
  ```

  is Hindley-Damas-Milner typeable

  After a so-called $(llet)$-reduction:

  ```
  let x = (y [], y True, seq x True)
      y = \u -> z
      z = const z x
  in x
  ```

  This expression is not Hindley-Damas-Milner-typeable (but iteratively)
- Reason: After the reduction x,y,z have to be typed together, before they can be typed separately

Conclusion: Type Safety

Not a real problem, since

- Type-Preservation holds for the iterative typing.
- well-typed programs are dynamically typed
- Hindley-Damas-Milner-typeable implies iterative typeable
- reduction preserve the iterative type