

## 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 → 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)



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



Naive definition:

*A KFPTS+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



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)



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

$$s := \text{if } \text{tmEncode } enc \text{ } inp \\ \text{then case}_{\text{Bool}} \text{ Nil of } \{ \text{True} \rightarrow \text{True}; \text{False} \rightarrow \text{False} \} \\ \text{else case}_{\text{Bool}} \text{ Nil of } \{ \text{True} \rightarrow \text{True}; \text{False} \rightarrow \text{False} \}$$

Then the following holds:

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

This shows:

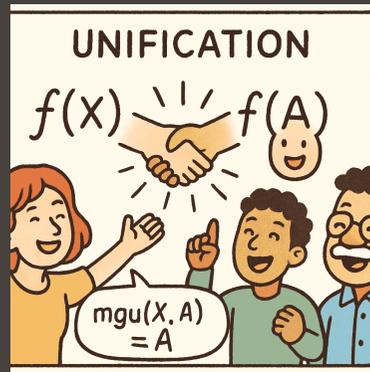
if we can decide whether  $s$  is dynamically untyped, then we can decide the halting problem

Thus:

### Proposition

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

## UNIFICATION



## Types

Syntax of **polymorphic Types**:

$$\mathbf{T} ::= TV \mid TC \mathbf{T}_1 \dots \mathbf{T}_n \mid \mathbf{T}_1 \rightarrow \mathbf{T}_2$$

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 → Int` are monomorphic types, but no base types,
- `[a]` und `a → a` are neither base nor monomorphic types (but polymorphic types)

## Quantified Types

For polymorphic types, we use the **universal quantifier**:

- If  $\tau$  is a polymorphic type with occurrences of type variables  $\alpha_1, \dots, \alpha_n$ , then  $\forall \alpha_1, \dots, \alpha_n. \tau$  is the **universally quantified type** for  $\tau$
- 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  $\{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$  of a finite set of type variables to types.

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

Formally, extension to types:  $\sigma_E$  mapping from types to types

$$\begin{aligned} \sigma_E(TV) &:= \sigma(TV), \text{ if } \sigma \text{ maps } TV \\ \sigma_E(TV) &:= TV, \text{ if } \sigma \text{ does not map } TV \\ \sigma_E(TC \mathbf{T}_1 \dots \mathbf{T}_n) &:= TC \sigma_E(\mathbf{T}_1) \dots \sigma_E(\mathbf{T}_n) \\ \sigma_E(\mathbf{T}_1 \rightarrow \mathbf{T}_2) &:= \sigma_E(\mathbf{T}_1) \rightarrow \sigma_E(\mathbf{T}_2) \end{aligned}$$

In the following, we do not distinguish between  $\sigma$  and its extension  $\sigma_E$ !

## Semantics of Polymorphic Types



### Semantics

Type substitution  $\sigma$  is **ground for a type**  $\tau$  iff

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

Semantics of type  $\tau$ :

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

This corresponds to the intuition of schematic types:

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

## Unification



### Definition

- A **unification problem** on types is a set  $E$  of equations of the form  $\tau_1 \doteq \tau_2$  where  $\tau_1$  and  $\tau_2$  are polymorphic types.
- A solution to a unification problem on types is a substitution  $\sigma$  (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  $\sigma$  such that for every unifier  $\rho$  of  $E$  there is a substitution  $\gamma$  such that  $\rho(x) = \gamma \circ \sigma(x)$  for all  $x \in \text{Vars}(E)$ .

## Typing Rules



Rule for Application:

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

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:

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

Instead of guessing  $\sigma$ ,  $\sigma$  can be computed: Using **Unification**

## Unification Algorithm



- 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



Fail-rules:

$$\text{FAIL1} \frac{E \cup \{(TC_1 \tau_1 \dots \tau_n) \dot{=} (TC_2 \tau'_1 \dots \tau'_m)\}}{\text{Fail}} \\ \text{if } TC_1 \neq TC_2$$

$$\text{FAIL2} \frac{E \cup \{(TC_1 \tau_1 \dots \tau_n) \dot{=} (\tau'_1 \rightarrow \tau'_2)\}}{\text{Fail}}$$

$$\text{FAIL3} \frac{E \cup \{(\tau'_1 \rightarrow \tau'_2) \dot{=} (TC_1 \tau_1 \dots \tau_n)\}}{\text{Fail}}$$

## Unification Algorithm: Inference Rules (2)



Decomposition:

$$\text{DECOMPOSE1} \frac{E \cup \{TC \tau_1 \dots \tau_n \dot{=} TC \tau'_1 \dots \tau'_n\}}{E \cup \{\tau_1 \dot{=} \tau'_1, \dots, \tau_n \dot{=} \tau'_n\}}$$

$$\text{DECOMPOSE2} \frac{E \cup \{\tau_1 \rightarrow \tau_2 \dot{=} \tau'_1 \rightarrow \tau'_2\}}{E \cup \{\tau_1 \dot{=} \tau'_1, \tau_2 \dot{=} \tau'_2\}}$$

## Unification Algorithm: Inference Rules (3)



Orientation and Elimination:

$$\text{ORIENT} \frac{E \cup \{\tau_1 \dot{=} \alpha\}}{E \cup \{\alpha \dot{=} \tau_1\}} \\ \text{if } \tau_1 \text{ is not a type variable and } \alpha \text{ is a type variable}$$

$$\text{ELIM} \frac{E \cup \{\alpha \dot{=} \alpha\}}{E} \\ \text{where } \alpha \text{ is a type variable}$$

## Unification Algorithm: Inference Rules (4)



Solve and Occurs-Check

$$\text{SOLVE} \frac{E \cup \{\alpha \dot{=} \tau\}}{E[\tau/\alpha] \cup \{\alpha \dot{=} \tau\}} \\ \text{if type variable } \alpha \text{ does not occur in } \tau, \\ \text{but } \alpha \text{ occurs in } E$$

$$\text{OCCURSCHECK} \frac{E \cup \{\alpha \dot{=} \tau\}}{\text{Fail}} \\ \text{if } \tau \neq \alpha \text{ and type variable } \alpha \text{ occurs in } \tau$$

## Examples

hsrm\*

Example 1:  $\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}$ :

$$\text{DECOMPOSE2} \frac{\{(a \rightarrow b) \doteq \text{Bool} \rightarrow \text{Bool}\}}{\{a \doteq \text{Bool}, b \doteq \text{Bool}\}}$$

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

## Examples

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

$$\text{OCCURSCHECK} \frac{\text{SOLVE} \frac{\{a \doteq [b], b \doteq [a]\}}{\{a \doteq [[a]], b \doteq [a]\}}}{\text{Fail}}$$

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

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{a \rightarrow [b] \doteq a \rightarrow c \rightarrow d\}}{\{a \doteq a, [b] \doteq c \rightarrow d\}} \\ \text{ELIM} \frac{\{a \doteq a, [b] \doteq c \rightarrow d\}}{\{[b] \doteq c \rightarrow d\}} \\ \text{FAIL2} \frac{\{[b] \doteq c \rightarrow d\}}{\text{Fail}} \end{array}$$

## Examples

hsrm\*

Example 2:  $\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$ :

$$\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}$$

$$\text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}$$

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \end{array}$$

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \end{array}$$

$$\text{SOLVE} \frac{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}}$$

$$\begin{array}{l} \text{DECOMPOSE2} \frac{\{[d] \doteq c, a \rightarrow [a] \doteq \text{Bool} \rightarrow c\}}{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}} \\ \text{ORIENT} \frac{\{[d] \doteq c, a \doteq \text{Bool}, [a] \doteq c\}}{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}} \\ \text{SOLVE} \frac{\{[d] \doteq c, a \doteq \text{Bool}, c \doteq [a]\}}{\{[d] \doteq [a], a \doteq \text{Bool}, c \doteq [a]\}} \end{array}$$

## Properties of the Unification Algorithm

hsrm\*

- 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  $\{\alpha_1 \doteq \tau_1, \dots, \alpha_n \doteq \tau_n\}$ , where  $\alpha_i$  are pairwise distinct and  $\alpha_i$  does not occur in any  $\tau_j$ .  
The unifier is  $\sigma = \{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}$ .
- 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)

- Types in the result can be of **exponential size**

E.g.  $\{\alpha_n \doteq \alpha_{n-1} \rightarrow \alpha_{n-1}, \alpha_{n-1} \doteq \alpha_{n-2} \rightarrow \alpha_{n-2}, \dots, \alpha_1 \doteq \alpha_0 \rightarrow \alpha_0\}$

The unifier maps  $\alpha_i$  to a type that contains  $2^i - 1$  type arrows. E.g.

$$\sigma(\alpha_1) = \alpha_0 \rightarrow \alpha_0,$$

$$\sigma(\alpha_2) = (\alpha_0 \rightarrow \alpha_0) \rightarrow (\alpha_0 \rightarrow \alpha_0),$$

$$\sigma(\alpha_3) = ((\alpha_0 \rightarrow \alpha_0) \rightarrow (\alpha_0 \rightarrow \alpha_0)) \rightarrow ((\alpha_0 \rightarrow \alpha_0) \rightarrow (\alpha_0 \rightarrow \alpha_0))$$

- 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

Let  $E$  be a unification problem and

- $Var(E)$  = number of **unsolved** type variables in  $E$   
a variable  $\alpha$  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  $\alpha \notin Vars(E') \cup Vars(\tau)$ ).
- $Size(E)$  = **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:  $tsize(TV) = 1$ ,  
 $tsize(TC\ T_1 \dots T_n) = 1 + \sum_{i=1}^n tsize(T_i)$  and  
 $tsize(T_1 \rightarrow T_2) = 1 + tsize(T_1) + tsize(T_2)$
- $OEq(E)$  = number of **not oriented** equations in  $E$   
an equation is oriented, if it is of the form  $\alpha \doteq \tau$  where  $\alpha$  is a type variable.
- $M(E) = (Var(E), Size(E), OEq(E))$ , where  $M(Fail) := (-1, -1, -1)$ .

## Sketch of the Termination Proof (Cont'd)

Change of the measure per rule

	$Var(E)$	$Size(E)$	$OEq(E)$
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.

## 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



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

if  $\sigma$  is an mgu for  $\tau_1 \doteq \tau_2 \rightarrow \alpha$  and  $\alpha$  is a fresh type variable

Example:

$$\frac{\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b], \text{not} :: \text{Bool} \rightarrow \text{Bool}}{(\text{map not}) :: \sigma(\alpha)}$$

if  $\sigma$  is an mgu for  $(a \rightarrow b) \rightarrow [a] \rightarrow [b] \doteq (\text{Bool} \rightarrow \text{Bool}) \rightarrow \alpha$   
and  $\alpha$  is a fresh type variable

Unification results in  $\{a \mapsto \text{Bool}, b \mapsto \text{Bool}, \alpha \mapsto [\text{Bool}] \rightarrow [\text{Bool}]\}$

Thus:  $\sigma(\alpha) = [\text{Bool}] \rightarrow [\text{Bool}]$

## 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  $\tau_1 \rightarrow \tau$
- How corresponds  $\tau_1$  with  $\tau$ ?
- $\tau_1$  is the type of  $x$
- If  $x$  occurs in  $s$ , then we need  $\tau_1$  for typing  $\tau$ !

## Typing with Binders (Cont'd)



Informal rule for abstractions:

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

How do we get  $\tau_1$ ?

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

Example:

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

Typing judgement:

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

Meaning:

Given a set  $\Gamma$  of type assumptions, for expression  $s$  the type  $\tau$  and the type equations  $E$  can be derived

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

Type derivation rules are written as

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

or more concrete:

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

As a simplification:

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

Axiom for variables:

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

Axiom for constructors:

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

where  $\beta_i$  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 known type):

$$(AxSC) \frac{}{\Gamma \cup \{SC :: \forall \alpha_1 \dots \alpha_n. \tau\} \vdash SC :: \tau[\beta_1/\alpha_1, \dots, \beta_n/\alpha_n], \emptyset}$$

where  $\beta_i$  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:

$$(RAPP) \frac{\Gamma \vdash s :: \tau_1, E_1 \quad \text{und} \quad \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha\}}$$

where  $\alpha$  is a fresh type variable

### Rule for seq:

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

## Typing Rules for KFPTS+seq-Expressions (4)



### Rule for abstractions:

$$(RABS) \frac{\Gamma \cup \{x :: \alpha\} \vdash s :: \tau, E}{\Gamma \vdash \lambda x. s :: \alpha \rightarrow \tau, E}$$

where  $\alpha$  is a fresh type variable

## Typing Rules for KFPTS+seq-Expressions (5)



### Typing of case: ideas

$$\left( \text{case}_T s \text{ of } \left\{ \begin{array}{l} (c_1 x_{1,1} \dots x_{1,ar(c_1)}) \rightarrow t_1; \\ \dots; \\ (c_m x_{m,1} \dots x_{m,ar(c_m)}) \rightarrow t_m \end{array} \right\} \right)$$

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

## Rule for case:

$$\begin{array}{c}
\Gamma \vdash s :: \tau, E \\
\text{for } i = 1, \dots, m: \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,ar(c_i)} :: \alpha_{i,ar(c_i)}\} \vdash (c_i x_{i,1} \dots x_{i,ar(c_i)}) :: \tau_i, E_i \\
\text{for } i = 1, \dots, m: \Gamma \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,ar(c_i)} :: \alpha_{i,ar(c_i)}\} \vdash t_i :: \tau'_i, E'_i \\
\text{(RCASE)} \frac{}{\Gamma \vdash \left( \text{case}_T s \text{ of } \left\{ \begin{array}{l} (c_1 x_{1,1} \dots x_{1,ar(c_1)}) \rightarrow t_1; \\ \dots; \\ (c_m x_{m,1} \dots x_{m,ar(c_m)}) \rightarrow t_m \end{array} \right\} \right) :: \alpha, E'} \\
\text{where } E' = E \cup \bigcup_{i=1}^m E_i \cup \bigcup_{i=1}^m E'_i \cup \bigcup_{i=1}^m \{\tau \doteq \tau_i, \tau \doteq \tau'_i\} \cup \bigcup_{i=1}^m \{\alpha \doteq \tau'_i\} \\
\text{and } \alpha_{i,j}, \alpha \text{ are fresh type variables}
\end{array}$$

$$\begin{array}{c}
\Gamma \vdash s :: \tau, E \quad \Gamma \vdash \text{True} :: \tau_1, E_1 \quad \Gamma \vdash \text{False} :: \tau_2, E_2 \quad \Gamma \vdash t_1 :: \tau'_1, E'_1 \quad \Gamma \vdash t_2 :: \tau'_2, E'_2 \\
\text{(RCASE)} \frac{}{\Gamma \vdash (\text{case}_{\text{Bool}} s \text{ of } \{\text{True} \rightarrow t_1; \text{False} \rightarrow t_2\}) :: \alpha, E'} \\
\text{where } E' = E \cup E_1 \cup E_2 \cup E'_1 \cup E'_2 \cup \{\tau \doteq \tau_1, \tau \doteq \tau_2\} \cup \{\alpha \doteq \tau'_1, \alpha \doteq \tau'_2\} \\
\text{and } \alpha_{i,j}, \alpha \text{ are fresh type variables}
\end{array}$$

$$\begin{array}{c}
\Gamma \vdash s :: \tau, E \\
\Gamma \vdash \text{Nil} :: \tau_1, E_1 \\
\Gamma \cup \{x_1 :: \alpha_1, x_2 :: \alpha_2\} \vdash \text{Cons } x_1 x_2 :: \tau_2, E_2 \\
\Gamma \vdash t_1 :: \tau'_1, E'_1 \\
\Gamma \cup \{x_1 :: \alpha_1, x_2 :: \alpha_2\} \vdash t_2 :: \tau'_2, E'_2 \\
\text{(RCASE)} \frac{}{\Gamma \vdash (\text{case}_{\text{List}} s \text{ of } \{\text{Nil} \rightarrow t_1; (\text{Cons } x_1 x_2) \rightarrow t_2\}) :: \alpha, E'} \\
\text{where } E' = E \cup E_1 \cup E_2 \cup E'_1 \cup E'_2 \cup \{\tau \doteq \tau_1, \tau \doteq \tau_2\} \cup \{\alpha \doteq \tau'_1, \alpha \doteq \tau'_2\} \\
\text{and } \alpha_{i,j}, \alpha \text{ are fresh type variables}
\end{array}$$

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  $\Gamma$  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  $\sigma$  be an mgu of  $E$ . Then the type of  $s$  is  $s :: \sigma(\tau)$ .

Additional rule to unify inbetween:

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

where  $E_\sigma$  is the solved equation system of  $E$  and  $\sigma$  is the corresponding unifier

**Definition**

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

## Example: Typing of (Cons True Nil)

Start with:

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

$$(R_{APP}) \frac{\Gamma_0 \vdash (\text{Cons True}) :: \tau_1, E_1, \quad \Gamma_0 \vdash \text{Nil} :: \tau_2, E_2}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_4\}}$$

$$(R_{APP}) \frac{\Gamma_0 \vdash (\text{Cons True}) :: \tau_1, E_1, \quad (AxC) \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, E_1 \cup \emptyset \cup \{\tau_1 \doteq [\alpha_3] \rightarrow \alpha_4\}}$$

$$(R_{APP}) \frac{\Gamma_0 \vdash \text{Cons} :: \tau_3, E_3, \Gamma_0 \vdash \text{True} :: \tau_4, E_4}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_2\} \cup E_3 \cup E_4, \quad (AxC) \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}$$

$$(R_{APP}) \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_2\} \cup E_3 \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}$$

$$(AxC) \frac{}{\Gamma_0 \vdash \text{Cons} :: \alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1], \emptyset, \Gamma_0 \vdash \text{True} :: \tau_4, E_4}$$

$$(R_{APP}) \frac{}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \tau_4 \rightarrow \alpha_2\} \cup E_4, \quad (AxC) \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}$$

$$(R_{APP}) \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \tau_4 \rightarrow \alpha_2\} \cup E_4 \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}$$

$$(R_{APP}) \frac{}{\Gamma_0 \vdash (\text{Cons True}) :: \alpha_2, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\}, \quad (AxC) \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_3], \emptyset}}$$

$$(R_{APP}) \frac{}{\Gamma_0 \vdash (\text{Cons True Nil}) :: \alpha_4, \{\alpha_1 \rightarrow [\alpha_1] \rightarrow [\alpha_1] \doteq \text{Bool} \rightarrow \alpha_2\} \cup \{\alpha_2 \doteq [\alpha_3] \rightarrow \alpha_4\}}$$

Example: Typing  $\lambda x.x$ 

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

$$(R_{ABS}) \frac{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \tau, E \quad (AxV) \frac{}{\Gamma_0 \cup \{x :: \alpha\} \vdash x :: \alpha, \emptyset}}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \rightarrow \tau, E} \quad (R_{ABS}) \frac{}{\Gamma_0 \vdash (\lambda x.x) :: \alpha \rightarrow \alpha, \emptyset}$$

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

## Example: Typing of $\Omega$

hsrm\*

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

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

$$\begin{array}{c}
 \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 \rightarrow \alpha_1\}} \\
 \text{(RAbs)} \frac{\{x :: \alpha_2\} \vdash (x x) :: \tau_1, E_1}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \tau_1, E_1}, \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2 \\
 \text{(RAPP)} \frac{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \alpha_1\}}{\{x :: \alpha_2\} \vdash x :: \tau_3, E_3, \{x :: \alpha_2\} \vdash x :: \tau_4, E_4,} \\
 \text{(RAPP)} \frac{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4}{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4}, \emptyset \vdash (\lambda y.(y y)) :: \tau_2, E_2 \\
 \text{(RAbs)} \frac{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\tau_3 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_3 \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \\
 \text{(AxV)} \frac{\{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 \rightarrow \alpha_3\} \cup E_4} \\
 \text{(RAPP)} \frac{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \\
 \text{(AxV)} \frac{\{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset, \{x :: \alpha_2\} \vdash x :: \alpha_2, \emptyset,}{\{x :: \alpha_2\} \vdash (x x) :: \alpha_3, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\}} \\
 \text{(RAbs)} \frac{\emptyset \vdash (\lambda x.(x x)) :: \alpha_2 \rightarrow \alpha_3, \{\alpha_2 \doteq \alpha_2 \rightarrow \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 \rightarrow \alpha_3\} \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}} \\
 \text{(RAPP)} \frac{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \alpha_2 \rightarrow \alpha_3\} \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}}{\emptyset \vdash (\lambda x.(x x)) (\lambda y.(y y)) :: \alpha_1, \{\alpha_2 \doteq \tau_4 \rightarrow \alpha_3\} \cup E_4 \cup E_2 \cup \{\alpha_3 \doteq \tau_2 \rightarrow \alpha_1\}}
 \end{array}$$

## Example: Expression with Supercombinators (1)

hsrm\*

Assumption: map and length are already typed.

We type:

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

We use the start assumption:

$$\begin{array}{l}
 \Gamma_0 = \{\text{map} :: \forall a, b. (a \rightarrow b) \rightarrow [a] \rightarrow [b], \\
 \text{length} :: \forall a. [a] \rightarrow \text{Int}, \\
 \text{Nil} :: \forall a. [a] \\
 \text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a] \\
 \}
 \end{array}$$

## Example: Expression with Supercombinators (2)

hsrm\*

Derivation tree:

$$\begin{array}{c}
 \text{(AxV)} \overline{B_3}, \text{(AxV)} \overline{B_9}, \text{(AxV)} \overline{B_7}, \text{(AxV)} \overline{B_{10}}, \text{(AxV)} \overline{B_{11}}, \text{(AxV)} \overline{B_{13}} \\
 \text{(RCASE)} \frac{\overline{B_3}, \text{(AxV)} \overline{B_9}, \text{(AxV)} \overline{B_7}, \text{(AxV)} \overline{B_{10}}, \text{(AxV)} \overline{B_{11}}, \text{(AxV)} \overline{B_{13}}}{\overline{B_2}} \\
 \text{(RAPP)} \frac{\overline{B_2}}{\overline{B_1}}
 \end{array}$$

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

## Example: Expression with Supercombinators (3)

hsrm\*

$$\begin{array}{c}
 \text{(AxV)} \overline{B_3}, \text{(AxV)} \overline{B_4}, \text{(AxV)} \overline{B_5}, \text{(AxV)} \overline{B_6}, \text{(AxV)} \overline{B_7}, \text{(AxV)} \overline{B_8}, \text{(AxV)} \overline{B_9}, \text{(AxV)} \overline{B_{10}}, \text{(AxV)} \overline{B_{11}}, \text{(AxV)} \overline{B_{12}}, \text{(AxV)} \overline{B_{13}}, \text{(AxV)} \overline{B_{14}}, \text{(AxV)} \overline{B_{15}} \\
 \text{(RCASE)} \frac{\overline{B_3}, \text{(AxV)} \overline{B_4}, \text{(AxV)} \overline{B_5}, \text{(AxV)} \overline{B_6}, \text{(AxV)} \overline{B_7}, \text{(AxV)} \overline{B_8}, \text{(AxV)} \overline{B_9}, \text{(AxV)} \overline{B_{10}}, \text{(AxV)} \overline{B_{11}}, \text{(AxV)} \overline{B_{12}}, \text{(AxV)} \overline{B_{13}}, \text{(AxV)} \overline{B_{14}}, \text{(AxV)} \overline{B_{15}}}{\overline{B_2}} \\
 \text{(RAPP)} \frac{\overline{B_2}}{\overline{B_1}}
 \end{array}$$

Labels:  $B_3 = \Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \alpha_1, \emptyset$   
 $B_4 = \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: [\alpha_2], \emptyset$   
 $B_5 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\text{Cons } y \text{ } ys) :: \alpha_7,$   
 $\{\alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6, \alpha_6 \doteq \alpha_4 \rightarrow \alpha_7\}$   
 $B_6 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash (\text{Cons } y) :: \alpha_6,$   
 $\{\alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \alpha_3 \rightarrow \alpha_6\}$   
 $B_7 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash ys :: \alpha_4, \emptyset$   
 $B_8 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset$   
 $B_9 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_3, ys :: \alpha_4\} \vdash y :: \alpha_3, \emptyset$   
 $B_{10} = \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: [\alpha_{14}], \emptyset$



## 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
```

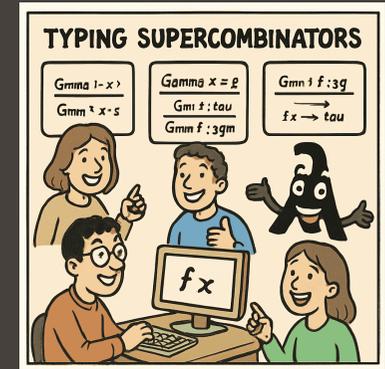
```
Inferred type: Bool -> a
```

```
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

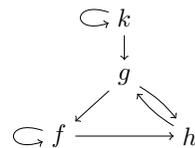
### Definition

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

- $SC_i \preceq SC_j$  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  $SC_i \preceq^+ SC_i$
- $SC_1, \dots, SC_m$  are **mutually recursive** if  $SC_i \preceq^+ SC_j$  for all  $i, j \in \{1, \dots, m\}$ .

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  $\Gamma$ ,  $SC$  can be typed with type  $\tau$

**Typing rule for (closed) non-recursive supercombinators:**

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

if  $\sigma$  is the solution of  $E$ ,

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

and  $SC$  is non-recursive,

and  $\mathcal{X} = \text{Vars}(\sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau))$

## Example: Typing of (.)

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

$\Gamma_0$  is empty, since no constructors or supercombinators occur

$$\frac{\frac{\frac{\frac{}{(AxV)}{\Gamma_1 \vdash f :: \alpha_1, \emptyset}}{\Gamma_1 \vdash f :: \alpha_1, \emptyset}, \frac{\frac{\frac{}{(AxV)}{\Gamma_1 \vdash g :: \alpha_2, \emptyset}, \frac{}{(AxV)}{\Gamma_1 \vdash x :: \alpha_3, \emptyset}}{\Gamma_1 \vdash (g x) :: \alpha_5, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5\}}}{\Gamma_1 \vdash (f (g x)) :: \alpha_4, \{\alpha_2 \dot{=} \alpha_3 \rightarrow \alpha_5, \alpha_1 = \alpha_5 \rightarrow \alpha_4\}}}{\emptyset \vdash_T (. ) :: \forall \mathcal{X}. \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4)}}{(RApp)} \quad \frac{}{(RSC1)}$$

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

Unification results in  $\sigma = \{\alpha_2 \mapsto \alpha_3 \rightarrow \alpha_5, \alpha_1 \mapsto \alpha_5 \rightarrow \alpha_4\}$ .

Thus:  $\sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4) = (\alpha_5 \rightarrow \alpha_4) \rightarrow (\alpha_3 \rightarrow \alpha_5) \rightarrow \alpha_3 \rightarrow \alpha_4$

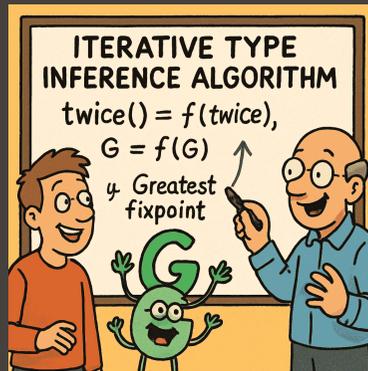
Now  $\mathcal{X} = \{\alpha_3, \alpha_4, \alpha_5\}$  and we may rename this to:

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

## 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  $\text{sem}(T) = \{\text{all monomorphic types}\}$ .

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

Rule to compute new assumptions:

$$(SC_{REC}) \frac{\Gamma \cup \{x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \sigma(\alpha_1 \rightarrow \dots \alpha_n \rightarrow \tau)}$$

if  $SC \ x_1 \dots x_n = s$  is the definition of  $SC$ ,  $\sigma$  the solution of  $E$

The same as RSC1, but  $\Gamma$  has to contain an assumption for  $SC$

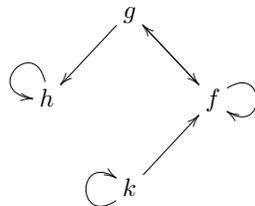
Because of mutual recursion:

- 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  $\simeq$ .

```
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**

**Input:** Mutually recursive supercombinators  $SC_1, \dots, SC_m$

- 1 Start assumption  $\Gamma$  contains types of the constructors and the already typed SCs
- 2  $\Gamma_0 := \Gamma \cup \{SC_1 :: \forall \alpha_1. \alpha_1, \dots, SC_m :: \forall \alpha_m. \alpha_m\}$  and  $j = 0$ .
- 3 For each  $SC_i$  ( $i = 1, \dots, m$ ) apply rule (SCREC) for  $\Gamma_j$ , 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:  $SC_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SC_m :: \forall \mathcal{X}_m. \tau_m$   
Set  $\Gamma_{j+1} := \Gamma \cup \{SC_1 :: \forall \mathcal{X}_1. \tau_1, \dots, SC_m :: \forall \mathcal{X}_m. \tau_m\}$
- 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  $\Gamma_j$  is **consistent**.

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

## Properties of the Algorithm

hsrcm\*

- 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.  $\text{sem}$ : " $\text{sem}(T_{j+1}) \subseteq \text{sem}(T_j)$ ")
- If the algorithm does not terminate, then no polymorphic type for the supercombinators exists (since computation is monotonic w.r.t.  $\text{sem}$  and starts with the largest set)
- The algorithm computes the **greatest fixpoint** w.r.t.  $\text{sem}$ :  
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.  $\text{sem}$ )

## Example: length (1)

hsrcm\*

$$\text{length } xs = \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } ys\}$$

Assumption:

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

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

$$\begin{array}{l} (a) \Gamma_0 \cup \{xs :: \alpha_1\} \vdash xs :: \tau_1, E_1 \\ (b) \Gamma_0 \cup \{xs :: \alpha_1\} \vdash \text{Nil} :: \tau_2, E_2 \\ (c) \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (y : ys) :: \tau_3, E_3 \\ (d) \Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \tau_4, E_4 \\ (e) \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\} \vdash (1 + \text{length } ys) :: \tau_5, E_5 \end{array}$$

$$\frac{\text{(RCASE)} \quad \Gamma_0 \cup \{xs :: \alpha_1\} \vdash (\text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow 0; (y : ys) \rightarrow 1 + \text{length } xs\}) :: \alpha_3, \quad E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup \{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \alpha_3 \doteq \tau_4, \alpha_3 \doteq \tau_5\}}{\text{(SCRCE)} \quad \Gamma_0 \vdash_T \text{length} :: \sigma(\alpha_1 \rightarrow \alpha_3)}$$

where  $\sigma$  is the solution of

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

## Example: length (2)

hsrcm\*

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

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

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

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

$$(c): \frac{\text{(AXC)} \frac{\Gamma'_0 \vdash (:) :: \alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9], \emptyset, \text{(AXV)} \frac{\Gamma'_0 \vdash y :: \alpha_4, \emptyset}{\Gamma'_0 \vdash y :: \alpha_4, \emptyset}}{\Gamma'_0 \vdash ((:) y) :: \alpha_7, \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \doteq \alpha_4 \rightarrow \alpha_8\}}, \text{(AXV)} \frac{\Gamma'_0 \vdash ys :: \alpha_5, \emptyset}{\Gamma'_0 \vdash ys :: \alpha_5, \emptyset}}{\text{(RAPP)} \quad \Gamma_0 \vdash (y : ys) :: \alpha_7, \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \doteq \alpha_4 \rightarrow \alpha_8, \alpha_8 \doteq \alpha_5 \rightarrow \alpha_7\}}$$

where  $\Gamma_0 = \Gamma_0 \cup \{xs :: \alpha_1, y :: \alpha_4, ys :: \alpha_5\}$   
i.e.,  $\tau_3 = \alpha_7$  and  $E_3 = \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \doteq \alpha_4 \rightarrow \alpha_8, \alpha_8 \doteq \alpha_5 \rightarrow \alpha_7\}$

## Example: length (3)

hsrcm\*

$$(d): \frac{\text{(AXC)}}{\Gamma_0 \cup \{xs :: \alpha_1\} \vdash 0 :: \text{Int}, \emptyset}$$

i.e.  $\tau_4 = \text{Int}$  und  $E_4 = \emptyset$

$$(e): \frac{\text{(ASC)} \frac{\Gamma'_0 \vdash (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \emptyset, \text{(ASC)} \frac{\Gamma'_0 \vdash 1 :: \text{Int}, \emptyset}{\Gamma'_0 \vdash 1 :: \text{Int}, \emptyset}, \text{(ASC)} \frac{\Gamma'_0 \vdash \text{length} :: \alpha_{13}, \emptyset}{\Gamma'_0 \vdash \text{length} :: \alpha_{13}, \emptyset}, \text{(ASV)} \frac{\Gamma'_0 \vdash (ys) :: \alpha_5, \emptyset}{\Gamma'_0 \vdash (ys) :: \alpha_5, \emptyset}}{\text{(RAPP)} \quad \Gamma'_0 \vdash ((+) 1) :: \alpha_{11}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \doteq \text{Int} \rightarrow \alpha_{11}\}}, \text{(RAPP)} \quad \Gamma'_0 \vdash (\text{length } ys) :: \alpha_{12}, \{\alpha_{13} \doteq \alpha_5 \rightarrow \alpha_{12}\}}{\text{(RAPP)} \quad \Gamma_0 \vdash (1 + \text{length } ys) :: \alpha_{10}, \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \doteq \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \doteq \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \rightarrow \alpha_{10}\}}$$

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

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

$$E_5 = \{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \doteq \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \doteq \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \doteq \alpha_{12} \rightarrow \alpha_{10}\}$$

## Example: length (4)

hsrm\*

In summary:  $\Gamma_0 \vdash_T \text{length} :: \sigma(\alpha_1 \rightarrow \alpha_3)$

where  $\sigma$  is the solution of

$$\begin{aligned} \{\alpha_9 \rightarrow [\alpha_9] \rightarrow [\alpha_9] \dot{=} \alpha_4 \rightarrow \alpha_8, \alpha_8 \dot{=} \alpha_5 \rightarrow \alpha_7, \\ \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \dot{=} \text{Int} \rightarrow \alpha_{11}, \alpha_{13} \dot{=} \alpha_5 \rightarrow \alpha_{12}, \alpha_{11} \dot{=} \alpha_{12} \rightarrow \alpha_{10}, \\ \alpha_1 \dot{=} [\alpha_6], \alpha_1 \dot{=} \alpha_7, \alpha_3 \dot{=} \text{Int}, \alpha_3 \dot{=} \alpha_{10}\} \end{aligned}$$

Unification results in the unifier:

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

thus  $\sigma(\alpha_1 \rightarrow \alpha_3) = [\alpha_9] \rightarrow \text{Int}$

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

Since  $\Gamma_0 \neq \Gamma_1$  another iteration is required.

2. iteration: It results in the same type, hence  $\Gamma_1$  is consistent.

## Iterative Typing is More General than Haskell

hsrm\*

Example

$$g \ x = 1 : (g \ (g \ 'c'))$$

$$\Gamma = \{1 :: \text{Int}, \text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}$$

$$\Gamma_0 = \Gamma \cup \{g :: \forall \alpha. \alpha\} \text{ (and } \Gamma'_0 = \Gamma_0 \cup \{x :: \alpha_1\})$$

$$\begin{aligned} & \frac{\text{(ASXC)} \frac{\Gamma'_0 \vdash g :: \alpha_8, \emptyset}{\Gamma'_0 \vdash g :: \alpha_6, \emptyset}, \text{(ASXC)} \frac{\Gamma'_0 \vdash 1 :: \text{Int}, \emptyset}{\Gamma'_0 \vdash 1 :: \text{Int}, \emptyset}}{\Gamma'_0 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \text{(ASXC)} \frac{\Gamma'_0 \vdash g :: \alpha_6, \emptyset}{\Gamma'_0 \vdash (g \ 'c') :: \alpha_7, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7\}}, \text{(ASXC)} \frac{\Gamma'_0 \vdash 'c' :: \text{Char}, \emptyset}{\Gamma'_0 \vdash 'c' :: \text{Char}, \emptyset}}{\Gamma'_0 \vdash (\text{Cons } 1) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \text{Int} \rightarrow \alpha_3}, \text{(RAFP)} \frac{\Gamma'_0 \vdash (g \ 'c') :: \alpha_7, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4\}}{\Gamma'_0 \vdash \text{Cons } 1 \ (g \ (g \ 'c')) :: \alpha_2, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \text{Int} \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}} \\ & \text{(SCRB)} \frac{\Gamma_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_1 \rightarrow [\text{Int}], \Gamma_0 \vdash_T 1 :: \text{Int}, \alpha_4 \mapsto [\text{Int}], \alpha_5 \mapsto \text{Int}, \alpha_6 \mapsto \alpha_7 \rightarrow [\text{Int}], \alpha_8 \mapsto \text{Char} \rightarrow \alpha_7}{\Gamma_0 \vdash_T \text{Cons } 1 \ (g \ (g \ 'c')) :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_1 \rightarrow [\text{Int}], \alpha_4 \mapsto [\text{Int}], \alpha_5 \mapsto \text{Int}, \alpha_6 \mapsto \alpha_7 \rightarrow [\text{Int}], \alpha_8 \mapsto \text{Char} \rightarrow \alpha_7} \text{ is the solution of} \\ & \{\alpha_2 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \text{Int} \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\} \end{aligned}$$

i.e.  $\Gamma_1 = \Gamma \cup \{g :: \forall \alpha. \alpha \rightarrow [\text{Int}]\}$ .

The next iteration shows that  $\Gamma_1$  is consistent.

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

hsrm\*

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)

hsrm\*

$$g \ x = x : (g \ (g \ 'c'))$$

$$\bullet \Gamma = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}$$

$$\bullet \Gamma_0 = \Gamma \cup \{g :: \forall \alpha. \alpha\}$$

$$\begin{aligned} & \frac{\text{(ASXC)} \frac{\Gamma'_0 \vdash g :: \alpha_8, \emptyset}{\Gamma'_0 \vdash g :: \alpha_6, \emptyset}, \text{(ASXC)} \frac{\Gamma'_0 \vdash x :: \alpha_1, \emptyset}{\Gamma'_0 \vdash x :: \alpha_1, \emptyset}}{\Gamma'_0 \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \text{(ASXC)} \frac{\Gamma'_0 \vdash g :: \alpha_6, \emptyset}{\Gamma'_0 \vdash (g \ 'c') :: \alpha_7, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7\}}, \text{(ASXC)} \frac{\Gamma'_0 \vdash 'c' :: \text{Char}, \emptyset}{\Gamma'_0 \vdash 'c' :: \text{Char}, \emptyset}}{\Gamma'_0 \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3}, \text{(RAFP)} \frac{\Gamma'_0 \vdash (g \ 'c') :: \alpha_7, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4\}}{\Gamma'_0 \vdash \text{Cons } x \ (g \ (g \ 'c')) :: \alpha_2, \{\alpha_8 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}} \\ & \text{(SCRB)} \frac{\Gamma_0 \vdash_T g :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_5 \rightarrow [\alpha_5], \Gamma_0 \vdash_T x :: \text{Int}, \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \rightarrow [\alpha_5], \alpha_8 \mapsto \text{Char} \rightarrow \alpha_7}{\Gamma_0 \vdash_T \text{Cons } x \ (g \ (g \ 'c')) :: \sigma(\alpha_1 \rightarrow \alpha_2) = \alpha_5 \rightarrow [\alpha_5], \alpha_4 \mapsto [\alpha_5], \alpha_6 \mapsto \alpha_7 \rightarrow [\alpha_5], \alpha_8 \mapsto \text{Char} \rightarrow \alpha_7} \text{ is the solution of} \\ & \{\alpha_2 \dot{=} \text{Char} \rightarrow \alpha_7, \alpha_6 \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha_1 \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\} \end{aligned}$$

i.e.  $\Gamma_1 = \Gamma \cup \{g :: \forall \alpha. \alpha \rightarrow [\alpha]\}$ .

## Example: Multiple Iterations are Required (2)

hsrm\*

Since  $\Gamma_0 \neq \Gamma_1$  another iteration is required.

Let  $\Gamma'_1 = \Gamma_1 \cup \{x :: \alpha_1\}$ :

$$\frac{\frac{\frac{\text{(ASCC)} \Gamma'_1 \vdash \mathbf{g} :: \alpha_8 \rightarrow [\alpha_8], \emptyset, \quad \text{(ASCC)} \Gamma'_1 \vdash 'c' :: \mathbf{Char}, \emptyset,}{\text{(RAPP)} \Gamma'_1 \vdash (\mathbf{Cons} \ x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \rightarrow [\alpha_5] \rightarrow [\alpha_5] \rightarrow \alpha_1 \rightarrow \alpha_3, \quad \text{(RAPP)} \Gamma'_1 \vdash (\mathbf{g} \ 'c') :: \alpha_7, \{\alpha_8 \rightarrow [\alpha_8] \doteq \mathbf{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \doteq \mathbf{Char} \rightarrow \alpha_7\}}{\text{(RAPP)} \Gamma'_1 \vdash \mathbf{Cons} \ x (\mathbf{g} \ (\mathbf{g} \ 'c')) :: \alpha_2, \{\alpha_8 \rightarrow [\alpha_8] \doteq \mathbf{Char} \rightarrow \alpha_7, \alpha_6 \rightarrow [\alpha_6] \doteq \mathbf{Char} \rightarrow \alpha_7, \alpha_5 \rightarrow [\alpha_5] \rightarrow \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}}{\text{(SCREC)} \Gamma_1 \vdash \mathbf{g} :: \sigma(\alpha_1 \rightarrow \alpha_2) = [\mathbf{Char}] \rightarrow [[\mathbf{Char}]]}$$

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

Hence  $\Gamma_2 = \Gamma \cup \{\mathbf{g} :: [\mathbf{Char}] \rightarrow [[\mathbf{Char}]]\}$ .

## Example: Multiple Iterations are Required (3)

hsrm\*

Since  $\Gamma_1 \neq \Gamma_2$  another iteration is required:

Let  $\Gamma'_2 = \Gamma_2 \cup \{x :: \alpha_1\}$ :

$$\frac{\frac{\frac{\text{(ASCC)} \Gamma'_2 \vdash \mathbf{g} :: [\mathbf{Char}] \rightarrow [[\mathbf{Char}]], \emptyset, \quad \text{(ASCC)} \Gamma'_2 \vdash 'c' :: \mathbf{Char}, \emptyset,}{\text{(RAPP)} \Gamma'_2 \vdash (\mathbf{Cons} \ x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \rightarrow [\alpha_5] \rightarrow \alpha_1 \rightarrow \alpha_3, \quad \text{(RAPP)} \Gamma'_2 \vdash (\mathbf{g} \ (\mathbf{g} \ 'c')) :: \alpha_4, \{[\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \mathbf{Char} \rightarrow \alpha_7, [\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \alpha_7 \rightarrow \alpha_4\}}{\text{(RAPP)} \Gamma'_2 \vdash \mathbf{Cons} \ x (\mathbf{g} \ (\mathbf{g} \ 'c')) :: \alpha_2, \{[\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \mathbf{Char} \rightarrow \alpha_7, [\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}}{\text{(SCREC)} \Gamma_2 \vdash \mathbf{g} :: \sigma(\alpha_1 \rightarrow \alpha_2)}$$

where  $\sigma$  is the solution of  
 $\{[\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \mathbf{Char} \rightarrow \alpha_7, [\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow \alpha_1 \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}$

Unification:

$$\frac{[\mathbf{Char}] \rightarrow [[\mathbf{Char}]] \doteq \mathbf{Char} \rightarrow \alpha_7, \dots}{[\mathbf{Char}] \doteq \mathbf{Char}, \dots} \frac{[\mathbf{Char}] \doteq \mathbf{Char}, \dots}{[[\mathbf{Char}]] \doteq \alpha_7, \dots} \frac{\dots}{Fail}$$

**g is not typeable.**

## The Example Shows ...

hsrm\*

### 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)

hsrm\*

$$\begin{aligned} f &= [g] \\ g &= [f] \end{aligned}$$

Since  $f \simeq g$ , the iterative typing types  $f$  and  $g$  together.

$$\Gamma = \{\mathbf{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], \mathbf{Nil} : \forall a. a\}.$$

$$\Gamma_0 = \Gamma \cup \{f :: \forall \alpha. \alpha, g :: \forall \alpha. \alpha\}$$

$$\frac{\frac{\text{(AXC)} \Gamma_0 \vdash \mathbf{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset, \quad \text{(AXSC)} \Gamma_0 \vdash \mathbf{g} :: \alpha_5}{\text{(RAPP)} \Gamma_0 \vdash (\mathbf{Cons} \ \mathbf{g}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3\}}, \quad \text{(AXC)} \Gamma_0 \vdash \mathbf{Nil} :: [\alpha_2], \emptyset}{\text{(RAPP)} \Gamma_0 \vdash [g] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}}}{\text{(SCREC)} \Gamma_0 \vdash \mathbf{f} :: \sigma(\alpha_1) = [\alpha_5]}$$

$$\sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \rightarrow [\alpha_5], \alpha_4 \mapsto \alpha_5\}$$

is the solution of  $\{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}$

## Non-Termination of the Iterative Typing (2)

hsrcm\*

$$\begin{array}{c}
 \text{(AXC)} \frac{}{\Gamma_0 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \text{(AXSC)} \frac{}{\Gamma_0 \vdash \mathbf{f} :: \alpha_5} \\
 \text{(RAPP)} \frac{}{\Gamma_0 \vdash (\text{Cons } \mathbf{g}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3\}}, \text{(AXC)} \frac{}{\Gamma_0 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{\Gamma_0 \vdash [\mathbf{f}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SCREC)} \frac{}{\Gamma_0 \vdash_T \mathbf{g} :: \sigma(\alpha_1) = [[\alpha_5]]} \\
 \sigma = \{\alpha_1 \mapsto [\alpha_5], \alpha_2 \mapsto \alpha_5, \alpha_3 \mapsto [\alpha_5] \rightarrow [\alpha_5], \alpha_4 \mapsto \alpha_5\} \text{ is} \\
 \text{the solution of } \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq \alpha_5 \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}
 \end{array}$$

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

## Non-Termination of the Iterative Typing (3)

hsrcm\*

$$\begin{array}{c}
 \text{(AXC)} \frac{}{\Gamma_1 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \text{(AXSC)} \frac{}{\Gamma_1 \vdash \mathbf{g} :: [\alpha_5]} \\
 \text{(RAPP)} \frac{}{\Gamma_1 \vdash (\text{Cons } \mathbf{g}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5] \rightarrow \alpha_3\}}, \text{(AXC)} \frac{}{\Gamma_1 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(SCREC)} \frac{}{\Gamma_1 \vdash [\mathbf{g}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5] \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \Gamma_1 \vdash_T \mathbf{f} :: \sigma(\alpha_1) = [[\alpha_5]] \\
 \sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \rightarrow [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\} \text{ is} \\
 \text{the solution of } \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5] \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}
 \end{array}$$

$$\begin{array}{c}
 \text{(AXC)} \frac{}{\Gamma_1 \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \text{(AXSC)} \frac{}{\Gamma_1 \vdash \mathbf{f} :: [\alpha_5]} \\
 \text{(RAPP)} \frac{}{\Gamma_1 \vdash (\text{Cons } \mathbf{f}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5] \rightarrow \alpha_3\}}, \text{(AXC)} \frac{}{\Gamma_1 \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(SCREC)} \frac{}{\Gamma_1 \vdash [\mathbf{f}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5] \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \Gamma_1 \vdash_T \mathbf{g} :: \sigma(\alpha_1) = [[\alpha_5]] \\
 \sigma = \{\alpha_1 \mapsto [[\alpha_5]], \alpha_2 \mapsto [\alpha_5], \alpha_3 \mapsto [[\alpha_5]] \rightarrow [[\alpha_5]], \alpha_4 \mapsto [\alpha_5]\} \text{ is} \\
 \text{the solution of } \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5] \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}
 \end{array}$$

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

## Non-Termination of the Iterative Typing (4)

hsrcm\*

Conjecture: The iterative typing does not terminate

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

$$\begin{array}{c}
 \text{(AXC)} \frac{}{\Gamma_i \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \text{(AXSC)} \frac{}{\Gamma_i \vdash \mathbf{g} :: [\alpha_5]^i} \\
 \text{(RAPP)} \frac{}{\Gamma_i \vdash (\text{Cons } \mathbf{g}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3\}}, \text{(AXC)} \frac{}{\Gamma_i \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{\Gamma_i \vdash [\mathbf{g}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SCREC)} \frac{}{\Gamma_i \vdash_T \mathbf{f} :: \sigma(\alpha_1) = [[\alpha_5]^i]} \\
 \sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \rightarrow [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\} \text{ is} \\
 \text{the solution of } \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}
 \end{array}$$

$$\begin{array}{c}
 \text{(AXC)} \frac{}{\Gamma_i \vdash \text{Cons} :: \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4], \emptyset}, \text{(AXSC)} \frac{}{\Gamma_i \vdash \mathbf{f} :: [\alpha_5]^i} \\
 \text{(RAPP)} \frac{}{\Gamma_i \vdash (\text{Cons } \mathbf{f}) :: \alpha_3, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3\}}, \text{(AXC)} \frac{}{\Gamma_i \vdash \text{Nil} :: [\alpha_2], \emptyset} \\
 \text{(RAPP)} \frac{}{\Gamma_i \vdash [\mathbf{f}] :: \alpha_1, \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}} \\
 \text{(SCREC)} \frac{}{\Gamma_i \vdash_T \mathbf{g} :: \sigma(\alpha_1) = [[\alpha_5]^i]} \\
 \sigma = \{\alpha_1 \mapsto [[\alpha_5]^i], \alpha_2 \mapsto [\alpha_5]^i, \alpha_3 \mapsto [[\alpha_5]^i] \rightarrow [[\alpha_5]^i], \alpha_4 \mapsto [\alpha_5]^i\} \text{ is} \\
 \text{the solution of } \{\alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \doteq [\alpha_5]^i \rightarrow \alpha_3, \alpha_3 \doteq [\alpha_2] \rightarrow \alpha_1\}
 \end{array}$$

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

Thus ...

hsrcm\*

### 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)

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

A typed program calculus fulfills [type safety](#) iff

- Typing is preserved by reduction ([type preservation](#)):  
For monomorphic type  $\tau$ : If  $t :: \tau$  and  $t \rightarrow t'$ , 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](#))

**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:

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

**Lemma (Type Preservation)**

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

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

## Type Safety (4)

hsrm\*

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)

hsrm\*

### Theorem

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

## Forcing Termination of Type Inference

hsrm\*

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

**Milner-Step:** Type  $SC_1, \dots, SC_m$  together with the type assumption:

$\Gamma_M = \Gamma \cup \{SC_1 :: \tau_1, \dots, SC_m :: \tau_m\}$ ; **without quantifiers**  
and the following rule (SCRecMilnerStep) ...

## Forcing Termination (Cont'd)

hsrm\*

$$(\text{SCREC MILNER STEP}) \frac{\text{for } i = 1, \dots, m: \Gamma_M \cup \{x_{i,1} :: \alpha_{i,1}, \dots, x_{i,n_i} :: \alpha_{i,n_i}\} \vdash s_i :: \tau'_i, E_i}{\Gamma_M \vdash_T \text{ for } i = 1, \dots, m SC_i :: \sigma(\alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau'_i)}$$

if  $\sigma$  is the solution of  $E_1 \cup \dots \cup E_m \cup \bigcup_{i=1}^m \{\tau_i \doteq \alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau'_i\}$   
and  $SC_1 x_{1,1} \dots x_{1,n_1} = s_1$   
...  
 $SC_m x_{m,1} \dots x_{m,n_m} = s_m$   
are the definitions of  $SC_1, \dots, SC_m$

As additional typing rule we add:

$$(\text{AxSC2}) \frac{}{\Gamma \cup \{SC :: \tau\} \vdash SC :: \tau}$$

if  $\tau$  is not universally quantified

## Forcing Termination (Cont'd)

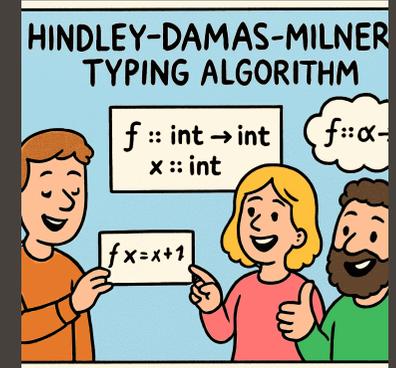
Differences to an iterative step:

- 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 (SCRECMILNERSTEP) 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  $\alpha_i$  (without quantifier!)
- consistency is enforced by additional unification equations

Haskell uses Hindley-Damas-Milner-typing

## The Hindley-Damas-Milner Type Inference Algorithm

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

- 1 Assumption  $\Gamma$  contains types of the already typed SCs and of the constructors (all universally quantified)
- 2 Type  $SC_1, \dots, SC_m$  with the rule (HDMSCREC):

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

if  $\sigma$  solution of  $E_1 \cup \dots \cup E_m \cup \bigcup_{i=1}^m \{\beta_i \doteq \alpha_{i,1} \rightarrow \dots \rightarrow \alpha_{i,n_i} \rightarrow \tau_i\}$   
 and  $SC_1 \ x_{1,1} \ \dots \ x_{1,n_1} = s_1$  are the definitions of  $SC_1, \dots, SC_m$   
 $\dots$   
 $SC_m \ x_{m,1} \ \dots \ x_{m,n_m} = s_m$

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

Simplification: Rule for one single recursive supercombinator:

$$\text{(HDMSCREC1)} \frac{\Gamma \cup \{SC :: \beta, x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash s :: \tau, E}{\Gamma \vdash_T SC :: \sigma(\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau)}$$

if  $\sigma$  is the solution of  $E \cup \{\beta \doteq \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \tau\}$   
 and  $SC \ x_1 \ \dots \ x_n = s$  is the definition of  $SC$

- 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 according to the hierarchy

Example

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

```

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

$$\Gamma_0 = \{\text{Cons} :: \forall a.a \rightarrow [a] \rightarrow [a], \text{Nil} :: \forall a.[a]\}$$

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

$$\text{(RCASE)} \frac{\begin{array}{l} (a) \Gamma \vdash xs :: \tau_1, E_1 \\ (b) \Gamma \vdash \text{Nil} :: \tau_2, E_2 \\ (c) \Gamma' \vdash (\text{Cons } y \text{ } ys) :: \tau_3, E_3 \\ (d) \Gamma \vdash \text{Nil} :: \tau_4, E_4 \\ (e) \Gamma' \vdash (\text{Cons } (f \ y) \ (\text{map } f \ ys)) :: \tau_5, E_5 \end{array}}{\Gamma \vdash \text{case } xs \text{ of } \{\text{Nil} \rightarrow \text{Nil}; \text{Cons } y \text{ } ys \rightarrow \text{Cons } y \ (\text{map } f \ ys)\} :: \alpha, E}$$

$$\text{(HDMSCREC1)} \frac{\Gamma \vdash_T \text{map} :: \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha)}{\Gamma \vdash_T \text{map} :: \sigma(\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha)}$$

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

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)

hsrm\*

(a)  $\frac{(\text{AxV})}{\Gamma \vdash xs :: \alpha_2, \emptyset}$   
 i.e.  $\tau_1 = \alpha_2$  and  $E_1 = \emptyset$ .

(b)  $\frac{(\text{AxC})}{\Gamma \vdash \text{Nil} :: [\alpha_5], \emptyset}$   
 i.e.  $\tau_2 = [\alpha_5]$  and  $E_2 = \emptyset$

(c)  $\frac{(\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash y :: \alpha_3, \emptyset} \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash \text{Cons} :: \alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6]}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{(\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (\text{Cons } y) :: \alpha_7, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7\}}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{(\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (\text{Cons } y \text{ } ys) :: \alpha_8, \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7, \alpha_7 \dot{=} \alpha_4 \rightarrow \alpha_8\}}}$   
 i.e.  $\tau_3 = \alpha_8$  and  $E_3 = \{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7, \alpha_7 \dot{=} \alpha_4 \rightarrow \alpha_8\}$

(d)  $\frac{(\text{AxV})}{\Gamma \vdash \text{Nil} :: [\alpha_9], \emptyset}$   
 i.e.  $\tau_4 = [\alpha_9]$  and  $E_4 = \emptyset$ .

## Example: map (3)

hsrm\*

(e)  $\frac{(\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash f :: \alpha_1, \emptyset}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash y :: \alpha_3, \emptyset}, \quad (\text{AxSC2}) \frac{(\text{AxSC2})}{\Gamma' \vdash \text{map} :: \beta, \emptyset}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash f :: \alpha_1, \emptyset}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{(\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash \text{Cons} :: \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}], \emptyset}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (f \ y) :: \alpha_{15}, \{\alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}\}}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (\text{map } f) :: \alpha_{12}, \{\beta \dot{=} \alpha_1 \rightarrow \alpha_{12}\}}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma' \vdash ys :: \alpha_4, \emptyset}}{(\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (\text{Cons } (f \ y)) :: \alpha_{11}, \{\alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}\}}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (\text{map } f \ ys) :: \alpha_{13}, \{\beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}}}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma' \vdash (\text{Cons } (f \ y) \ (\text{map } f \ ys)) :: \alpha_{14}, \{\alpha_{11} \dot{=} \alpha_{13} \rightarrow \alpha_{14}, \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}, \beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}}}$

i.e.  $\tau_5 = \alpha_{14}$  and  
 $E_5 = \{\alpha_{11} \dot{=} \alpha_{13} \rightarrow \alpha_{14}, \alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}, \beta \dot{=} \alpha_1 \rightarrow \alpha_{12}, \alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}\}$

## Example: map (4)

hsrm\*

Unify equations  $E \cup \{\beta \dot{=} \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha\}$ :

$\{\alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \dot{=} \alpha_3 \rightarrow \alpha_7, \alpha_7 \dot{=} \alpha_4 \rightarrow \alpha_8, \alpha_{11} \dot{=} \alpha_{13} \rightarrow \alpha_{14},$   
 $\alpha_{10} \rightarrow [\alpha_{10}] \rightarrow [\alpha_{10}] \dot{=} \alpha_{15} \rightarrow \alpha_{11}, \alpha_1 \dot{=} \alpha_3 \rightarrow \alpha_{15}, \beta \dot{=} \alpha_1 \rightarrow \alpha_{12},$   
 $\alpha_{12} \dot{=} \alpha_4 \rightarrow \alpha_{13}, \alpha_2 \dot{=} [\alpha_5], \alpha_2 \dot{=} \alpha_8, \alpha \dot{=} \alpha_9, \alpha \dot{=} \alpha_{14},$   
 $\beta \dot{=} \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha\}$

Unification results in

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

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

## Examples Known from Iterative Typing

hsrm\*

$g \ x = x : (g \ (g \ 'c'))$

Iterative typing results in Fail (after multiple iterations)

Hindley-Damas-Milner:  $\Gamma = \{\text{Cons} :: \forall a. a \rightarrow [a] \rightarrow [a], 'c' :: \text{Char}\}$ .

Let  $\Gamma' = \Gamma \cup \{x :: \alpha, g :: \beta\}$ .

$\frac{(\text{AxV}) \frac{(\text{AxV})}{\Gamma \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma \vdash x :: \alpha, \emptyset}, \quad (\text{AxSC2}) \frac{(\text{AxSC2})}{\Gamma \vdash g :: \beta, \emptyset}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma \vdash (g \ 'c') :: \alpha_7, \{\beta \dot{=} \text{Char} \rightarrow \alpha_7\}}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma \vdash 'c' :: \text{Char}, \emptyset}, \quad (\text{AxV}) \frac{(\text{AxV})}{\Gamma \vdash g :: \beta, \emptyset}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma \vdash (g \ (g \ 'c')) :: \alpha_4, \{\beta \dot{=} \text{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4\}}}{(\text{RAPP}) \frac{(\text{RAPP})}{\Gamma \vdash (\text{Cons } x) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha \rightarrow \alpha_3}, \quad (\text{RAPP}) \frac{(\text{RAPP})}{\Gamma \vdash (g \ (g \ 'c')) :: \alpha_4, \{\beta \dot{=} \text{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4\}}}, \quad (\text{HDMSCRec}) \frac{(\text{HDMSCRec})}{\Gamma \vdash \text{Cons } x \ (g \ (g \ 'c')) :: \alpha_2, \{\beta \dot{=} \text{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2\}}}, \quad (\text{HDMSCRec}) \frac{(\text{HDMSCRec})}{\Gamma \vdash_T g :: \sigma(\alpha \rightarrow \alpha_2)}, \quad \text{where } \sigma \text{ is the solution of}$

$\{\beta \dot{=} \text{Char} \rightarrow \alpha_7, \beta \dot{=} \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \dot{=} \alpha \rightarrow \alpha_3, \alpha_3 \dot{=} \alpha_4 \rightarrow \alpha_2, \beta \dot{=} \alpha \rightarrow \alpha_2\}$

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

## Examples Known from Iterative Typing (2)



$g\ x = 1 : (g\ (g\ 'c'))$

Iterative type:  $g :: \forall \alpha. \alpha \rightarrow [\text{Int}]$

Hindley-Damas-Milner: Let  $\Gamma' = \Gamma \cup \{x :: \alpha, g :: \beta\}$ .

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\Gamma \vdash \text{Cons} :: \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5], \emptyset}{(\text{AxSC2})}, \frac{\Gamma \vdash 1 :: \text{Int}, \emptyset}{(\text{AxSC2})}}{\Gamma \vdash (\text{Cons } 1) :: \alpha_3, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3}{(\text{RApP})}, \frac{\frac{\frac{\Gamma \vdash g :: \beta, \emptyset}{(\text{AxSC2})}, \frac{\Gamma \vdash (g\ 'c') :: \alpha_7, \{\beta \doteq \text{Char} \rightarrow \alpha_7\}}{(\text{AxC})}}{\Gamma \vdash (g\ (g\ 'c')) :: \alpha_4, \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4\}}{(\text{RApP})}}{\Gamma \vdash \text{Cons } 1\ (g\ (g\ 'c')) :: \alpha_2, \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2\}}{(\text{SCRec})}}{\Gamma \vdash_T g :: \sigma(\alpha \rightarrow \alpha_2)} \text{ where } \sigma \text{ is the solution of } \{\beta \doteq \text{Char} \rightarrow \alpha_7, \beta \doteq \alpha_7 \rightarrow \alpha_4, \alpha_5 \rightarrow [\alpha_5] \rightarrow [\alpha_5] \doteq \text{Int} \rightarrow \alpha_3, \alpha_3 \doteq \alpha_4 \rightarrow \alpha_2, \beta \doteq \alpha \rightarrow \alpha_2\}$$

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

## Iterative Typing May Return More General Types



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

Types of the constructors

Empty ::  $\forall a. \text{Tree } a$  and

Node ::  $\forall a. a \rightarrow \text{Tree } a \rightarrow \text{Tree } a \rightarrow \text{Tree } a$

$g\ x\ y = \text{Node True } (g\ x\ y)\ (g\ y\ x)$

Hindley-Damas-Milner:  $g :: a \rightarrow a \rightarrow \text{Tree Bool}$

Iterative Typing:  $g :: a \rightarrow b \rightarrow \text{Tree Bool}$

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 Haskell:

```
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 preserves the iterative type